

Supercomputing Frontiers and Innovations

2021, Vol. 8, No. 2

Scope

- Future generation supercomputer architectures
- Exascale computing
- Parallel programming models, interfaces, languages, libraries, and tools
- Supercomputer applications and algorithms
- Novel approaches to computing targeted to solve intractable problems
- Convergence of high performance computing, machine learning and big data technologies
- Distributed operating systems and virtualization for highly scalable computing
- Management, administration, and monitoring of supercomputer systems
- Mass storage systems, protocols, and allocation
- Power consumption minimization for supercomputing systems
- Resilience, reliability, and fault tolerance for future generation highly parallel computing systems
- Scientific visualization in supercomputing environments
- 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 Hoefler**, 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 Tyrtyshnikov**, 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

Special Issue on Advance Methods and Technologies on Vector Computing and Data-Processing Using NEC SX-Aurora TSUBASA	
H. Kobayashi, S. Momose	4
Accelerating Seismic Redatuming Using Tile Low-Rank Approximations on NEC SX-Aurora TSUBASA	
Y. Hong, H. Ltaief, M. Ravasi, L. Gatineau, D. Keyes	6
Porting and Optimizing Molecular Docking onto the SX-Aurora TSUBASA Vector Computer	
L. Solis-Vasquez, E. Focht, A. Koch	27
First Experience of Accelerating a Field-Induced Chiral Transition Simulation Using the SX-Aurora TSUBASA	
S. Yoshida, A. Endo, H. Kaneyasu, S. Date	43
Evaluating the Performance of OpenMP Offloading on the NEC SX-Aurora TSUBASA Vector Engine	
T. Cramer, B. Kosmynin, S. Moll, M. Römmmer, E. Focht, M.S. Müller	59
Performance and Power Analysis of a Vector Computing System	
K. Komatsu, A. Onodera, E. Focht, S. Fujimoto, Y. Isobe, S. Momose, M. Sato, H. Kobayashi	75
Distributed Graph Algorithms for Multiple Vector Engines of NEC SX-Aurora TSUBASA Systems	
I.V. Afanasyev, Vad.V. Voevodin, K. Komatsu, H. Kobayashi	95
Optimizing Load Balance in a Parallel CFD Code for a Large-scale Turbine Simulation on a Vector Supercomputer	
O. Watanabe, K. Komatsu, M. Sato, H. Kobayashi	114



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.

Guest Editors' Introduction for Special Issue on
**Advance Methods and Technologies on Vector Computing
and Data-Processing Using NEC SX-Aurora TSUBASA**

Hiroaki Kobayashi, Tohoku University
Shintaro Momose, NEC Corporation

Contemporary high-performance computing is based on diverse architectures. Some of them like using heterogeneous systems have become trends of nowadays, but some are not so widely used, still providing significant advantage in a number of use cases. Vector machines are one of those. The key principles of such systems have a long history, and some of these principles implementations could be definitely welcome to the hall of fame of supercomputing, if one existed. This issue is devoted to the latest achievements in the field of development and using vector machines, having a special focus on NEC SX-Aurora TSUBASA.

SX-Aurora TSUBASA is the most recent generation of NEC's supercomputer having vector processors. The vector processor is implemented onto a PCIe card as Vector Engine (VE) and it works on an x86/Linux environment. The VE processor uses a combination of powerful vector cores and high-bandwidth memory and it executes whole application code on VE with avoiding frequent transactions between VE and X86 host processors. This allows SX-Aurora TSUBASA to achieve high sustained performance even in applications that require high memory performance, which significantly extends the class of problems, suitable for this architecture.

The objective of this special issue is to allow researchers working with SX-Aurora TSUBASA vector architecture to present and discuss methodologies, approaches and solutions of using the potential of vector data processing both for the design of promising high-performance computing systems and for development of various classes of applications. Seven papers have been selected for this special issue.

Yuxi Hong, Hatem Ltaief, Matteo Ravasi, Laurent Gatineau, David Keyes in their work "Accelerating Seismic Redatuming Using Tile Low-Rank Approximations on SX-Aurora TSUBASA" report on accelerating expensive operators by using Tile Low-Rank (TLR) approximations to perform one of the most time-consuming computational kernels, i.e., the Matrix-Vector Multiplication (MVM) operation, on the NEC vector computing SX-Aurora TSUBASA hardware solution.

Leonardo Solis-Vasquez, Erich Focht, Andreas Koch in "Porting and Optimizing Molecular Docking onto the SX-Aurora TSUBASA Vector Computer" present their methodology for efficiently porting and optimizing AutoDock onto the SX-Aurora TSUBASA.

Shinji Yoshida, Arata Endo, Hirono Kaneyasu, Susumu Date in "First experience of accelerating a field-induced chiral transition simulation using the SX-Aurora TSUBASA" show how acceleration of the FICT (field-induced chiral transition simulation) is achieved as well as how much the efforts of users is required.

Tim Cramer, Boris Kosmynin, Simon Moll, Manoel Rmmer, Erich Focht, Matthias S. Mller in "Evaluating the Performance of OpenMP Offloading on the NEC SX-Aurora TSUBASA Vector Engine" give scientific programmers new opportunities and flexibilities for the development of scalable OpenMP offloading applications for SX-Aurora TSUBASA.

Kazuhiro Komatsu, Akito Onodera, Erich Focht, Soya Fujimoto, Yoko Isobe, Shintaro Momose, Masayuki Sato, Hiroaki Kobayashi in "Performance and Power Analysis of a Vector Com-

puting System” discuss the potential through the optimization of two benchmarks, the Himeno and HPCG benchmarks, for the latest vector computing system SX-Aurora TSUBASA.

Ilya Victorovich Afanasyev, Kazuhiko Komatsu, Hiroaki Kobayashi, Vadim Voevodin in their paper “Distributed Graph Algorithms for Multiple Vector Engines of NEC SX-Aurora TSUBASA Systems” describe distributed implementations of three widely-used graph algorithms: Page Rank (PR), Bellman-Ford Single Source Shortest Paths (further referred as SSSP) and Hyperlink-Induced Topic Search (HITS), evaluating their performance and scalability on SX-Aurora TSUBASA 8 system.

Osamu Watanabe, Kazuhiko Komatsu, Masayuki Sato, Hiroaki Kobayashi in “Optimizing Load Balance in a Parallel CFD Code for a Large-scale Turbine Simulation on a Vector Supercomputer” give an overview of the Numerical Turbine code and characteristics of the code in MPI parallel execution and propose a method for reducing load imbalance using hybrid parallelization.

These works give a clear vision of the architecture capabilities and demonstrate some strong use cases for real life applications. We thank authors and reviewers for their contributions to this special issue and wish all the teams further frontier achievements in the rapidly advancing field of high-performance computing.

We also thank Prof. Voevodin for providing us with the opportunity of this special issue and Dr. Nikitenko for his assist to make this issue a success. We hope readers will enjoy this special issue and find the potential of the modern vector system.

Accelerating Seismic Redatuming Using Tile Low-Rank Approximations on NEC SX-Aurora TSUBASA

Yuxi Hong¹, Hatem Ltaief¹, Matteo Ravasi¹, Laurent Gatineau², David Keyes¹

© The Authors 2021. This paper is published with open access at SuperFri.org

With the aim of imaging subsurface discontinuities, seismic data recorded at the surface of the Earth must be numerically re-positioned inside the subsurface where reflections have originated, a process referred to as redatuming. The recently developed Marchenko method is able to handle full-wavefield data including multiple arrivals. A downside of this approach is that a multi-dimensional convolution operator must be repeatedly evaluated to solve an expensive inverse problem. As such an operator applies multiple dense matrix-vector multiplications (MVM), we identify and leverage the data sparsity structure for each frequency matrix and propose to accelerate the MVM step using tile low-rank (TLR) matrix approximations. We study the TLR impact on time-to-solution for the MVM using different accuracy thresholds whilst at the same time assessing the quality of the resulting subsurface seismic wavefields and show that TLR leads to a minimal degradation in terms of signal-to-noise ratio on a 3D synthetic dataset. We mitigate the load imbalance overhead and provide performance evaluation on two distributed-memory systems. Our MPI+OpenMP TLR-MVM implementation reaches up to 3X performance speedup against the dense MVM counterpart from NEC scientific library on 128 NEC SX-Aurora TSUBASA cards. Thanks to the second generation of high bandwidth memory technology, it further attains up to 67X performance speedup compared to the dense MVM from Intel MKL when running on 128 dual-socket 20-core Intel Cascade Lake nodes with DDR4 memory. This corresponds to 110 TB/s of aggregated sustained bandwidth for our TLR-MVM implementation, without suffering deterioration in the quality of the reconstructed seismic wavefields.

Keywords: seismic redatuming, tile low-rank approximations, matrix-vector multiplication, load balancing, high bandwidth memory, NEC SX-Aurora TSUBASA.

Introduction

Exploration geophysics is an applied branch of geophysics that uses several physical measurements at the surface of the Earth (e.g., seismic, gravity, electromagnetic) to estimate the physical properties of the first few kilometers of the subsurface. Originally developed with the aim of mapping anomalies corresponding to mineral or hydrocarbon accumulations, these methods are nowadays also used in the context of geothermal exploration, carbon capture, and storage evaluation and monitoring, as well as to assess the integrity of the near subsurface for offshore wind farms.

Seismic reflection is a popular remote sensing technique that utilizes reflected seismic waves to produce high-resolution images of the geological structures as well as estimates of elastic properties of the subsurface. Its success is motivated by the fact that the waves propagating in the subsurface and being recorded at the surface of the Earth are governed by the well known elastic wave equation; various techniques have been developed during the years to harness the information contained in such recordings – see [35] for a detailed treatise. With the aim of imaging subsurface discontinuities, seismic data recorded at the surface of the Earth must be numerically re-positioned at locations in the subsurface where reflections have originated, a process generally referred to as *redatuming* by the geophysical community [7]. Historically, this

¹King Abdullah University of Science and Technology, Kingdom of Saudi Arabia

²NEC Deutschland GmbH, HPC Division

process has been carried out by numerically time-reversing the data recorded along an open boundary of surface receivers into the subsurface. Despite its simplicity, such an approach is only able to handle seismic energy from primary arrivals (i.e., waves that interact only once with the medium discontinuities) failing to explain multi-scattering in the subsurface. As a result, seismic images are contaminated by artificial reflectors if data are not pre-processed prior to imaging such that multiples are removed from the data. In the last decade, a novel family of methods has emerged under the name of *Marchenko redatuming* [12, 30, 33]. Such methods allow for accurate redatuming of the full-wavefield recorded seismic data including multiple arrivals. This is achieved by solving an inverse problem, the adjoint modeling of which can be shown to be equivalent to the standard single-scattering redatuming method of [7]. Whilst being more accurate, this new approach calls for solution of an inverse problem that requires repeated application of the so-called multi-dimensional convolution (MDC) operator and its adjoint. Mostly because of the extremely expensive nature of these operators in terms of complexity and memory footprint, the *Marchenko redatuming* method has not been widely adopted by the geophysical community yet. It also shows poor scalability due to their inherent memory-bound behavior.

In this paper, we report on accelerating these expensive operators by using Tile Low-Rank (TLR) approximations to perform one of the most time-consuming computational kernels, i.e., the Matrix-Vector Multiplication (MVM) operation, on the NEC vector computing SX-Aurora TSUBASA hardware solution. In fact, MVM is the workhorse of *Marchenko redatuming* for seismic imaging since it is repeatedly applied for hundreds of frequencies at each step of the iterative process. Instead of operating the MVM on the original dense data structure, our numerical technique consists in (1) splitting it into tiles with elements contiguously stored in memory, (2) compressing the tile matrix using TLR approximations (e.g., using randomized SVD [21]) up to an application-dependent accuracy threshold, and (3) performing the MVM directly on the compressed TLR data storage. This translates into a reduction of the number of floating-point operations, while further saving memory footprint. The latter is especially critical when working with large 3D seismic datasets. This TLR algorithmic redesign of the MVM introduces load imbalance when processing low and high frequencies that does not occur with the traditional dense MVM. Indeed, TLR matrices associated with high frequencies reveal higher ranks than those from low frequencies. We design and implement a load balancing technique to map processing units into frequencies so that the overall application's idle time is limited. Our MPI+OpenMP TLR-MVM implementation saturates the second generation of high bandwidth memory (HBM2) from the SX-Aurora TSUBASA cards and maintains a decent scalability when increasing the number of vector engines. We assess the accuracy of TLR-MVM and demonstrate its numerical robustness on representative 3D seismic datasets. We benchmark our TLR-MVM on two distributed-memory systems. It reaches up to 3X performance speedup against the dense MVM counterpart from NEC scientific library on 128 NEC SX-Aurora TSUBASA cards. Thanks to HBM2, it further attains up to 67X performance speedup in time compared to the dense MVM from Intel MKL (i.e., the CGEMV kernel) when running on 128 dual-socket 20-core Intel Cascade Lake nodes with DDR4 memory. This corresponds to 110 TB/s of aggregated sustained bandwidth for our TLR-MVM implementation.

The contributions of this paper are as follows. We democratize the *Marchenko redatuming* method for seismic imaging simulations by integrating TLR-MVM as the core computational engine for solving the inverse problem. We conduct performance profiling of our TLR-MVM code

using NEC *Ftrace* profiler tool and identify hot spots and room for improvement. In particular, we mitigate the load imbalance engendered by TLR-MVM when operating matrices from several frequencies in an embarrassingly parallel fashion. We highlight the performance advantage of TLR-MVM over the traditional dense MVM on Intel x86 and NEC vector computing hardware solution, without suffering deterioration in the image quality. We evaluate our implementation using the roofline model [34] and show how TLR-MVM is able to leverage HBM2 technology.

The remainder of the paper is as follows. Section 1 presents related work on low-rank matrix approximations. Section 2 describes the seismic *Marchenko redatuming* method. Section 3 recalls the general TLR-MVM algorithm. Section 4 provides implementation details of multiple inlined TLR-MVM calls and introduces the necessary load balancing strategies. Section 5 shows the TLR impact on numerical accuracy using proxy 3D seismic datasets. Section 6 reports on performance analysis and experimental results of TLR-MVM on two distributed-memory systems composed of x86 and vector computing architectures. Section 7 discusses potential future work along this herein research direction, and we conclude in Section 7.

1. Related Work

Low-rank matrix approximation is a class of algebraic compression methods, that permits to exploit the data sparsity of large matrices. This becomes critical when performing linear algebra operations on these large operators since the algorithmic complexity and the memory footprint can be reduced [9, 18].

While the literature is rich in the theory of low-rank approximations, e.g., hierarchical matrix (\mathcal{H} -matrix) low-rank format [17, 20], supporting weak [16, 31] and strong [8] admissibility, or flat tile low-rank (TLR) matrix approximations [5], there exist only a few works on HPC implementations targeting x86 [2–4, 13, 24] and GPU hardware accelerators [10, 14, 23].

For instance, in the context of wave-equation-based seismic processing methods, the estimation of primaries by sparse inversion [22] (EPSI) suggests that low-rank approximations of the integral operators may be utilized to reduce the storage and computation cost of the MVM. This work is however only applied as a proof-of-concept to 2D datasets using the Hierarchical Semi-Separable (HSS) compression data format. While HSS provides linear complexity, it may face challenges in compressing 3D datasets resulting in an increase of the arithmetic complexity. Moreover, one of the main reasons that slows down the wide adoption of low-rank approximations in scientific applications on current petascale supercomputers is the lack of support for advanced numerical kernels from the vendor numerical libraries. Indeed, \mathcal{H} -matrix computations require the development of new kernels that are versatile enough to effectively support a range of arithmetic intensity, while exhibiting low overheads during kernel launch. On the contrary, flat TLR matrix approximations is a pragmatic approach, which represents a compromise between algorithmic complexity and software development/deployment on emerging HPC platforms.

Referred as batched matrix operations [1, 10, 15], the idea behind these advanced numerical kernels is to simultaneously execute many linear algebra kernels accessing different matrices so that one may achieve high hardware occupancy. While the support from optimized vendor libraries has improved over the last few years, developers may still have to implement their own kernels (e.g., on GPUs) or simply fall back to the *OpenMP for loop* pragma to execute kernels in batched mode. The former raises concerns on software sustainability while the latter may not extract performance of the underlying hardware architecture.

The authors in [26] have introduced the algorithm of a single TLR-MVM to enhance real-time performance when identifying the atmospheric turbulence for ground-based telescopes. Based on batched MVM, their TLR-MVM implementations rely on OpenMP due to standardization constraints required in the computational astronomy community for code sustainability and portability purposes. Performance results have been reported on several cutting-edge architectures.

In this paper, we extend this previous work [26] to process multiple TLR-MVM (i.e., a batch of batched MVM) required by the seismic redatuming method and deploy the application to distributed-memory systems equipped with x86 nodes and vector computing engines. Given the few but strong cores (i.e., eight cores) on NEC SX-Aurora TSUBASA cards compared to x86 architectures, NEC hardware solution makes up the low core count with vectorizations and high bandwidth memory (HBM2) to achieve high performance. This is quite different than GPU architectures that promote massive parallelism via the single instruction, multiple data (SIMD) paradigm. Therefore, porting on NEC vector engines represents a similar effort than deploying on x86, i.e., with high user productivity, with the favorable exception that HBM2 may provide a significant performance boost to memory-bound kernels compared to x86's DDR4 memory technology. And to further maximize performance on NEC vector engines, it is also paramount to fully utilize the vector units. All in all, the hardware design of NEC SX-Aurora TSUBASA cards facilitates the deployment of these advanced numerical kernels, which intrinsically drives the performance of low-rank matrix approximations.

To our knowledge, this is the first time TLR-MVM is successfully applied to 3D datasets using NEC vector computing hardware solutions in the context of seismic redatuming.

2. Seismic Redatuming

Seismic redatuming is the process of numerically re-positioning seismic data physically recorded at the surface of the Earth to any location of interest in the subsurface. Whilst historically able to target only so-called primary arrivals in the recorded data, recent theoretical advances have led to the creation of so-called *Marchenko redatuming*, which is capable of handling full-wavefield seismic data including any order and type of internal scattering. This entails an inverse problem to be solved that can be expressed concisely as the following system of equations [27]:

$$\begin{bmatrix} \Theta R f_d^+ \\ 0 \end{bmatrix} = \begin{bmatrix} I & -\Theta R \\ -\Theta R^* & I \end{bmatrix} \begin{bmatrix} f^- \\ f_m^+ \end{bmatrix}, \quad (1)$$

where \mathbf{R} and \mathbf{R}^* are so-called convolution and correlation integral operators, Θ is a time-space window, \mathbf{I} is the identity operator, \mathbf{f}^- and \mathbf{f}_m^+ are the up-going and the coda of the down-going focusing functions to invert for, and \mathbf{f}_d^+ on the left-hand side of Eq. 1 is the direct component of the down-going focusing function that can be obtained by numerical modelling in a reference velocity medium. Finally, the overall down-going focusing function can be created as $\mathbf{f}^+ = \mathbf{f}_d^+ + \mathbf{f}_m^+$. For simplicity, Eq. 1 can be written compactly as $\mathbf{d} = \mathbf{M}\mathbf{f}$, where \mathbf{d} , \mathbf{f} and \mathbf{M} are the overall data, model, and Marchenko operator of the problem we wish to solve.

Once the focusing functions are retrieved, the up- and down-going separated Green's functions \mathbf{g}^- and \mathbf{g}^+ can be computed to be evaluating the following equations:

$$\begin{bmatrix} -\mathbf{g}^- \\ \mathbf{g}^{+*} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & -\mathbf{R} \\ -\mathbf{R}^* & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{f}^- \\ \mathbf{f}^+ \end{bmatrix}. \quad (2)$$

Note that due to the extremely large size of these matrices, whilst the problem is written in a compact matrix-vector formulation, its numerical solution is performed using matrix-free operators and iterative solvers such as conjugate gradient least-squares (CGLS) or LSQR [29]. Focusing now our attention on the multi-dimensional convolution (MDC) integral operator, which represents the most expensive computations in the overall chain of operations, its inner working can be written more explicitly as follows:

$$\mathbf{y} = \mathbf{Rx} : \quad y(t, \mathbf{x}_B, \mathbf{x}_A) = \mathcal{F}_{\omega_{max}}^{-1} \left(\int_{\delta\mathbb{D}} R(\omega, \mathbf{x}_B, \mathbf{x}_R) \mathcal{F}_{\omega_{max}}(x(t, \mathbf{x}_R, \mathbf{x}_A)) d\mathbf{x}_R \right). \quad (3)$$

Similarly, the adjoint of such an operator can be written as:

$$\mathbf{x} = \mathbf{R}^H \mathbf{y} : \quad x(t, \mathbf{x}_R, \mathbf{x}_A) = \mathcal{F}_{\omega_{max}}^{-1} \left(\int_{\delta\mathbb{D}} R^*(\omega, \mathbf{x}_B, \mathbf{x}_R) \mathcal{F}_{\omega_{max}}(y(t, \mathbf{x}_B, \mathbf{x}_A)) d\mathbf{x}_B \right), \quad (4)$$

where \mathcal{F} and \mathcal{F}^{-1} represent the forward and inverse Fourier transforms, ω is the angular frequency, \mathbf{x}_A , \mathbf{x}_B and \mathbf{x}_R represent spatial locations with the latter two spanning the integration domain $\delta\mathbb{D}$. ω_{max} is used to indicate that the output of the forward Fourier transform is truncated to contain only frequencies where the signal spectrum resides. Finally, $R(\omega, \mathbf{x}_B, \mathbf{x}_R)$ represents the kernel of the integral operator in the frequency-space domain and can be created upfront by applying the Fourier transform along the time axis of the physically recorded seismic data $R(t, \mathbf{x}_B, \mathbf{x}_R)$. Moreover, once the spatial integral is discretized, the kernel simply becomes a stack of matrices (one for each frequency ω within the specified frequency spectrum of the seismic data) and the integral can be interpreted as a batched matrix-vector multiplication (MVM) operations. This is true for both the forward and adjoint computations, with the main difference that the latter requires such matrices to be transposed and complex conjugated. Finally, in order to validate our statement that the operators \mathbf{R} and \mathbf{R}^H represent the most expensive computations in the solution of the inverse problem in Eq. 1, a single iteration of CGLS is evaluated and the overall computational time is divided into atomic contributions (Fig. 1). A single-node implementation of the Marchenko redatuming equations is used in this example as provided by the PyLops framework for large-scale inverse problem [28]. More specifically, since an iteration of CGLS requires the application of both a forward (\mathbf{M}) and adjoint (\mathbf{M}^H) passes, we observe that almost ninety percent of the time is spent on evaluating the \mathbf{R} (and \mathbf{R}^*) and their adjoints, while the remaining time percent of the time is roughly split between other computations involved in the \mathbf{M} operator and vector-vector operations in the CGLS step. Finally, we observe here a slight time difference in the computation of the forward and adjoint passes; this results from the transposition of the matrix stack in the adjoint step. Moreover, complex conjugation must be performed on each element of the kernel. In practice, as explained in more detail in [29], complex conjugation is applied to the input and output vectors, which are much smaller than the kernel, so the kernel itself is not transposed. Nevertheless, since the matrices in the stack are stored in a row-major order in main memory, the timing of the forward step is more favourable.

Alongside with advances in processing algorithms, the size and scale of seismic surveys have increased since the late 20th century. Nowadays, large-scale high-resolution 3D surveys

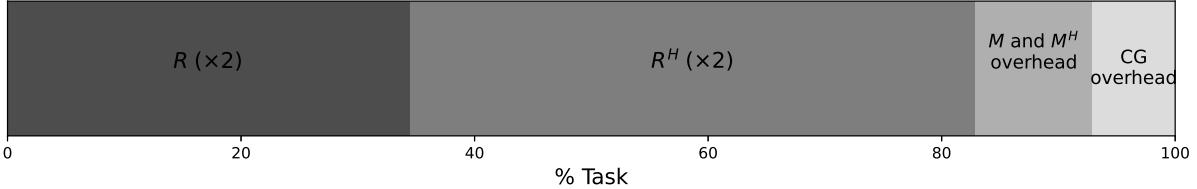


Figure 1. Task profiling for a single iteration of CGLS during the solution of the inverse problem in Eq. 1

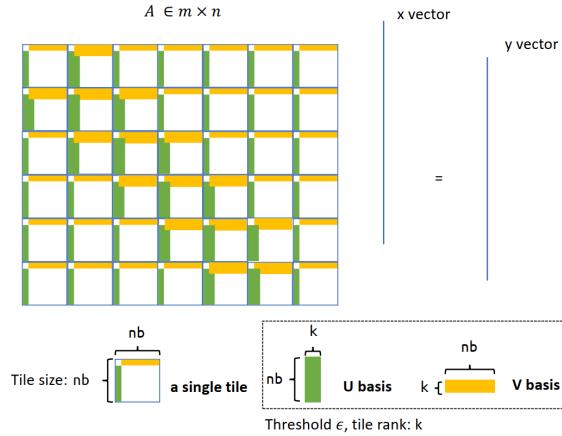
are routinely acquired where the recorded data can easily be on the order of several Terabytes. Recently, the implementation of the 3D Marchenko equations has been discussed in [29] and [11]. In both cases, special attention has been placed on the implementation of the integral operator and the handling of kernels that cannot directly fit in the main memory of single compute node. In the former approach, the embarrassingly parallel nature of the batched MVM is leveraged by reading different frequency batches in the main memory of multiple compute nodes only once prior, to solving the inverse problem in Eq. 1. The latter approach, on the other hand, utilizes the ZFP-based compression algorithm of [25] to reduce the size of the reflection response to be stored on disk and the read-in time to memory. Authors report a compression factor of four for this lossless compression when applied to their frequency-space reflection seismic data. In fact, even when the data is compressed, on-the-fly decompression is still required to be able to perform the computations in Eqs. 3 and 4. In both implementations, however, no attempt is made to expedite the dense MVM required in both the forward and adjoint processes.

Whilst the redatuming Eq. 1 is relatively new to the field of geophysics, integral operators of the kind of \mathbf{R} (Eq. 3) and \mathbf{R}^H (Eq. 4) are common to a number of other wave-equation-based seismic processing methods, such as surface-related multiple elimination (SRME – [32]), estimation of primaries by sparse inversion (EPSI – [19]), and up/down deconvolution [6] just to name a few. This work may therefore have a broader impact beyond the Marchenko redatuming technique.

3. Background on TLR-MVM

We briefly recall here the algorithmic design of the tile low-rank matrix-vector multiplication (TLR-MVM) kernel [26]. While the traditional dense MVM stores the matrix elements in column-major data layout format, TLR-MVM first splits the dense matrix into tiles in which elements are now contiguous in memory. This tiling technique is important in terms of memory access as it shortens the strided memory access to better fit in the high levels of the memory subsystem. Once the tile matrix data structure is constructed, TLR-MVM compresses each dense tile in an embarrassingly parallel fashion using an algebraic method of choice (e.g., rank-revealing QR, randomized SVD, etc.). The numerical lossy compression depends on the accuracy threshold required by the application to sustain its numerical robustness.

Figure 2 represents the TLR-MVM operation $A \times x = y$, with $A \in \mathbf{R}^{m \times n}, x \in \mathbf{R}^n, y \in \mathbf{R}^m$. The matrix A is split into 6-by-7 tiles with a tunable tile size parameter nb . After compression using the accuracy threshold ϵ , each tile is decomposed into U and V bases containing the k most significant singular values $\Sigma_{i,j}$ and their associated singular vectors, as defined by the following formula based on the Frobenius norm: $\|A_{i,j} - U_{i,j}^\epsilon \Sigma_{i,j}^{\epsilon} V_{i,j}^{T\epsilon}\|_F \leq \epsilon \|A\|_F$. The selected singular values $\Sigma_{i,j}$ may be absorbed by one of the bases after applying corresponding scaling

**Figure 2.** TLR-MVM data structure

operations, which facilitates the MVM implementation. Once compressed, the tiles may have different ranks k , which may create load imbalance situations.

Moreover, Level-2 BLAS MVM kernels are inherently memory-bound and their performance solely depends on the memory bandwidth. Therefore, it is critical to optimized memory access to avoid additional data motion between main memory and the last-level cache. While tiling technique helps for the traditional dense MVM, TLR-MVM ends up dealing with several new compressed U and V data structures that may not be stored contiguously in memory. As introduced in [26], we stack the U bases together and the V bases together and design a new MVM algorithm that leverages the TLR data structure.

The actual TLR-MVM operation can then proceed with the following three successive computational phases: (1) we multiply the stacked V bases to the vector x , (2) we project the result from phase 1 to the V bases, and (3) we multiply the stacked U bases with the result from the reshuffling phase 2 and compute the final result vector y .

In this paper, we extend the real precision arithmetic used for computational astronomy [26] to complex precision arithmetic in order to support seismic imaging applications. Furthermore, we implement a driver to launch several TLR-MVM kernels, i.e., one for each frequency ω , coming from the discretization of the spatial integral $R(\omega, \mathbf{x}_B, \mathbf{x}_R)$ (see Section 2), as explained in the next section.

4. Launching Multiple TLR-MVM Kernels for Seismic Redatuming

In this section, we describe the implementation of a driver that launches multiple TLR-MVM kernels to support the workload of seismic redatuming. We first identify the challenges introduced by TLR-MVM on 3D seismic datasets and propose optimization techniques to address them.

4.1. Challenges with 3D Seismic Datasets

The 3D seismic dataset used in this study contain stacked matrices for 150 frequencies. This is representative of the workload of seismic redatuming, although the number of frequencies may be further increased to include higher frequencies to produce seismic wavefields and images of higher resolution. The size of each matrix is $m = n = 9801$. The matrices considered herein

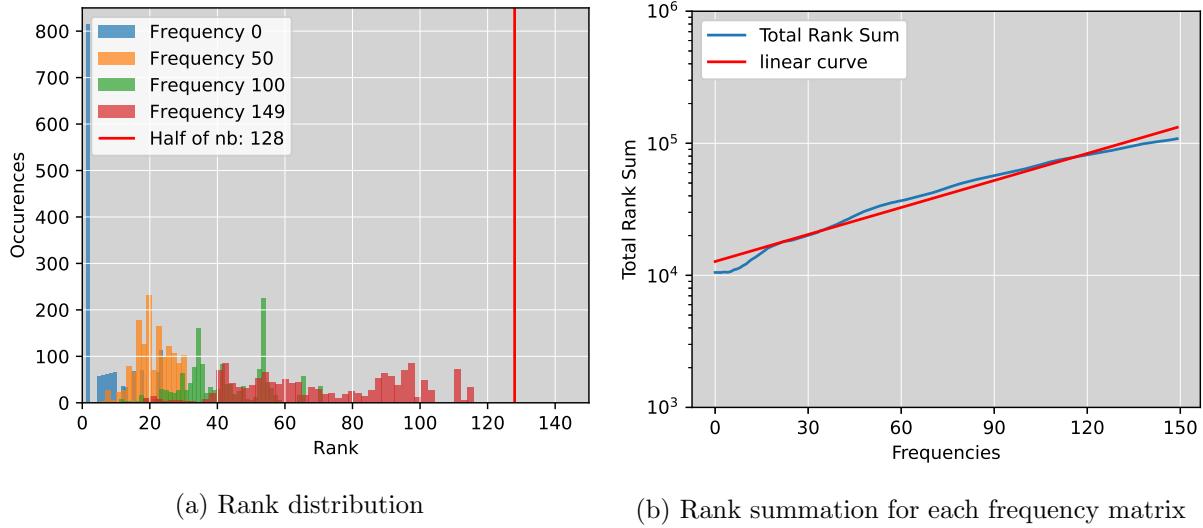


Figure 3. Rank analysis of the 3D seismic dataset using $nb = 256$ and $\epsilon = 0.001$

are square, but rectangular matrices are also supported. The relationship between the frequency index F_i and the value of the frequency is given by $f_{F_i} = \frac{i}{d_t n_t}$, where $d_t = 0.0025$ and $n_t = 1201$.

Figure 3 shows the rank analysis of the 3D seismic datasets. In particular, Fig. 3a highlights the rank distribution of F_i matrices for $i = 0, 50, 100$ and 149 . We set $nb = 256$ and $\epsilon = 0.001$. As expected, the figure captures how the rank distribution shifts to the right with higher ranks for matrices corresponding to higher frequency components of the data. The red vertical line in Fig. 3b shows the rank limit $nb/2 = 128$. If the rank distribution goes beyond this red line, the accumulated sizes of the bases for a single tile will be higher than the size of the original dense tile. On the contrary, if the rank distribution stays on the left of the red line, as pictured in Fig. 3a, TLR-MVM remains competitive compared to dense MVM. Figure 3b reports the summation for all ranks for a given frequency matrix. The total rank summation is a good metric to evaluate the algorithmic complexity. We observe an increase in rank for higher frequencies, which corroborates the analysis of the rank distribution in Fig. 3a. For this particular 3D seismic dataset, we can observe from Fig. 3b that the log-scale of the number of floating-point operations (FLOPS) of TLR-MVM has a near-linear relationship with the frequency matrix index. This relationship provides insights on how to orchestrate the TLR-MVM scheduling for all frequencies, given the workload heterogeneity. It is now clear that one of the main challenges is the load imbalance introduced by TLR-MVM within and across all frequency matrices, compared to the homogeneous dense MVM. We implement two optimizations techniques and present their corresponding pseudo-codes in Fig. 4. The codes are written in C and rely on MPI+OpenMP programming models.

We address in subsequent sections how these techniques mitigate the load imbalance overhead.

4.2. Merge-Phase Strategy for Intra-Node Load Balancing

Figure 4a pictures the *Merge-Phase* strategy (in orange color) proposed for our TLR-MVM reference implementation (in blue color).

The *Merge-Phase* strategy is designed to achieve intra-node load balancing by evenly distributing the computation on each thread (or processing units). Processing a collection of fre-

quencies F is not performed one by one anymore. Instead, the strategy fuses each individual phase across all frequencies. This increases the workload per OpenMP loops within each phase, engenders a larger amount of computational tasks, and reduces the scheduling overheads. The default static scheduling mode of operation may increase data locality, especially when dealing with small datasets. For large datasets, static scheduling may lose its performance advantage and may create idle time while work is available. We further enable the OpenMP dynamic scheduling so that the runtime has opportunities to prevent idle time situations by scheduling tasks as soon as they enter ready state. While this strategy alleviates the intra-node load imbalance bottleneck, the inter-node load imbalance remains an issue, as observed in Fig. 3b.

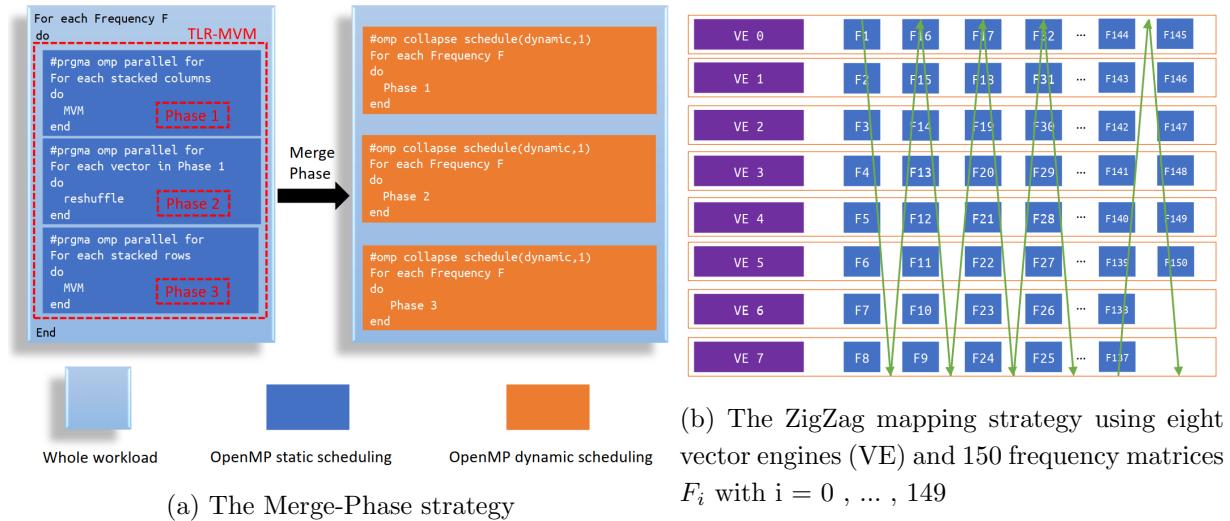


Figure 4. Pseudo-codes of the *Merge-Phase* and the *ZigZag* mapping strategy

4.3. ZigZag Mapping Strategy for Inter-Node Load Balancing

To leverage performance on distributed-memory systems, we evenly allocate frequency matrices across computational nodes. Dynamic load balancing on distributed-memory systems is a challenging approach that may require data movement to compensate for the idle time. However, we have identified some relationship between the frequency index and the corresponding FLOPS, as explained in Section 4.1. The *ZigZag* mapping strategy is used to achieve inter-node load balancing. We design the *ZigZag* mapping strategy to statically map frequencies to processing units and achieve inter-node load balancing. Figure 4b highlights the *ZigZag* pattern for frequency mapping using 150 frequencies on eight Vector Engines (VE).

In the first sweep of VEs, we map the frequencies in increasing index order. We continue mapping the next set of frequencies in decreasing index order for the VEs. We repeat the above *ZigZag* pattern until all frequencies are mapped to the VEs. This strategy has several advantages. Not only does it exploit the linear relationship between frequency index and FLOPS, it also balances the memory footprint per VE. For instance, given that the on-chip memory capacity is limited to 48 GB on NEC VEs, the *ZigZag* mapping strategy allows to scale memory-intensive applications. Furthermore, this mapping is performed offline and does not incur runtime overheads. Although it is important to mention that this linear relationship may not be characteristic of all seismic datasets. Network interconnect congestions may even further exacerbate the load

imbalance even if the load is properly distributed. Therefore, we believe dynamic load balancing is an interesting research direction and we leave it as future work.

4.4. Algorithmic Complexity and Code Balance

As discussed in [26], the number of floating-point operations (FLOPS) of the dense MVM is $2mn$ and the memory bandwidth can be calculated as $\frac{B(mn+n+m)}{t}$, with $B(W)$ the number of bytes of W elements (stored in single complex precision) and t the execution time. The FLOPS and memory bandwidth of TLR-MVM are $4K \times nb$ and $\frac{B(2K \times nb + 4K + m + n)}{t}$, respectively, with K the sum of the ranks across all tiles of any single frequency matrix (see Fig. 2) and nb the tile size.

In seismic application, suppose there are F frequency matrices, the overall FLOPS and memory bandwidth of dense MVM is $F \times 2mn$ and $F \times \frac{B(mn+n+m)}{t}$, respectively. The overall FLOPS and memory bandwidth of TLR-MVM for all frequency matrices F is $4K_F \times nb$ and $\frac{B(2K_F \times nb + 4K_F + m + n)}{t}$, where K_F is the sum of all ranks across all frequencies. According to the FLOPS calculation of TLR-MVM, the rank sum K of the frequency matrix plays an important role. If the rank sum K is not large enough, the algorithm may not saturate the memory bandwidth due to a suboptimal hardware occupancy.

5. Numerical Accuracy Assessment

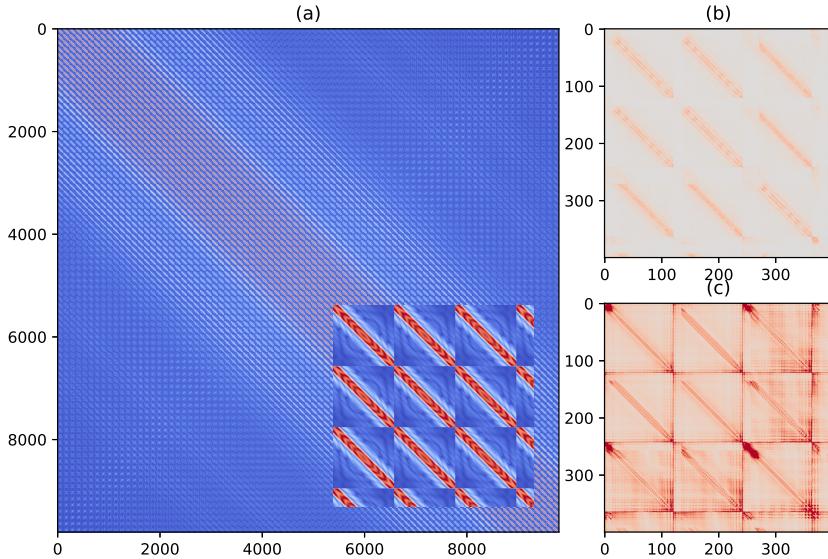


Figure 5. (a) Original Matrix of reflection response frequency slice at 33 Hz ($R(\omega = 33\text{Hz}, \mathbf{x}_B, \mathbf{x}_R)$, Insert is zoomed-in version on the first 400 rows and columns (b) and (c) show the same zoomed-in version of TLR compression error in (a), nb is tile size, ϵ is accuracy

To begin with, we consider the same synthetic geological model used in [29] (their Fig. 3) and compare the reconstruction of the total Green's function $\mathbf{g} = \mathbf{g}^+ + \mathbf{g}^-$ obtained using the original R kernel and the TLR compressed R kernel with different combinations of tile size and accuracy. The observation that seismic data in the frequency domain can be compressed in a low-rank form is not surprising when considering the form of matrices representing seismic data in the frequency domain as shown in Fig. 5a. Such a matrix represents the seismic data at a single frequency ($f = 33\text{ Hz}$) with sources placed along the rows and receivers along with the

columns. In other words, the element i, j of this complex-valued matrix contains the frequency at 33 Hz coefficient of the seismic recording at $i - th$ source and $j - th$ receiver. Looking at the insert in Fig. 5a, we can see how the entire matrix is composed of smaller diagonally dominant submatrices. This is due to the fact that we are dealing with 3D seismic data and each submatrix represents the responses of a single source line to a single receiver line. Moving from one source line to the next leads to the block pattern of this matrix in the row space, whilst moving from one receiver line to the next leads to the block pattern in the column space. Another important observation that was made by [22] is that within the available seismic bandwidth, kernel matrices at higher frequencies are of higher rank, i.e., require more basis functions to be approximated at the selected accuracy. Whilst not discussed by the authors in [22], this imbalance in the rank of the different matrices inevitably leads to a load imbalance in the computation of the batch matrix-vector multiplications (MVM), as explained in Section 4.1. The effect of solving the inverse problem in Eq. 1 using different TLR compressed R kernels is presented in Fig. 6. More specifically, Fig. 6a shows the reconstructed Green's function along eight different receiver lines, as shown in the insert using the original uncompressed reflection response. Such an estimate has been shown in [29] to be very accurate and is used here as our benchmark. Figures 6b-d show the reconstructed Green's function using TLR-MVM with different compression parameters. We choose $nb = 256, \epsilon = 0.001, 0.005, 0.01$. Figure 6e shows element-wise absolute value difference between Fig. 6a and Fig. 6b. Figure 6f is 10 \times of the value in Fig. 6e. Figures 6g-h are also 10 \times the element-wise absolute value difference with Fig. 6a using corresponding compression parameters.

We use Signal-to-Noise Ratio (SNR) to quantify the error shown in Fig. 6. The formula is $SNR = -20 * \log_{10} \frac{\|R_{org} - R_{approx}\|_2}{\|R_{approx}\|_2}$: the larger the SNR, the better the approximation.

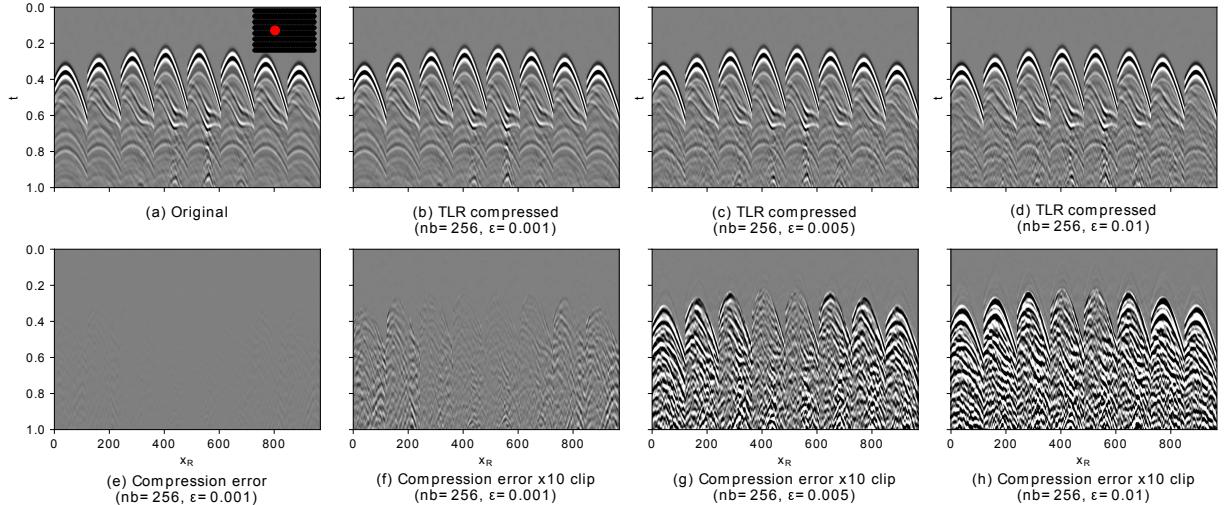


Figure 6. Green's function estimates (t^2 gain applied to all panels), with nb tile size and ϵ the accuracy threshold

Next, we investigate the impact of compression on the SNR for various tile sizes and accuracy thresholds, as shown in Fig. 7a. We compute the ratio between the FLOPS executed by dense MVM versus TLR-MVM. This ratio of FLOPS savings corresponds to how much fewer FLOPS are performed by TLR-MVM compared to dense MVM. For instance, TLR-MVM achieves an SNR around 30 while performing 2.4 fewer FLOPS than dense MVM when using $nb = 128$ and $\epsilon = 0.001$. We use colormap to show the corresponding SNR value. There is a general trend that one needs a more restrictive accuracy threshold in cases with smaller tile sizes. Figure 7b

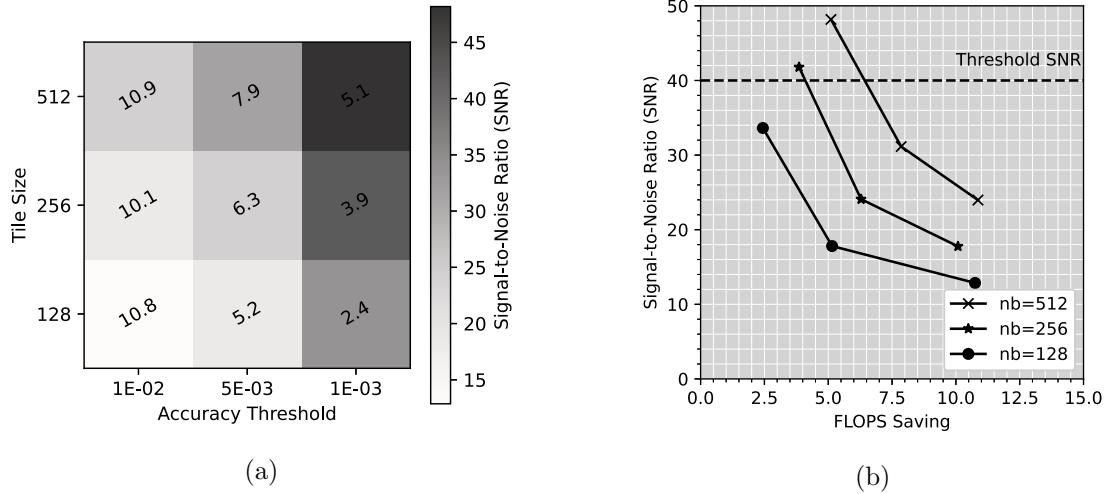


Figure 7. (a) Heat map plot of Signal-to-Noise Ratio (SNR), against tile size nb , accuracy threshold ϵ , and FLOPS saving (b) Line plot of SNR versus FLOPS savings for different tile sizes

shows the relationship of FLOPS savings with SNR. We choose a threshold SNR value of 40 that satisfies the quality requirement of the application. We observe two sets of compression parameters ($nb = 256, \epsilon = 0.001$ and $nb = 512, \epsilon = 0.001$), which satisfy this requirement. Indeed, under these two compression parameters, TLR-MVM algorithm does not affect the final image quality.

Once we complete the assessment on the numerical accuracy with the identification of this couple of sets of parameters, we can now study their impact on performance with the hardware systems studied in this paper.

6. Experimental Results

This section reports the performance results of our TLR-MVM implementation, compares it against dense MVM, and highlights the impact of the optimization techniques introduced in Sections 4.2 and 4.3.

6.1. Environment and Compilation Settings

The experiments are carried out on two x86 systems. The first system is a 16-node NEC B300-8 cluster, where each node is equipped with 8 NEC SX-Aurora TSUBASA-20B Vector Engines (VE). The memory capacity of each VE is 48GB. The second system is a single x86 node with a dual-socket 20-core Intel Cascade Lake. We refer to this shared-memory node to CSL in the subsequent sections. The memory capacity of the Intel server is 350 GB. The OS of both systems is Linux RedHat 7.7. We use OpenMPI as the MPI implementation. For the NEC server, we use NEC Compilers tools to compile the code and rely on the NEC Numeric Library Collection for the vendor optimized BLAS implementation. For the Intel server, we use Intel Parallel Studio 2019 to compile the code and link it against Intel Math Kernel Library for the vendor optimized BLAS implementation. We use `-fopenmp -O3` for both compilations. To analyze the performance results, we rely on NEC `Ftrace` profile analysis tool. All experiments are run in single complex precision arithmetics.

6.2. Performance Results on Synthetic Datasets

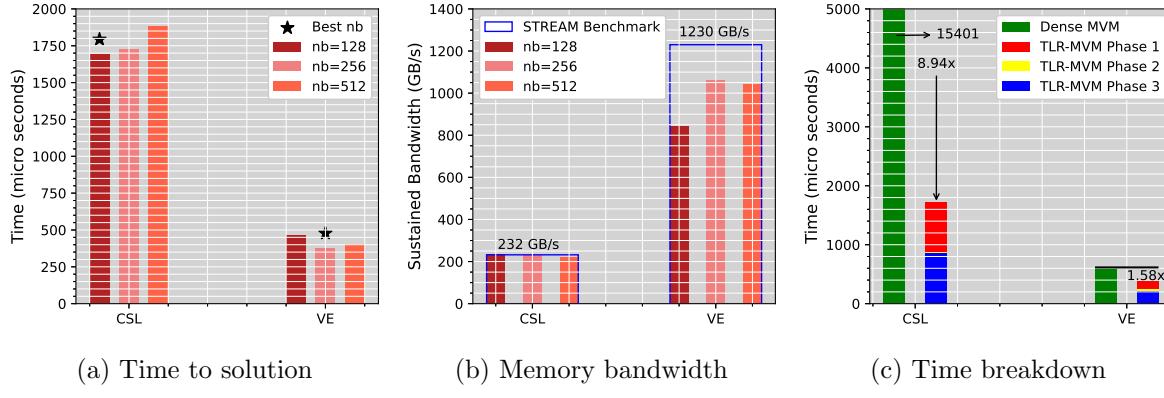


Figure 8. Performance analysis of TLR-MVM on synthetic datasets

We first study TLR-MVM on synthetic datasets to demonstrate the robustness and software capabilities of our TLR-MVM implementation. We randomly generate U and V bases in single complex precision. Without loss of generality, we employ square matrices with $m = n = 10000$ because it is closer to the real seismic datasets. The rank k is set to $nb/4$ and is constant across all tiles. This specific rank translates into a theoretical FLOPS saving factor of 2, i.e., TLR-MVM performs twice fewer FLOPS than dense MVM. Figures 8a and 8b show the time to solution and memory bandwidth using the synthetic datasets, respectively, on the single Intel CSL node and one NEC VE. We test three tile sizes, i.e., $nb = 128, 256, 512$. We also show the memory bandwidth obtained by STREAM Benchmark. This is the upper limit of sustained memory bandwidth from DDR4 memory (i.e., Intel CSL) and HBM2 (i.e., NEC VE). For Intel CSL, we run one MPI process per socket with 20 OpenMP threads. For the single NEC VE, we run one MPI process with 8 OpenMP threads. We see that a single NEC VE is much faster than Intel CSL thanks to HBM2 technology with up to 85 % of bandwidth saturation. On NEC VE, the configuration with the best time / bandwidth (annotated with a star) runs in less than 375 micro seconds and exceeds 1 TB/s.

Figure 8c shows the time breakdown of the three phases in TLR-MVM on Intel CSL and one NEC VE. As expected, the computational phases 1 and 3 are the most time-consuming with the batched MVM capturing most of the elapsed time. We also compare TLR-MVM with dense MVM, as implemented in the Level-2 BLAS CGEMV kernel in the vendor optimized libraries. TLR-MVM reaches 8.94 and 1.58 performance speedups compared to dense MVM on Intel CSL and one NEC VE, respectively. Compared to the theoretical FLOPS saving factor of 2, our TLR-MVM implementation gets higher performance speedup on Intel CSL thanks to a full saturation of the memory bandwidth. The dense MVM implementation from Intel MKL may not saturate the bandwidth enough and it seems to be poorly optimized. On the NEC VE card, our TLR-MVM implementation achieves less performance speedup when compared against the dense vectorized MVM from NEC NLC. As shown in Fig. 8b, there may still be some room for improvement for our TLR-MVM implementation in further saturating the main memory.

6.3. Performance Results on 3D Seismic Datasets

In this section, we run against 3D seismic datasets and conduct experiments using 150 frequencies. Each frequency matrix is of size 9801×9801 . As identified in Section 5, we use $nb = 256$

and $\epsilon = 0.001$ to compress the dense matrix into a tile low-rank (TLR) matrix, while delivering application's accuracy *and* performance on the NEC VE-based system.

6.3.1. Performance analysis over all frequencies

Figure 9a compares the time to solution for each frequency matrix on Intel CSL and NEC VE systems when running the dense MVM and TLR-MVM. The time for dense MVM is constant across all frequencies on both systems. However, the time for TLR-MVM increases with the frequency index. This trend confirms the outcome of the rank statistics analysis made in Section 4.1, where the ranks (i.e., the workloads) grow with the index of the frequency matrix. For high frequencies, TLR-MVM on NEC VE outperforms its counterpart on Intel CSL by up to a factor 4, which is aligned with results on synthetic datasets from Section 6.2. Figure 9b shows the same performance analysis but with the sustained bandwidth obtained for each frequency matrix on the two systems. The TLR-MVM bandwidth increases with the frequency index and achieves more than 1 TB/s for high frequencies. On Intel CSL, the dense MVM implemented in CGEMV kernel from MKL shows limited sustained bandwidth while TLR-MVM on the same system is able to saturate the bandwidth, as already demonstrated for synthetic datasets in Section 6.2. However, there is a clear load imbalance issue when looking at all frequencies and their respective makespan.

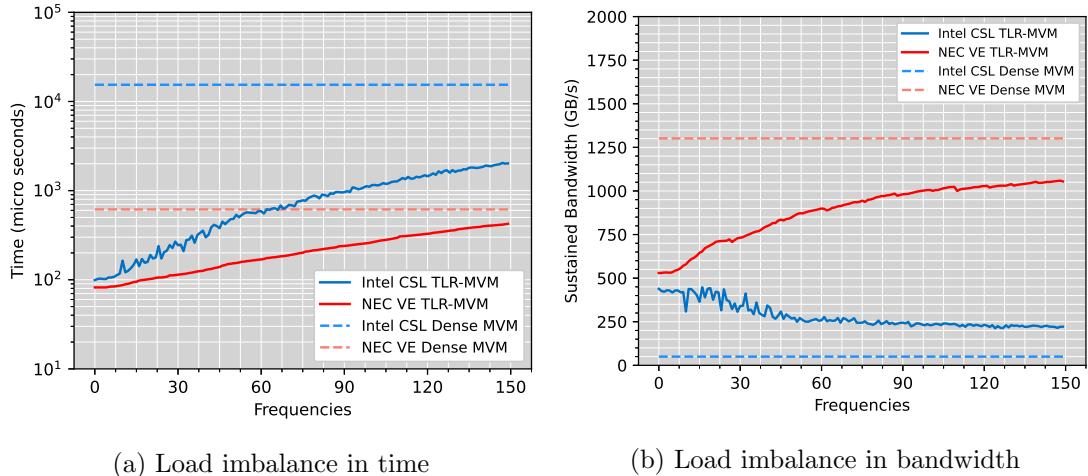


Figure 9. Performance analysis (time and bandwidth) for all frequency matrices (stored in dense and TLR) on Intel CSL and NEC VE systems, using $nb = 256$ and $\epsilon = 0.001$ for compression

6.3.2. Performance impact of optimization techniques

In this section, we assess the performance impact of the two previously introduced optimization techniques from Sections 4.2 and 4.3. We investigate four scenarios to solve the load imbalance issue: the reference TLR-MVM, the Merge-Phase TLR-MVM, the reference TLR-MVM with ZigZag mapping strategy and the Merge-Phase TLR-MVM with ZigZag mapping strategy. We conduct experiments on the 3D seismic dataset with 150 frequencies on 8 NEC VEs. We use NEC Ftrace analysis to get the FLOPS count of each OpenMP thread to illustrate how the various strategies can mitigate the load imbalance overhead. The default mode of NEC Ftrace is Vector Operation profiling. We need to set the environment variable `VE_PERF_MODE` to `VECTOR-MEM` to get the FLOPS count. Figure 10 reports the FLOPS count of each thread

for the aforementioned scenarios. Figure 10b shows how the Merge-Phase strategy improves the load balancing among threads within each VE. Figure 10c highlights how the ZigZag mapping strategy further distributes evenly the workload between NEC VE cards. Figure 10d pictures the performance impact when both strategies are activated. It permits to achieve inter-node and intra-node load balancing. It is also noteworthy to mention that the workload among threads

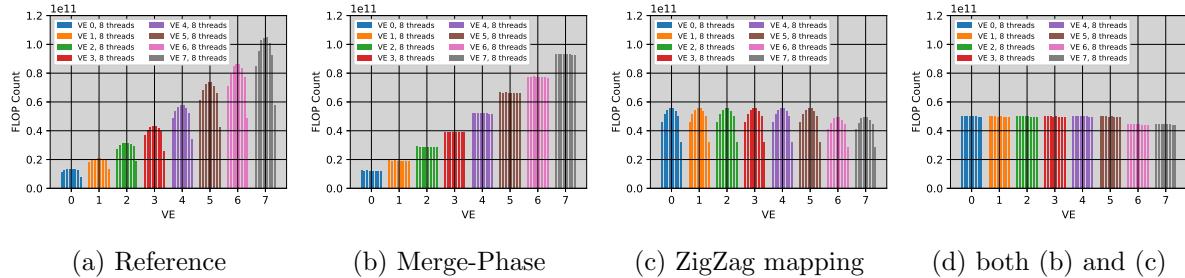


Figure 10. FLOPS count of different load balancing optimization technique for all threads when running against 150 frequencies using 8 VEs, (d) uses both Merge-Phase and ZigZag Mapping optimization

from VE 0 to VE 5 is higher than VE 6 and 7. This is because the number of frequencies (i.e., 150) is not divisible by the number of VEs (i.e., 8). Therefore, the first 6 VEs end up having one more frequency matrix to process.

Figure 11a compares time to solution of our TLR-MVM implementation with various optimization techniques against vendor optimized dense MVM on the overall application using one single Intel CSL node and a single NEC VE. TLR-MVM achieves up to 16X performance speedup against dense MVM on Intel CSL. This speedup factor is due to a suboptimal implementation of MKL multithreaded CGEMV kernel. Moreover, TLR-MVM scores up to 3.3X performance speedup compared to dense MVM on a single NEC VE. This result is on par with the theoretical FLOPS saving factor of 3.9 reported in Fig. 7a. The optimization techniques do not impact significantly the TLR-MVM performance variants on the single NEC VE since the number of OpenMP threads is limited to 8. However, they do impact the TLR-MVM performance on the Intel CSL system, due to the large thread count, i.e., a total of 40 threads.

Figure 11b shows the roofline performance model using a single NEC VE, while combining Merge-Phase + ZigZag mapping strategies. We select frequency indices 1, 50, 100, and 150 and show how TLR-MVM performance increases with the frequency index and gets near the HBM2 sustained bandwidth. Although the gain in time is important, the fine-grained computation of TLR-MVM may prevent matrices with low frequencies from getting closer to the sustained bandwidth on the NEC VE system due to low vector units utilization.

Figures 12a and 12b show the bandwidth (left y-axis) and time to solution (right y-axis) for the overall simulation with 150 frequency matrices. The figures report performance for each of the four strategies on Intel CSL and 8 NEC VEs. By combining the Merge-Phase + Zigzag strategies, we achieve the best time to solution and over 85 % memory bandwidth of the STREAM Benchmark for both systems, thanks to a better hardware occupancy. Our TLR-MVM achieves around 9 TB/s aggregated bandwidth on 8 NEC VEs: this bandwidth score is approximately equivalent to 36 Intel CSL nodes. This result highlights the performance advantage of HBM2 over DDR4 memory technology.

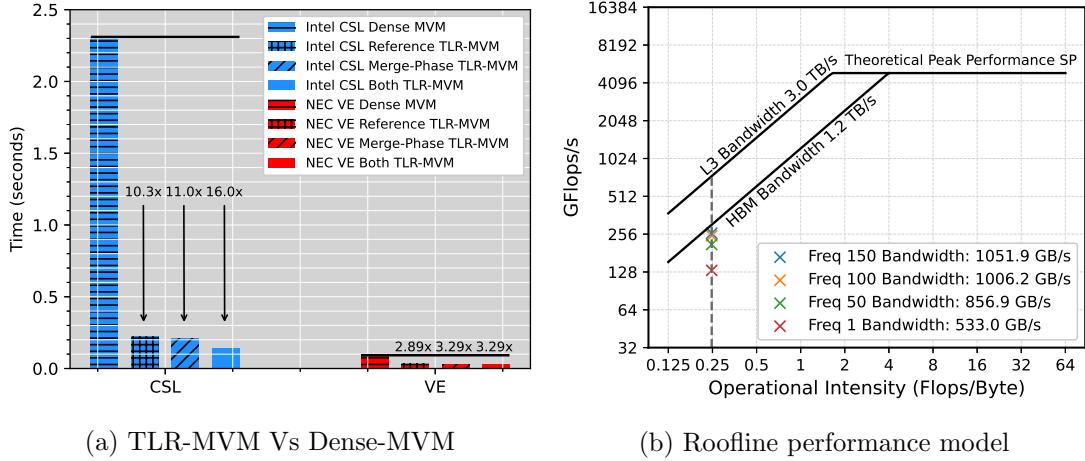


Figure 11. Performance comparison and assessment of TLR-MVM on Intel CSL node and single NEC VE

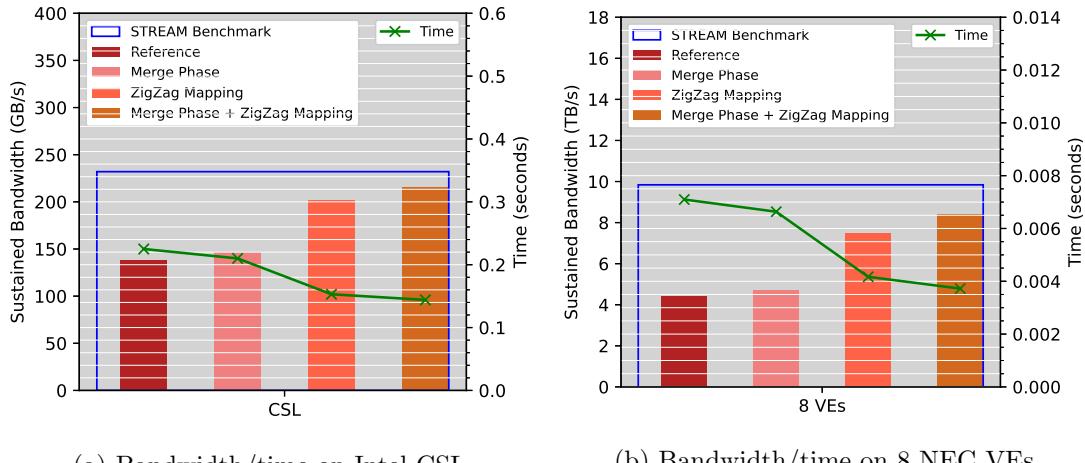


Figure 12. Performance impact of optimization techniques on TLB-MVM

6.3.3. Performance scalability

We scale up the number of VEs as well as the problem size to study the performance scalability of our TLR-MVM implementation with both Merge-Phase and ZigZag mapping strategies activated. In order to prove the scalability of our algorithm even beyond the 150 frequency matrices previously used, 3 additional fictitious matrices are created in between each of the available matrices by means of linear interpolation. This leads to an augmented dataset composed of 596 frequency matrices, which is on par with real seismic applications that deal with broadband data (i.e., data spanning a broad range of frequencies). Figure 13a shows scalability results on 596 frequency matrices up to 128 NEC VEs. Our TLR-MVM implementation reaches 67 % of the sustained bandwidth and 77.7 % of linear scalability when using 128 NEC VEs. Figure 13b is a close-up view for up to 32 VEs. Note that the reference and Merge-Phase strategies can only run starting from 5 VEs because there is not enough memory on the VEs to host all frequency matrices. The ZigZag mapping strategy alleviates this bottleneck and permits to balance the memory allocation as well. Figure 13c shows time to solution comparison of vendor optimized dense MVM CGEMV kernel against TLR-MVM on up to 128 VEs. The average acceleration of TLR-MVM over dense MVM is $3.13\times$ on NEC VEs. This number is again on par with the theoretical FLOPS saving factor of 3.9 reported in Fig. 7a, since the interpolation used

to extend the 3D seismic datasets is linear. Comparing against CGEMV from MKL on 128 dual-socket 20-core Intel Cascade Lake nodes with DDR4 memory, our TLR-MVM implementation achieves 67X performance speedup when using the same number of NEC VEs, i.e., 128 cards. This corresponds to an aggregated bandwidth around 110 TB/s.

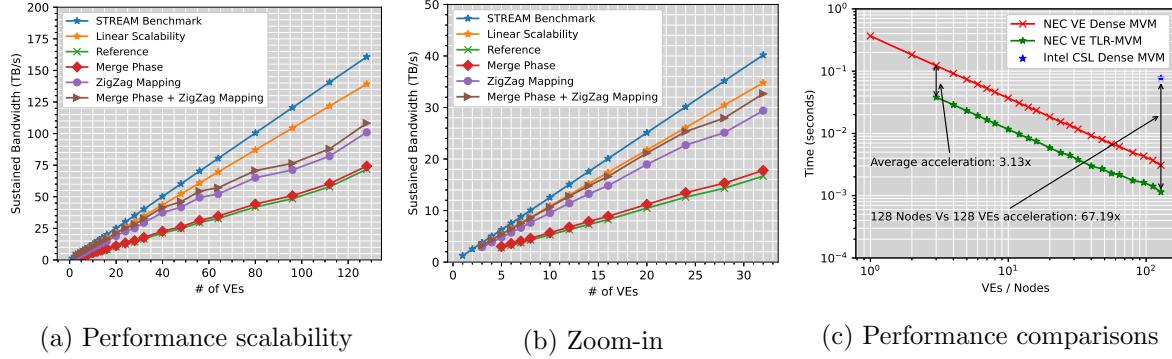


Figure 13. Performance scalability up to 128 NEC VEs

7. Discussions and Future Work

In this work, our attention has primarily focused on reducing the memory footprint of the kernel of the multi-dimensional convolution operator as well as improving its computational efficiency. Both goals have been largely achieved by means of the proposed TLR-MVM implementation. Given the nature of the inverse problem that we wish to solve (Eq. 1), several questions remain to be answered in our future endeavors. First, whilst we currently focus on improving the inner working of the \mathbf{R} operator, the algorithm requires four slightly different computations to be implemented, namely \mathbf{R} , \mathbf{R}^* , \mathbf{R}^* , $(\mathbf{R}^*)^H = \mathbf{R}^T$. The latter three computations call for the application of the transpose and/or complex conjugate kernel to the input vector. Similar to its dense counterpart [29], we envision a TLR-MVM implementation where complex conjugation is applied to the input and output vectors instead of to the kernel itself. Moreover, given that the overall kernel is divided into tiles, the application of the transposed kernel is expected to benefit from the fact that elements of different rows of the U and V bases are closer in memory than their dense counterparts. Whilst this suggests a similar workload for the forward and adjoint passes, numerical validation is required. Moreover whilst our focus has so far not included the forward and inverse FFTs that comprise part of the operator in Eq. 3, future research will investigate the possibility to begin their computations as soon as some of the TLR-MVM have been executed without waiting for the computations over the entire frequency range to be finalized. Another opportunity lies in the fact that the inverse problem we wish to solve can be slightly modified to include more than one spatial coordinates \mathbf{x}_B at the time [29]; in other words, our batched TLR-MVM can be replaced by a batched tile low-rank matrix-matrix multiplication (TLR-MMM) where each column of the input matrix represents the wavefield originated from a different virtual source.

Conclusion

In this paper, we investigate and deploy the Tile Low-Rank (TLR) Matrix-Vector Multiplication (MVM) performance to accelerate 3D seismic application workloads using vector computing hardware solutions based on NEC SX-Aurora TSUBASA architecture. We propose

and implement strategies to mitigate the load imbalance overheads that inherently emerge from such workloads. Our TLR-MVM implementation not only improves the overall performance but also permits to scale up in terms of memory footprint. Thanks to its fine-grained computations and memory-friendly data layout, our TLR-MVM implementation can leverage the HBM2 technology of NEC vector engines, which translates into a performance boost compared to Intel CSL architecture with DDR4 memory technology. We assess accuracy of our TLR matrix approximations and demonstrate the numerical robustness of our method by investigating the impact of compression on the subsurface image quality using signal-to-noise ratio as a qualitative metric. We then employ the roofline performance model to show how TLR-MVM is able to effectively extract performance from the underlying architecture. On distributed-memory environment, our TLR-MVM implementation reaches around 110 TB/s of aggregated bandwidth on 128 NEC vector engines, which converts to 67X performance speedup in time against vendor optimized dense MVM (i.e., MKL CGEMV kernel) with the same number of Intel CSL nodes. We believe these results are promising in the context of 3D seismic imaging. TLR-MVM may enable to increase the problem sizes further and eventually improve the quality of 3D land surveys.

Acknowledgements

For computer time, this research used the *Ibex* system hosted at the Supercomputing Laboratory at KAUST.

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. Abdelfattah, A., Haidar, A., Tomov, S., Dongarra, J.J.: Performance, design, and autotuning of batched GEMM for GPUs. In: Kunkel, J.M., Balaji, P., Dongarra, J.J. (eds.) High Performance Computing. ISC High Performance 2016. Lecture Notes in Computer Science, vol. 9697, pp. 21–38. Springer (2016). https://doi.org/10.1007/978-3-319-41321-1_2
2. Akbudak, K., Ltaief, H., Mikhalev, A., Keyes, D.: Tile Low Rank Cholesky Factorization for Climate/Weather Modeling Applications on Manycore Architectures. In: High Performance Computing. ISC 2017. Lecture Notes in Computer Science, vol. 10266, pp. 22–40. Springer (2017). https://doi.org/10.1007/978-3-319-58667-0_2
3. Akbudak, K., Ltaief, H., Mikhalev, A., *et al.*: Exploiting data sparsity for large-scale matrix computations. In: Aldinucci, M., Padovani, L., Torquati, M. (eds.) High Performance Computing. ISC High Performance 2016. Lecture Notes in Computer Science, vol. 11014, pp. 721–734. Springer (2018). https://doi.org/10.1007/978-3-319-96983-1_51
4. Al-Harthi, N., Alomairy, R., Akbudak, K., *et al.*: Solving Acoustic Boundary Integral Equations Using High Performance Tile Low-Rank LU Factorization. In: High Performance Computing. ISC High Performance 2020. Springer (2020). https://doi.org/10.1007/978-3-030-50743-5_11

5. Amestoy, P., Ashcraft, C., Boiteau, O., *et al.*: Improving Multifrontal Methods by Means of Block Low-Rank Representations. *SIAM Journal on Scientific Computing* 37(3), A1451–A1474 (2015). <https://doi.org/10.1137/120903476>
6. Amundsen, L.: Elimination of Free-surface Related Multiples Without Need of a Source Wavelet. *Geophysics* 66, 327–341 (2001). <https://doi.org/10.1190/1.1444912>
7. Berryhill, J.R.: Wave-equation Datuming Before Stack. *Geophysics* 49, 2064–2066 (1984). <https://doi.org/10.1190/1.1441620>
8. Börm, S.: Efficient Numerical Methods for Non-Local Operators: H2-matrix Compression, Algorithms and Analysis, vol. 14. European Mathematical Society (2010). <https://doi.org/10.4171/091>
9. Börm, S., Grasedyck, L., Hackbusch, W.: Introduction to Hierarchical Matrices with Applications. *Engineering Analysis with Boundary Elements* 27(5), 405–422 (2003). [https://doi.org/10.1016/S0955-7997\(02\)00152-2](https://doi.org/10.1016/S0955-7997(02)00152-2)
10. Boukaram, W.H., Turkiyyah, G., Ltaief, H., Keyes, D.E.: Batched QR and SVD Algorithms on GPUs with Applications in Hierarchical Matrix Compression. *Parallel Computing* 74(C), 19–33 (2018). <https://doi.org/10.1016/j.parco.2017.09.001>
11. Brackenhoff, J., Thorbecke, J., Koehne, V., *et al.*: Implementation of the 3D Marchenko method (2020). <https://doi.org/10.1190/geo2017-0108.1>
12. Broggini, F., Snieder, R., Wapenaar, K.: Focusing the Wavefield Inside an Unknown 1D Medium: Beyond Seismic Interferometry. *Geophysics* 77(5), A25–A28 (2012). <https://doi.org/10.1190/geo2012-0060.1>
13. Cao, Q., Pei, Y., Akbudak, K., *et al.*: Extreme-Scale Task-Based Cholesky Factorization Toward Climate and Weather Prediction Applications. In: Proceedings of the Platform for Advanced Scientific Computing Conference. pp. 2:1–2:11. ACM (2020). <https://doi.org/10.1145/3394277.3401846>
14. Charara, A., Keyes, D., Ltaief, H.: Tile Low-Rank GEMM Using Batched Operations on GPUs. In: Aldinucci, M., Padovani, L., Torquati, M. (eds.) Euro-Par 2018: Parallel Processing. Lecture Notes in Computer Science, vol. 11014, pp. 811–825. Springer (2018). https://doi.org/10.1007/978-3-319-96983-1_57
15. Charara, A., Keyes, D., Ltaief, H.: Batched Triangular Dense Linear Algebra Kernels for Very Small Matrix Sizes on GPUs. *ACM Transactions on Mathematical Software* 45(2) (2019). <https://doi.org/10.1145/3267101>
16. Corona, E., Martinsson, P.G., Zorin, D.: An $O(N)$ Direct Solver for Integral Equations on the Plane. *Applied and Computational Harmonic Analysis* 38(2), 284–317 (2015). <https://doi.org/10.1016/j.acha.2014.04.002>
17. Goreinov, S., Tyrtyshnikov, E., Yeremin, A.Y.: Matrix-Free Iterative Solution Strategies for Large Dense Linear Systems. *Numerical Linear Algebra with Applications* 4(4), 273–294 (1997)

18. Grasedyck, L., Kressner, D., Tobler, C.: A Literature Survey of Low-Rank Tensor Approximation Techniques. *GAMM-Mitteilungen* 36(1), 53–78 (2013). <https://doi.org/10.1002/gamm.201310004>
19. van Groenestijn, G.J., Verschuur, D.J.: Estimating Primaries by Sparse Inversion and Application to Near-offset Data Reconstruction. *Geophysics* 74(3), 1MJ–Z54 (2009). <https://doi.org/10.1190/1.3111115>
20. Hackbusch, W.: A Sparse Matrix Arithmetic Based on \mathcal{H} -matrices. Part I: Introduction to \mathcal{H} -Matrices. *Computing* 62(2), 89–108 (1999). <https://doi.org/10.1007/s006070050015>
21. Halko, N., Martinsson, P.G., Tropp, J.A.: Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. *SIAM Review* 53(2), 217–288 (2011). <https://doi.org/10.1137/090771806>
22. Jumah, B., Herrmann, F.J.: Dimensionality-reduced Estimation of Primaries by Sparse Inversion. *Geophysical Prospecting* 62(5), 972–993 (2014). <https://doi.org/10.1111/1365-2478.12113>
23. Keyes, D.E., Ltaief, H., Turkiyyah, G.: Hierarchical Algorithms on Hierarchical Architectures. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378(2166), 20190055 (2020). <https://doi.org/10.1098/rsta.2019.0055>
24. Kriemann, R.: H -LU Factorization on Many-Core Systems. *Computing and Visualization in Science* 16(3), 105–117 (2013). <https://doi.org/10.1007/s00791-014-0226-7>
25. Lindstrom, P.: Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics* 20(12), 2674–2683 (2014). <https://doi.org/10.1109/TVCG.2014.2346458>
26. Ltaief, H., Cranney, J., Gratadour, D., *et al.*: Meeting the Real-Time Challenges of Ground-Based Telescopes Using Low-Rank Matrix Computations (2021), <http://hdl.handle.net/10754/669813>
27. van der Neut, J., Thorbecke, J., Wapenaar, K., Slob, E.: Inversion of the Multidimensional Marchenko Equation. In: 77th Conference and Exhibition, EAGE, Extended Abstracts. vol. 2015, pp. 1–5. European Association of Geoscientists & Engineers (2015). <https://doi.org/10.3997/2214-4609.201412939>
28. Ravasi, M., Vasconcelos, I.: PyLops – A Linear-operator Python Library for Scalable Algebra and Optimization. *SoftwareX* 11, 100361 (2020). <https://doi.org/10.1016/j.softx.2019.100361>
29. Ravasi, M., Vasconcelos, I.: An Open-source Framework for the Implementation of Large-scale Integral Operators with Flexible, Modern HPC Solutions - Enabling 3D Marchenko Imaging by Least Squares Inversion. *Geophysics* pp. 1–74 (2021). <https://doi.org/10.1190/geo2020-0796.1>
30. Ravasi, M., Vasconcelos, I., Kritski, A., *et al.*: Target-oriented Marchenko Imaging of a North Sea Field. *Geophysical Journal International* 205(1), 99–104 (2016). <https://doi.org/10.1093/gji/ggv528>

31. Rouet, F.H., Li, X.S., Ghysels, P., Napov, A.: A Distributed-memory Package for Dense Hierarchically Semi-separable Matrix Computations Using Randomization. *ACM Transactions on Mathematical Software (TOMS)* 42(4), 27 (2016). <https://doi.org/10.1145/2930660>
32. Verschuur, D.J.: Surface-related Multiple Elimination in Terms of Huygens Sources. *Journal of Seismic Exploration* 1, 49–59 (1992)
33. Wapenaar, K., Thorbecke, J., van der Neut, J., *et al.*: Marchenko Imaging. *Geophysics* 79(3), WA39–WA57 (2014). <https://doi.org/10.1190/geo2013-0302.1>
34. Williams, S., Waterman, A., Patterson, D.: Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM* 52(4), 65–76 (2009). <https://doi.org/10.1145/1498765.1498785>
35. Yilmaz, O.: *Seismic Data Analysis*. Society of Exploration Geophysicists (2001)

Porting and Optimizing Molecular Docking onto the SX-Aurora TSUBASA Vector Computer

Leonardo Solis-Vasquez¹ , **Erich Focht²** , **Andreas Koch¹** 

© The Authors 2021. This paper is published with open access at SuperFri.org

In computer-aided drug design, the rapid identification of drugs is critical for combating diseases. A key method in this field is molecular docking, which aims to predict the interactions between two molecules. Molecular docking involves long simulations running compute-intensive algorithms, and thus, can profit a lot from hardware-based acceleration. In this work, we investigate the performance efficiency of the SX-Aurora TSUBASA vector computer for such simulations. Specifically, we present our methodology for porting and optimizing AutoDock, a widely-used molecular docking program. Using a number of platform-specific code optimizations, we achieved executions on the SX-Aurora TSUBASA that are in average $3.6\times$ faster than on modern 128-core CPU servers, and up to a certain extent, competitive to V100 and A100 GPUs. To the best of our knowledge, this is the *first* molecular docking implementation for the SX-Aurora TSUBASA.

Keywords: application porting, performance optimization, molecular docking, AutoDock, vector computing, SX-Aurora.

Introduction

In recent years, the NEC SX-Aurora TSUBASA computer system has been introduced to the High Performance Computing (HPC) landscape. Besides its core technologies, i.e., vector-based processing and high memory bandwidth (1.53 TB/s), the SX-Aurora TSUBASA offers a programming framework based on standard C/C++, which eases the porting of existing programs. Simulations in computational dynamics, electromagnetism, and other fields have been accelerated on this platform [6, 13, 21], and thus, the SX-Aurora TSUBASA has become an alternative accelerator platform in HPC.

The applicability of the SX-Aurora TSUBASA in other scientific areas is yet to be investigated. An example is the field of computer-aided drug design, which leverages compute-intensive molecular docking simulations. Basically, molecular docking predicts close-distance interactions of two molecules: the receptor and the ligand, both of known three-dimensional structure. A receptor models a biological target, while a ligand acts as a drug candidate. Identifying new ligands can be done by screening large databases of small molecules, aiming to find those that interact favorably with a given receptor [9]. This process, called virtual screening, typically requires thousands of molecular docking executions. However, it enables using only promising (and fewer!) ligands in the subsequent costly and slow wet lab experiments. Software tools for molecular docking have become relevant at combating diseases. One of these is the widely-used AutoDock, which has been used as the docking engine in world-wide community grid projects such as *Fight-AIDS@Home* [1] as well as *OpenPandemics: COVID-19* [3]. In algorithmic terms, AutoDock explores several spatial geometrical arrangements between a receptor and a ligand (i.e., poses) using nested loops and divergent control flows. Moreover, AutoDock computes a score for each pose, and in turn, performs millions of score evaluations per execution.

In this paper, we present our methodology for efficiently porting and optimizing AutoDock onto the SX-Aurora TSUBASA. The code developed in this work, termed AutoDock-Aurora, has been ported from an existing OpenCL version of AutoDock. For achieving higher performance,

¹Technical University of Darmstadt, Darmstadt, Germany

²NEC Deutschland GmbH, Stuttgart, Germany

platform-specific coding styles (e.g., loop pushing, data compression, predication) as well as optimization practices (e.g., those based on compiler technologies) were applied. With **AutoDock-Aurora**, we aim to expand the applicability of the SX-Aurora TSUBASA to solving a wider range of scientific problems. The organization of this paper is as follows. Section 1 provides background information on the target platform and application under analysis. Section 2 describes our porting and optimization methodology. Section 3 discusses the results of evaluating **AutoDock-Aurora** on the SX-Aurora TSUBASA as well as on modern high-end GPUs and CPUs. Section 4 reviews the related work. Finally, our conclusions and future work are presented.

1. Background

1.1. SX-Aurora Vector Engine

The SX-Aurora TSUBASA Vector Engine (VE) is an accelerator in the shape of a full profile dual-slot PCI card. The VE1 and VE2 generation processors have 6×8 GB HBM2 stacks for a total of 48 GB RAM with up to 1.53 TB/s memory bandwidth. The regular VEs that we used have only eight cores connected to a 16 MB Last Level Cache (LLC) through a fast 2D network-on-chip. Each core consists of a scalar processing unit with RISC instruction set, out-of-order execution, L1 and L2 caches, that is attached to a vector processing unit with 64 long vector registers of 256×64 bit words and several vector execution units. Unlike normal SIMD or SIMT architectures, the vector units are implemented as a combination of 32×64 bit wide SIMD units with 8-cycle deep pipelines. A vector length register controls the number of elements processed in vector operations, and 16 vector mask registers enable predication.

The VE’s normal mode has uniform memory access (UMA), with all cores being able to access and use any part of the LLC and HBM2 memory. It can be reconfigured to *partitioned* mode with non-uniform memory access (NUMA), where cores are split into two equally sized groups, and by default access only their segment of LLC and HBM2 memory. NUMA mode reduces memory port and memory network conflicts, and can bring performance benefits for certain classes of programs.

VEs need to run inside a normal Linux system usually called Vector Host (VH). The VH runs the VE Operating System (VEOS) which manages VE resources, schedules processes, and manages physical and virtual memory of VEs. Programs can run natively on the VEs, with system calls offloaded to the VH and processed on behalf of their user. They behave as if running under Linux on the host, although the VE itself runs as “bare metal”, without any kind of on-board operating system. Native VE programs can call functions that run on the VH, in a pattern called reverse offloading.

Vector Engine Offloading (VEO) [8] is a programming model which executes the main program on the VH and offloads kernels to the VEs. While the API somewhat resembles OpenCL, it differs from it due to the SIMD/vector nature of kernels, and due to their ability to execute almost any Linux system call. VEO is the lowest-level API for accelerator style programming and is the technique used for this project.

Alternative hybrid programming approaches include VEDA [4], which implements a CUDA-like device API on top of VEO, neoSYCL [12], OpenMP target offloading [5] integrated with the LLVM compiler, HAM [20], and NEC Hybrid MPI.

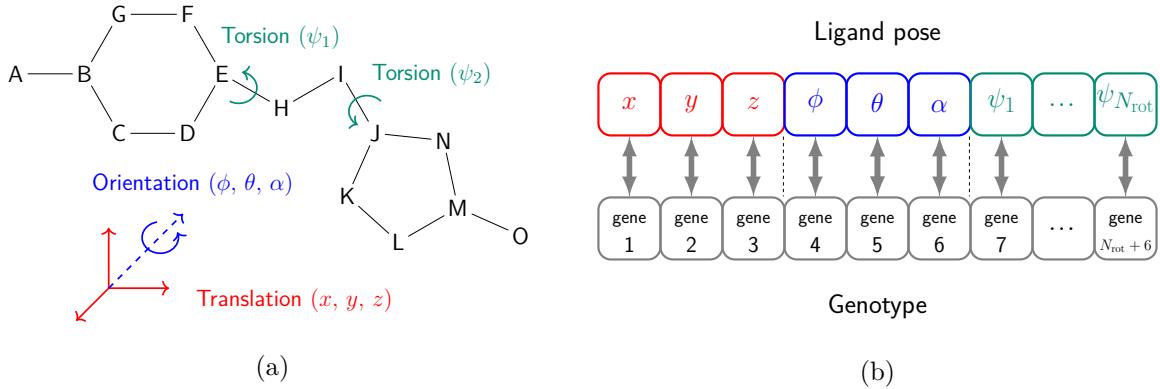


Figure 1. (a) Degrees of freedom of a theoretical ligand composed of atoms A, B, C, …, O. Bonds between atoms are depicted as connecting lines. Each rotatable bond such as E–H and I–J corresponds to a torsion, i.e., rotation of affected ligand atoms around the rotatable-bond axis. (b) Mapping between a ligand pose (a set of degrees of freedom) and a genotype (set of genes). The number of rotatable bonds in a ligand is denoted as N_{rot}

1.2. Molecular Docking

Molecular docking consists of solving an optimization problem that explores the poses adopted by a ligand with respect to a receptor. It is based on the *lock and key* concept by Fischer [7], in which a *perfect* binding between a ligand and receptor occurs when they have *exactly complementary* geometric shapes. The two main aims of molecular docking are: first, to predict the ligand poses within a certain binding site of a receptor; and second, to estimate the affinity of their corresponding interactions. As shown in Fig. 1a, a ligand pose can be represented by the degrees of freedom (i.e., translation, rotations, and torsions) experienced during interaction. As such representation typically involves many degrees of freedom, the docking optimization problem suffers from a combinatorial explosion. To cope with that, molecular docking performs a systematic exploration via heuristic *search methods* (e.g., genetic algorithms, simulated annealing, etc.), which are assisted by *scoring functions* that estimate the binding affinity. Extensive discussions on the categories of search methods and scoring functions are available in [15, 29].

Particularly, AutoDock [16] is one of the most-cited software tools in molecular docking. It is implemented in C++, and provided as open source. The main computation engine in AutoDock is a Lamarckian Genetic Algorithm (LGA), which hybridizes two methods: a Genetic Algorithm and a Local Search.

A Genetic Algorithm (GA) is inspired by the Darwinian evolution theory, and hence, it maps the docking search into a biological evolution process. In this context, each of the ligand's degrees of freedom corresponds to a *gene*. A ligand pose, composed of the entire set of degrees of freedom, corresponds to an *individual*, which in turn is represented by its *genotype* (i.e., full set of genes) as shown in Fig. 1b. Individuals experience genetic modifications such as *crossover* and *mutation*. Moreover, the population of individuals undergoes a *selection* procedure that chooses the *stronger* ones for the next generation. An individual's strength is quantified with its score, which is evaluated with a scoring function.

In the context of molecular docking, a score enhancement implies a *minimization* of its value. In other words, the lower the scores, the stronger the ligand-receptor interactions. In AutoDock, the Local Search (LS) aims to improve the scores of the poses already generated via the GA. For that purpose, AutoDock subjects a population subset of randomly-chosen individuals (LS_{rate} ,

default: 6 %) to the method of Solis-Wets [24]. Basically, this is an adaptive-iterative method that takes a genotype as input, and generates a new one by adding small changes (constrained random amount) to each of the input genes. Then, the scores of these two genotypes are computed and compared. If the score is not minimized, a second genotype is generated by subtracting (instead of adding) small changes to the input genes. Afterwards, a second score comparison is performed. The termination criterion is adapted at runtime according to the number of successful or failed attempts at minimizing the score. In each generation, the poses which could actually be improved by the LS are then re-introduced into the LGA population.

Furthermore, AutoDock uses the scoring function (SF) in Eq. 1, which computes the binding affinity as a semi-empirical free-energy force field (kcal/mol) [10]. The first terms involve the summation of four interaction types (Van der Waals, hydrogen bonding, electrostatics, and desolvation) over all the ligand and receptor atoms. The fifth term represents the (unfavorable) loss of ligand entropy upon binding due to the N_{rot} rotatable bonds. All terms are characterized by constant weights (W_{vdw} , W_{hb} , W_{el} , W_{ds} , W_{rot}) and look-up tables (A_{ij} , B_{ij} , C_{ij} , D_{ij} , S , V), as well as by other parameters. Most importantly, the score is determined by the interatomic distance r_{ij} . The value of r_{ij} is calculated at runtime from the atomic coordinates of atoms i and j , and depends entirely on a genotype (generated either via GA or LS), that in turn encodes a respective ligand movement.

$$\begin{aligned} \text{SF} = \sum_{i,j} \left[W_{\text{vdw}} \left(\frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^6} \right) + W_{\text{hb}} E(t) \left(\frac{C_{ij}}{r_{ij}^{12}} - \frac{D_{ij}}{r_{ij}^{10}} \right) + \right. \\ \left. W_{\text{el}} \left(\frac{q_i q_j}{\epsilon(r_{ij}) r_{ij}} \right) + W_{\text{ds}} (S_i V_j + S_j V_i) e^{-\frac{r_{ij}^2}{2\sigma^2}} \right] + W_{\text{rot}} N_{\text{rot}} \end{aligned} \quad (1)$$

The block diagram in Fig. 2 depicts the functionality of AutoDock, along with the default values of LGA parameters. The generation of genotypes via GA is parameterized with the ratios of crossover (R_{cross}), mutation (R_{mut}), and selection (R_{sel}), while the termination of LS is controlled by the maximum number of iterations ($N_{\text{LS-iters}}^{\text{MAX}}$), as well as by the minimum change step (step^{MIN}). A single LGA-run optimizes the scores of a population (of $N_{\text{pop-size}}$ individuals) until reaching the maximum number of score evaluations ($N_{\text{score-evals}}^{\text{MAX}}$) or generations ($N_{\text{gens}}^{\text{MAX}}$), whichever comes first. An AutoDock docking job consists of the execution of several LGA runs, typically $N_{\text{LGA-runs}} = 50$, which are completely independent from each other. Furthermore, evaluating the score of an individual involves three steps. First, the generated pose (expressed as genotype) is transformed into its corresponding atomic coordinates. Then, intermolecular (ligand-receptor) and intramolecular (ligand-ligand) interactions are calculated using Eq. 1. Note that the interactions between receptor atoms are not calculated, as this molecule is treated as rigid [27].

2. Methodology

2.1. Porting

As reported in Section 4, AutoDock has been ported to other accelerators such as GPUs and FPGAs. In this work, we used `ocladock-fpga` [27] (OpenCL implementation for FPGAs) as the starting point of our development for SX-Aurora. Compared to `AutoDock-GPU` [23] (OpenCL implementation for GPUs/CPUs), `ocladock-fpga` is intuitively close to the programming model of the VE and thus should allow for easier code porting.

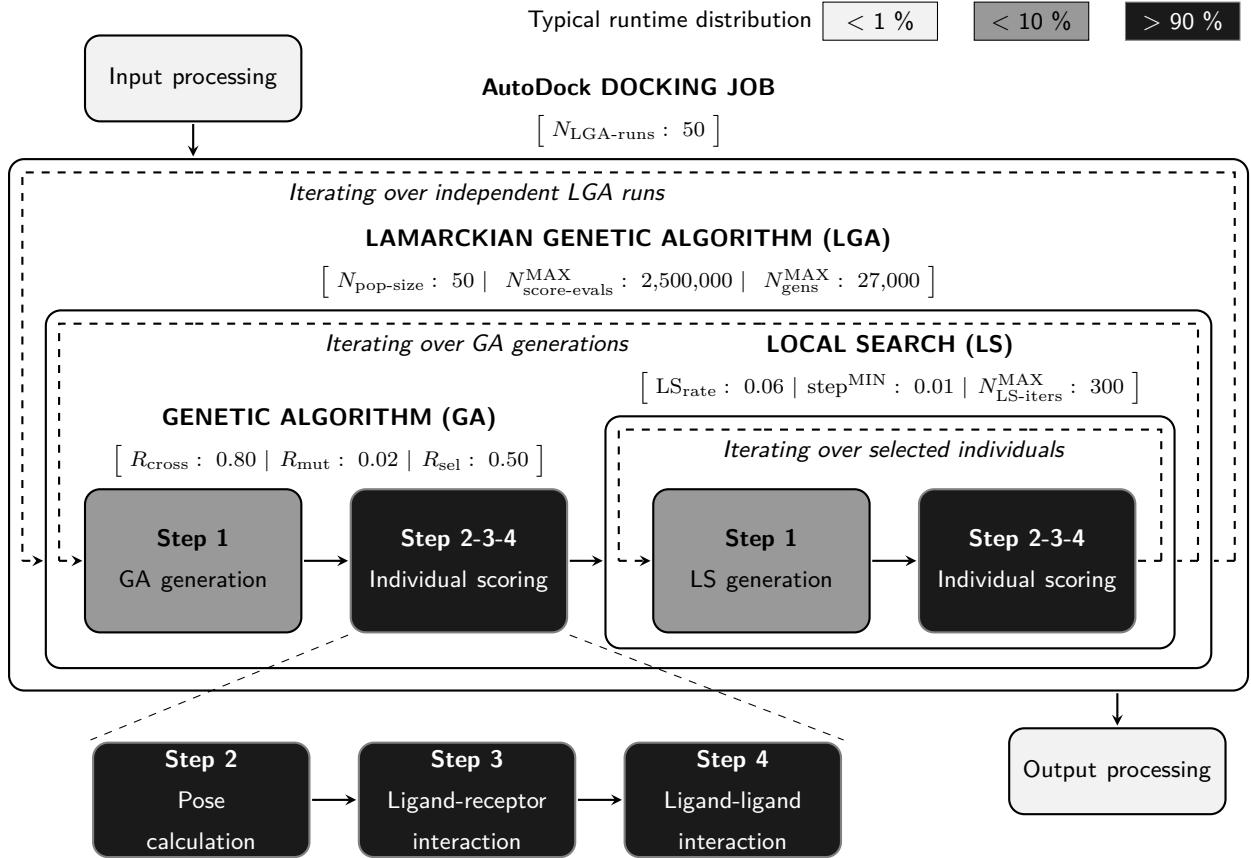


Figure 2. AutoDock block diagram [26] with default values of LGA parameters

For instance, like any other OpenCL/CUDA program for GPUs, AutoDock-GPU follows a SIMD programming style, where data to be processed is accessed through a grid of threads indexes, e.g., those obtained via the OpenCL `get_global_id()`/`get_local_id()` built-in functions. In contrast, each of the component kernels of `ocladock-fpga` was coded as a *single-threaded task*. This implementation approach keeps most of the loop structures shown in Fig. 2 intact, and thus, allows porting such loops with only minor effort, as well as allowing the use of vectorization for loop-level acceleration.

In AutoDock-Aurora, we use the same host and device code partitioning already defined in `ocladock-fpga`. Thus, we adopt the VEO programming model, where the overall program management is assigned to the host, and the independent LGA runs are offloaded onto the VE. Regarding the host code, we kept most of it intact, except for the calls to OpenCL APIs that we replaced with their VEO counterparts. For adapting the device code, we removed all language-specific qualifiers, so that OpenCL kernels were transformed into plain standard C++ functions. In particular, the baseline code in `ocladock-fpga` uses OpenCL pipes (*on-chip* FIFO-like structures) to pass data between kernels without resorting to any external memory. Hence, we removed the calls to OpenCL `read_pipe()`/`write_pipe()`, and replaced them with required function calls. While the porting just described might appear to require a little effort, it was not a trivial task. In fact, the non-determinism (due to randomness) in the GA heuristics was a major cause of masked errors. Consequently, we had to spend significant development times verifying AutoDock-Aurora's functionality, so that the resulting ligand poses and scores actually reach the expected level of convergence.

In an initial optimization pass, we followed the compiler hints, as well as the NEC performance tuning guidelines [18]. Examples of code optimizations applied here include the removal of data dependencies, and the usage of more suitable data types (e.g., four-byte `int` instead of single-byte `char`) for index and loop-control variables. As a result, we achieved a full vectorization of the time-consuming functions computing the ligand-receptor and ligand-ligand scores (Fig. 2). Furthermore, by adding the directive `#pragma omp parallel for` to the outermost loop of the device code, we were able to parallelize the independent LGA runs, and hence, to distribute them among the eight VE cores.

2.2. Optimization

While the ported version already ran correctly, was vectorized and parallelized, its performance was not quite satisfactory yet, executing *slower* by a factor of $\sim 2.2\times$ compared to the host CPU. Each thread of the OpenCL-derived SIMD code being mapped to one VE core was using the vector pipes only for the innermost loops, which are generally quite short. They iterate over the number of atoms of the ligand, or over the number of rotational degrees of freedom of the ligand. For the examples tested, both loop lengths were of the order of magnitude $O(10)$, leading to vector lengths of just 1/10th (or even less!) of the maximum vector length of the VE.

The main optimization approach for increasing the vector lengths of the device kernels was to switch from mapping one SIMD thread to one VE *core*, to mapping one SIMD thread to one *vector lane*. As depicted in Fig. 2, there is no obvious outer loop in the LGA that starts with a genetic population and evolves it while selecting the best scoring individuals. **Step 2**, **Step 3**, **Step 4** (individual scoring) are the most computationally intensive parts of the algorithm, and can be rewritten so that they handle a large number of individuals at once, in parallel. For each of the individuals, the operations in the scoring functions are basically the same. Thus, we can express this convergent code as either an outer loop over individuals calling the three steps inside, or as three functions which handle, each, the *entire* set of individuals. The loop over individuals can be *pushed* into each of the functions in such a way that it then becomes the innermost, data parallel, and easily vectorizable loop (Fig. 3). For optimal performance, this well-known *loop-pushing* technique must be paired with changes in the data layout, such that the vectorized code accesses data with unit-strides as much as possible.

The Local Search part of the LGA algorithm is computationally divergent code because each of the genetic individuals in the population evolves differently, can mutate with various parameters into different directions, or might already have converged. We were able to perform the *loop-pushing* optimization within the Local Search part by using predication, and compressing the data for the non-converged part of the population (Fig. 4). This aims to keep the compute-intensive scoring functions working with unit-stride accesses and without additional predication. During Local Search, already-converged individuals are removed from the computation, thus reducing the length of the innermost loop. A large population size is thus beneficial for performance, because it increases the average loop length, and thus, the performance during Local Search.

In order to vectorize the Local Search code performing the generation of individuals according to Solis-Wets [24], we needed to replace the linear congruential random generator that was originally employed. The reason for this is that each of the generated random values in the aforementioned scheme depends on the previous one, i.e., $X_{n+1} = f(X_n)$, thus hindering vectorization/parallelization. Instead, we used a 64-bit Mersenne Twister pseudorandom generator [19] implemented in the NEC NLC libraries.

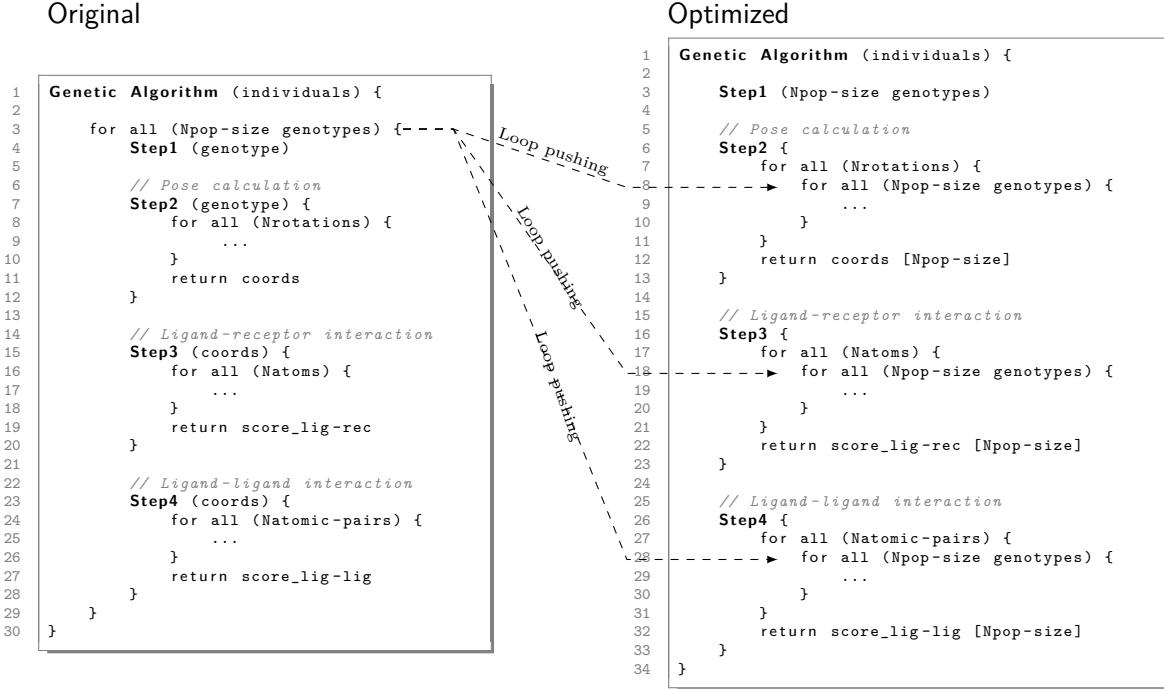


Figure 3. Optimization in Genetic Algorithm (GA): pushing the outer loop into the three components of the scoring function (Step 2, Step 3, Step 4)

Previous work `ocladock-fpga` [27] has shown that reducing the numerical precision from double to single floating-point does not impact the ability of the Genetic Algorithm to localize minimal energy configurations. This has been further exploited in AutoDock-GPU [23] by replacing the single precision functions like `expf()`, `sinf()`, `sqrtf()` by `native_exp()`, `native_sin()`, `native_sqrt()`, etc. ..., which provide less numerically-accurate implementations, but with higher performance. We followed this path here as well, but replaced only the single precision functions `sqrtf()` and `expf()` by simplified implementations, yielding reduced-accuracy results, but also requiring fewer floating-point operations.

Furthermore, vectorization of single precision computations on the SX-Aurora VE can be done in two ways: (1) by using vector instructions with up to 256 single-precision elements that are located either in the upper or lower half of the vector register; or (2) by using *packed* vector instructions where each 64-bit vector element of a vector register contains two 32-bit float entities. The later case is called *packed vectorization*, and effectively allows vector lengths of up to 512, with twice the performance of (1).

Unfortunately, packed vectorization has limitations and is more complex to implement in a compiler backend. The vector length must be even, predicated packed vector instructions need two vector mask registers instead of one, and memory access requires several instructions instead of one to account for possible misalignments. Moreover, all operations inside a packed vectorized loop must be executed in single-precision packed mode, otherwise the stream of vector instructions could be disturbed and the performance benefit is lost. Therefore, double-to-float casts, or calls to double-precision mathematical builtin functions (e.g., `ceil()` instead of `ceilf()`), would break packed vectorization. Recent work on the LLVM-VE project [2] has focused on enabling and improving packed vectorization on the VE, and AutoDock's `energy_ia.c` (ligand-ligand) scoring function was used as a benchmark for the progress. Since LLVM-VE's performance using packed vectorization (2) exceeds that of the NEC ncc compiler using the traditional approach (1) for the

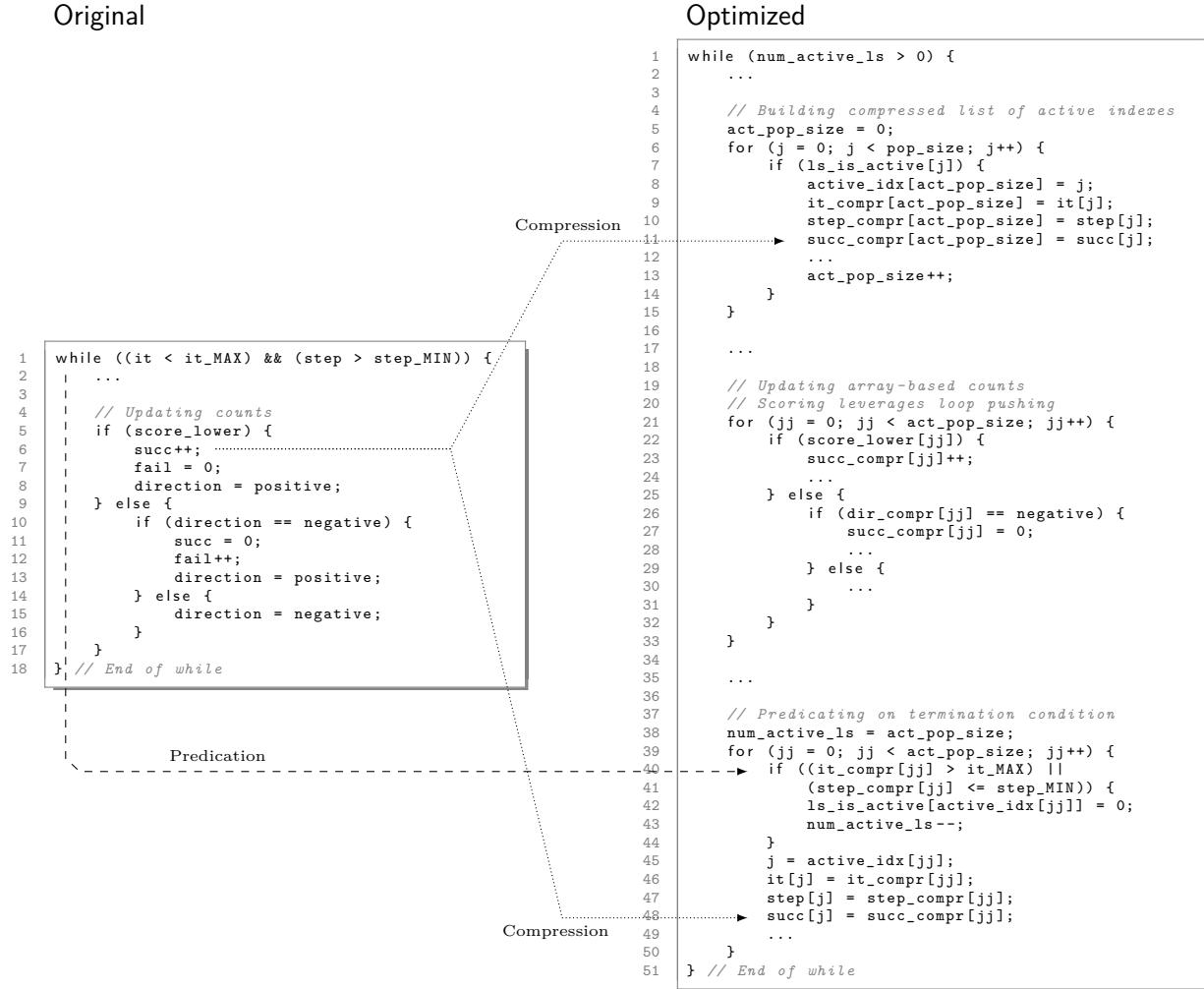


Figure 4. Optimization in Local Search (LS): usage of predication and compression. In the optimized code, predication updates the number of active individuals. An example of compression-based optimization is the replacement of the `succ` scalar variable with the `succ_compr[]` array counterpart. In both cases, the number of successful search attempts is counted. In the optimized code, however, the array compresses data for all active individuals

performance-critical function `energy_ia.c`, we compiled this function with LLVM-VE, while using ncc for the rest of the code.

3. Results and Discussion

For validating the docking functionality, we selected a total of 31 ligand-receptor inputs from the list used for validation in [14]. Table 1 shows a dataset subset. The maximum number of rotatable bonds (N_{rot}) in any of our inputs is eight, as recommended when using the Solis-Wets method as Local Search [23].

For profiling executions on the VE, we used the PROGINFO and FTRACE [17] utilities, both providing a large set of performance counters as well as derived performance metrics. The former provides program execution information, while the latter focuses on functions and user regions. As discussed in Section 2.2, the first major optimization was based on loop pushing. Table 2 compares relevant execution metrics for the `1ig3` input, and is used here to analyze the performance impact of this technique. First, the real time represents the wall-clock elapsed time, while the user time

Table 1. Subset of ligand-receptor inputs with their respective number of rotatable bonds (N_{rot}) and atoms (N_{atom})

Input	1ac8	1hnn	1yv3	1owe	1p62	1n46	1ig3	1t46	2bm2	1mzc
N_{rot}	0	2	2	3	4	5	6	6	7	8
N_{atom}	8	18	23	27	22	28	21	40	33	38

accounts for the time spent by all eight cores in the VE. Since the independent LGA runs are distributed among all VE cores via `#pragma omp parallel for` (Section 2.1), the user time is $\sim 8\times$ that of the real time. Moreover, it can be noted that both real and user times were improved by a factor of ~ 21.3 , while the execution time for vector instructions was reduced $\sim 4.9\times$.

Interestingly, the number of all instruction executions (Inst. Count) was reduced $\sim 51\times$, while the FLOP count is almost the *same*. The program does roughly the same number of floating point operations because it computes the same problem as before, but many of the formerly scalar loops are now vectorized with large vector length (> 200), thus the instruction count reduction in the order of $50\times$. The formerly shorter vector loops with the average vector length of 72 are now executed in longer loops, with the average length of 217, reducing the number of vector instructions approximately by a factor of $3.8\times$.

The streamlined vectorized execution, visible in the heavily increased vector operation ratio from 66.4 % to 99.2 %, and the grown average vector length of 217, lead to an increase of the number of overall operations per second (MOPS), and floating-point operations per second (MFLOPS) by $12.5\times$ (1/0.08) and $20.4\times$ (1/0.049), respectively. While the optimal average vector length would be 256, the value could not be reached in practice due to code divergence in the Local Search section. A further significant improvement of the changed code is shown in the reduction of the Level 1 Cache Miss time, from 44.2 s to 2.1 s. A L1 Cache Miss can only occur in scalar code, and the lowered value reflects the significantly reduced number of scalar instructions executed in the optimized version.

Table 3 summarizes the impact of further optimizations as relative time changes of the total evaluation. Experiments were performed on one VE using the 1ig3 input with $N_{\text{LGA-runs}} = 100$. The relative time change was computed as the percentage of the time change compared to the unoptimized case: $100 \times (t_{\text{optimized}} - t_{\text{unoptimized}})/t_{\text{unoptimized}}$. The first line reflects the gains described by Tab. 2 that lead to the reduction of the compute time by 95.3 %. The following lines show further optimizations of the loop pushing code.

When the VE was switched to NUMA partitioned mode, the multi-process code benefits from the reduced memory network contention and CPU port conflicts. As this problem only impacts the ligand-receptor energy calculation that is dominated by indirect memory accesses, the impact is small (-6.9%) and depends a lot on the specific problem. As NUMA measurements force us to use two processes (on a VE) with 4 cores each, the values in Tab. 3 are compared against similar runs on non-NUMA VEs. Using two processes in non-NUMA mode has a higher overhead than running just one process with 8 cores. Since the NUMA benefits are in the range of this overhead, 2×4 cores with NUMA and 1×8 cores UMA timings are practically the same. A significant improvement could be achieved by the packed vectorization, that in turn, could only be achieved with the LLVM-VE compiler using the RegionVectorizer RV. This change almost doubled the execution speed of the ligand-ligand interaction scoring function, and led to an evaluation time reduction of 36.2 %. At this point, it is important to indicate that, we opted to use a single input

Table 2. Execution metrics of AutoDock-Aurora for the 1ig3 input, before and after applying loop pushing. Information was obtained using NEC PROGINF [17]

Metric	Optimization: loop pushing		Ratio Before / After
	Before	After	
Real Time [sec]	307.5	14.5	21.3
User Time [sec]	2,458.1	115.0	21.4
Vector Time [sec]	510.2	104.0	4.91
Inst. Count	5,085,000,001,257	98,888,607,313	51.4
Vec. Inst. Count	120,865,697,285	32,136,492,289	3.76
FLOP Count	4,982,577,754,822	4,826,280,301,843	1.03
MOPS	6,012.0	75,174.3	0.08
MOPS (Real)	48,082.1	597,857.0	0.08
MFLOPS	2,027.0	41,960.7	0.048
MFLOPS (Real)	16,211.5	333,711.3	0.049
Avg. Vec. Length	71.5	216.9	0.33
V. Op. Ratio [%]	66.4	99.2	0.67
L1 Cache Miss [sec]	44.2	2.1	20.4

(i.e., 1ig3) in order to provide a simple but yet reasonable analysis. Using the full dataset for such analysis would be ideal, but not strictly necessary. The reason is that, e.g., for ligands with larger N_{rot} and N_{atom} , the length of compute-intensive loops is in turn larger, and thus, we can safely expect larger benefits than those for 1ig3 in Tab. 3. Finally, the use of reduced-precision replacements for `sqrtf` and `expf` inside the ligand-ligand interaction scoring function led to a gain of 25.4 % with LLVM-VE, but to a time loss (slowdown!) with the ncc compiler, a sign that the NEC ncc compiler provides fast implementations for these functions.

Table 3. Overview of improvements obtained through various optimizations on one VE using the 1ig3 input. Larger negative values for the relative time change are better. All optimizations below the *loop pushing* line show additional gains (or losses) on top of this vectorized code

Optimization	Rel. time change
Loop pushing and vectorized random generator	-95.3 %
VE in NUMA partitioned mode	-6.9 %
Packed vectorization of <code>energy_ia.c</code> with llvm-ve	-36.2 %
Reduced precision <code>sqrtf</code> and <code>expf</code> with ncc	29.5 %
Reduced precision <code>sqrtf</code> and <code>expf</code> with llvm	-25.4 %

Finally, the execution runtimes of AutoDock-Aurora are compared against those of AutoDock-GPU [23], the state-of-the-art OpenCL-based implementation of AutoDock for GPUs/CPUs. Table 4 lists the accelerator devices equipping the systems employed. We used version v1.1 of AutoDock-GPU, in order to ensure a fair comparison, aiming for having equivalent functionality in both GPU and VE implementations. For all executions of both AutoDock-Aurora and AutoDock-GPU, we set the LS_{rate} to 100 % instead of the default 6 %, as real-world experiments with

AutoDock-GPU typically use the highest practical LS_{rate} value. In this configuration, all members of a population undergo Local Search, and thus, the program has higher chances to produce more-accurate molecular predictions.

Table 4. Technical characteristics of the SX-Aurora VEs, GPUs and CPUs used in the evaluation: base clock frequency (Freq), number of cores (Ncores), FP32 performance (Perf), memory bandwidth (MemBW). GPUs are composed of independent Streaming Multiprocessors (SM). Both CPU platforms posses two sockets each

Characteristics	SX-Aurora		GPU			CPU	
	10B	20B	RTX2070	V100	A100	EPYC 7502	EPYC 7742
Freq [GHz]	1.4	1.6	1.61	1.23	0.76	2.5	2.25
Ncores	8	8	2560	5120	6912	32×2	64×2
Perf [TFLOPS]	4.3	4.9	9.1	14.1	19.5	2.6	4.6
MemBW [GB/s]	1220	1530	448	897	1555	204.8×2	204.8×2
L1 Cache	32 kB (SPU I\$) 32 kB (SPU O\$)	(per SM)	64 kB (per SM)	128 kB (per SM)	192 kB (per SM)	96 kB (per core)	96 kB (per core)
L2 Cache	256 kB (SPU) 128 kB (VPU)	(shared)	4 MB (shared)	6 MB (shared)	40 MB (shared)	512 kB (per core)	512 kB (per core)
L3 Cache	16 MB LCC (shared)	-	-	-	-	128 MB (shared)	256 MB (shared)

As shown in Fig. 5, larger population sizes increase the performance on the VE, but do not impact it as much on any other architectures. We attribute the different performance behavior of AutoDock-GPU to the workload distribution strategy used in OpenCL. In AutoDock-GPU, the population size directly affects the number of spawned OpenCL work-groups ($N_{WG} = N_{pop\text{-size}} \times N_{LGA\text{-run}}$), but has no impact on the execution time of a score evaluation. As described in Section 1.2, the LGA terminates when the number of score evaluations reaches an upper bound (i.e., $N_{score\text{-evals}}^{MAX}$). As a consequence, processing larger populations requires *fewer* iterations per LGA run, and thus compensates for the seemingly bigger workload imposed by the need for more individuals to be processed. The slight increase of runtimes on GPUs/CPPUs for the larger populations is likely due to the synchronization overhead introduced by the additional OpenCL work-groups. On the other hand, in AutoDock-Aurora, larger populations positively impact the performance of the pushed-in loops, enabled by their longer vector lengths (Section 2.2). Since the purpose of molecular docking with genetic algorithms is to test larger number of genetic individuals, we see no disadvantage for other architectures when choosing a large population size that is optimal for the VE.

For the sake of clarity, Fig. 6 shows only the results when using $N_{pop\text{-size}} = 2048$. It can be observed that the VE 20B significantly outperforms the dual socket state-of-the-art AMD EPYC nodes by average factors of $6.5 \times$ ($= 44.3/6.8$) and $3.6 \times$ ($= 25.1/6.8$).

In terms of the underlying semiconductors, the VEs are at the same 16nm-node as the V100 GPU, but have a lower peak performance of $2.8 \times$ ($= 14.1/4.9$, Tab. 4) compared to the GPU. The performance of our implementation is slower than the V100 by a factor of $2.4 \times$ ($= 6.8/2.8$, Fig. 6) and thus comes very close to the theoretical peak performance ceiling. Comparing to the A100 GPU, while its peak only increased by $\sim 1.4 \times$ ($= 19.5/14.1$, Tab. 4) over the V100, the A100's average execution time shows much better performance than the latter: $\sim 2.5 \times$ ($= 3.8/1.5$). We

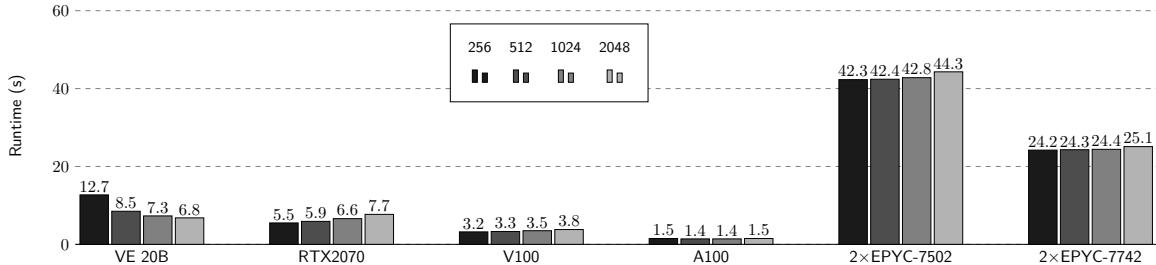


Figure 5. Geometric mean of execution runtimes over 31 inputs, comparing the impact of the chosen population size: $N_{\text{pop-size}} = \{256, 512, 1024, 2048\}$. AutoDock-Aurora was executed on the VE 20B, while AutoDock-GPU v1.1 on the GPUs and CPUs. In all executions: $N_{\text{LGA-runs}} = 100$, $\text{LS}_{\text{rate}} = 100\%$. Other parameters were left at default values

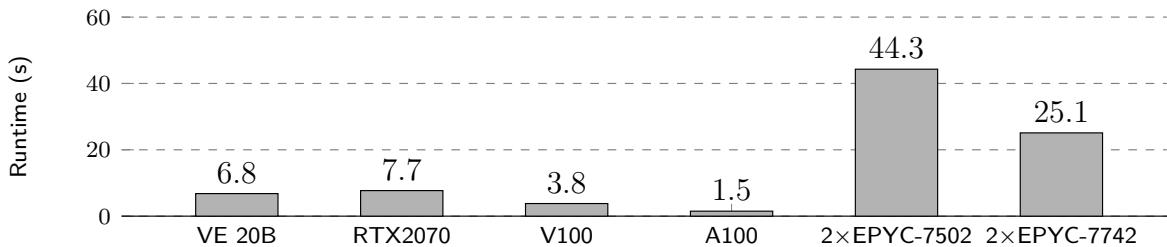


Figure 6. Geometric mean of execution runtimes over 31 inputs. AutoDock-Aurora was executed on the VE 20B, while AutoDock-GPU v1.1 on the GPUs and CPUs. In all executions: $N_{\text{pop-size}} = 2048$, $N_{\text{LGA-runs}} = 100$, $\text{LS}_{\text{rate}} = 100\%$. Other parameters were left at default values

assume that the higher performance and bandwidth capabilities of the A100 are the main reason for the fastest executions on this platform.

As reported in [23], AutoDock-GPU achieves faster executions on GPUs than on CPUs, which is attributed to the more suitable mapping of OpenCL elements onto the underlying hardware. On CPUs, each OpenCL work-group is executed by a single CPU core, and thus, work-items (threads) are executed serially [11]. On GPUs, work-groups and work-items are executed in parallel by the fine-grain GPU streaming multiprocessors.

4. Related Work

Studies on benchmarking the performance of the SX-Aurora using various applications are reported as follows. Komatsu et al. used standard benchmarks and a tsunami numerical simulation code [13]. These authors introduced a performance model based on different *Byte per FLOP* (B/F) rates to analyze the bottleneck causes in applications. Onodera et al. optimized the Himeno benchmark, which solves the Poisson's equation using the Jacobi iteration method. The vector systems used in Onodera et al.'s evaluation were configured with a single and up to eight VEs [21]. While experiments in [13] and [21] were performed on SX-Aurora Type 10B, Egawa et al. used *second-generation* Type 20B devices to benchmark their optimization strategies on different scientific applications [6]. Moreover, Egawa et al. refined the model introduced by Komatsu et al. in [13]. This newly-proposed model considers a *possible* peak performance (FLOP/s), which can be determined by the FMA instruction rate of applications and sustained memory bandwidth (B/s), instead of employing their *peak* values.

Furthermore, Takizawa et al. proposed an OpenCL-like offload programming framework for SX-Aurora [28]. As the OpenCL execution model (originally designed for GPUs) does not fit well

for all compute architectures (e.g., FPGAs, VEs), this framework allows the usage of different programming languages for implementing the host and device code. Particularly, the host code can be written in OpenCL C/C++, while the device code in standard C++. In general, using such a framework would potentially reduce the porting effort from existing OpenCL code as in our case. However, we opted to implement both host and device code in standard C++, by explicitly invoking VEO APIs to interact with the VE, and writing vectorizable loops. In this manner, we avoided relying on external APIs, and exploited the compiler’s capability of automatic vectorization.

To the best of our knowledge, our work here on AutoDock-Aurora is the *first one* leveraging vector computing for molecular docking. The closest studies related to ours are the hardware-accelerated implementations reported in [25]. From these studies, accelerator devices such as CPUs, GPUs, and even FPGAs, have been used in single computing nodes. Moreover, parallelization strategies are not only based on the docking algorithm under analysis, but also on hardware-specific aspects of the target device. Here, we report previous studies describing AutoDock.

Regarding GPUs, AutoDock has its official release rebranded as AutoDock-GPU. It has been originally written in OpenCL [23], and afterwards, ported from the original OpenCL to CUDA [14]. This CUDA implementation was used as the docking engine for COVID-19 research on the Summit supercomputer, where OpenCL was not supported. Furthermore, AutoDock has been ported to FPGAs as well, where various efforts differ in the design approaches they adopted. Pechan et al. followed the *traditional* development (for FPGAs) by describing the docking functionality in terms of low-level transfers between hardware registers and synchronous logic design [22]. On the other hand, Solis-Vasquez et al. followed a *high-level* design approach based on OpenCL to develop `ocladock-fpga` [27]. In contrast to AutoDock-GPU, which parallelizes over multiple *data items* (i.e., genotypes), `ocladock-fpga` executes multiple *tasks* concurrently. Furthermore, it relies on pipelined hardware logic and custom memory hierarchies. In terms of performance, `ocladock-fpga` on an Arria-10 FPGA runs $\sim 2\times$ faster than the original AutoDock on a CPU, but it is still significantly slower compared to AutoDock-GPU. Hence, it is not being deployed for realistic docking problems.

Conclusions and Future Work

In this work, we have ported and optimized AutoDock onto the SX-Aurora TSUBASA platform. The molecular docking application consists of a Genetic Algorithm coupled with a Local Search part, which has a divergent flow with frequent calls to compute-intense score evaluations. To the best of our knowledge, this is the *first* molecular docking and Genetic Algorithm implementations for the SX-Aurora TSUBASA.

Regarding the porting process, the programming framework provided by NEC enabled a smooth experience. However, the optimization was much more involved, requiring the combination of a number of different strategies. Of these, loop pushing improved the performance significantly, but required more code refactoring in the Local Search part. Basically, the original SIMD nature of the Local Search, treating individuals in different threads, is now expressed in an explicit, vectorizable loop. Regarding the floating-point arithmetic, the computations offloaded to the VE were expressed using single-precision only, and in turn, were vectorized in *packed mode* for higher performance. Furthermore, we explored mixed compilation, employing both LLVM-VE and NEC compilers for the scoring-function components. As a result, our implementation is in average $3.6\times$ faster than 128-core CPU servers, while being competitive to V100 and A100 GPUs.

As a future work, we will incorporate alternative methods for the Local Search part. Namely, ADADELTA [30], which compared to the Solis-Wets method examined here performs more complex computations, but yields molecular predictions of higher quality [23]. Moreover, we plan to analyze the achieved efficiency using performance models, e.g., those based on Byte/FLOP ratios, as proposed in [6, 13].

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. FightAIDS@Home. <https://www.worldcommunitygrid.org/research/faah/overview.do> (2021), accessed: 2021-06-01
2. LLVM-VE project GitHub repository. <https://github.com/sx-aurora-dev/llvm-project> (2021), accessed: 2021-05-31
3. OpenPandemics: COVID-19. <https://www.worldcommunitygrid.org/research/opn1/overview.do> (2021), accessed: 2021-06-01
4. VEDA GitHub repository. <https://github.com/SX-Aurora/veda> (2021), accessed: 2021-05-19
5. Cramer, T., Römmer, M., Kosmynin, B., *et al.*: OpenMP Target Device Offloading for the SX-Aurora TSUBASA Vector Engine. In: Wyrzykowski, R., Deelman, E., Dongarra, J.J., *et al.* (eds.) Parallel Processing and Applied Mathematics - 13th Int. Conf., PPAM 2019, Bialystok, Poland, Sept. 8-11, 2019, Revised Selected Papers, Part I. Lecture Notes in Computer Science, vol. 12043, pp. 237–249. Springer (2019). https://doi.org/10.1007/978-3-030-43229-4_21
6. Egawa, R., Fujimoto, S., Yamashita, T., *et al.*: Exploiting the potentials of the second generation SX-Aurora TSUBASA. In: 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, PMBS@SC 2020, Atlanta, GA, USA, Nov. 12, 2020. pp. 39–49. IEEE (2020). <https://doi.org/10.1109/PMBS51919.2020.00010>
7. Fischer, E.: Einfluss der Configuration auf die Wirkung der Enzyme. II. Berichte der deutschen chemischen Gesellschaft 27(3), 3479–3483 (1894). <https://doi.org/10.1002/cber.189402703169>
8. Focht, E.: VEO and PyVEO: Vector Engine Offloading for the NEC SX-Aurora Tsubasa. In: Resch, M.M., Kovalenko, Y., Bez, W., *et al.* (eds.) Sustained Simulation Performance 2018 and 2019. pp. 95–109. Springer (2020). https://doi.org/10.1007/978-3-030-39181-2_9
9. Halperin, I., Ma, B., Wolfson, H., Nussinov, R.: Principles of docking: An overview of search algorithms and a guide to scoring functions. Proteins: Struct., Funct., Bioinf. 47(4), 409–443 (2002). <https://doi.org/10.1002/prot.10115>
10. Huey, R., Morris, G.M., Olson, A.J., Goodsell, D.S.: A semiempirical free energy force field with charge-based desolvation. J. Comput. Chem. 28(6), 1145–1152 (2007). <https://doi.org/10.1002/jcc.20634>

11. Kaeli, D., Mistry, P., Schaa, D., Zhang, D.P.: Heterogeneous Computing with OpenCL 2.0. Morgan Kaufmann, 3 edn. (2015)
12. Ke, Y., Agung, M., Takizawa, H.: neoSYCL: a SYCL implementation for SX-Aurora TSUBASA. In: Hwang, S., Yeom, H.Y. (eds.) HPC Asia 2021: The Int. Conf. on High Performance Computing in Asia-Pacific Region, Virtual Event, Republic of Korea, Jan. 20-21, 2021. pp. 50–57. ACM (2021). <https://doi.org/10.1145/3432261.3432268>
13. Komatsu, K., Momose, S., Isobe, Y., *et al.*: Performance Evaluation of a Vector Supercomputer SX-Aurora TSUBASA. In: SC18: Int. Conf. for High Performance Computing, Networking, Storage and Analysis. pp. 685–696. IEEE (2018). <https://doi.org/10.1109/SC.2018.00057>
14. LeGrand, S., Scheinberg, A., Tillack, A.F., *et al.*: GPU-Accelerated Drug Discovery with Docking on the Summit Supercomputer: Porting, Optimization, and Application to COVID-19 Research. In: BCB ’20: 11th ACM Int. Conf. on Bioinformatics, Computational Biology and Health Informatics, Virtual Event, USA, Sept. 21-24, 2020. pp. 43:1–43:10. ACM (2020). <https://doi.org/10.1145/3388440.3412472>
15. Liu, J., Wang, R.: Classification of Current Scoring Functions. *J. Chem. Inf. Model.* 55(3), 475–482 (2015). <https://doi.org/10.1021/ci500731a>
16. 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. *J. Comput. Chem.* 19(14), 1639–1662 (1998). [https://doi.org/10.1002/\(SICI\)1096-987X\(19981115\)19:14<1639::AID-JCC10>3.0.CO;2-B](https://doi.org/10.1002/(SICI)1096-987X(19981115)19:14<1639::AID-JCC10>3.0.CO;2-B)
17. NEC: PROGINF/FTRACE User Guide. https://www.hpc.nec/documents/sdk/pdfs/g2at03e-PROGINF_FTRACE_User_Guide_en.pdf (2018), accessed: 2021-05-31
18. NEC: SX-Aurora TSUBASA Performance Tuning Guide. https://www.hpc.nec/documents/guide/pdfs/AuroraVE_TuningGuide.pdf (2020), accessed: 2021-05-29
19. Nishimura, T.: Tables of 64-bit Mersenne Twisters. *ACM Trans. Model. Comput. Simul.* 10(4), 348–357 (2000). <https://doi.org/10.1145/369534.369540>
20. Noack, M., Focht, E., Steinke, T.: Heterogeneous active messages for offloading on the NEC SX-Aurora TSUBASA. In: IEEE Int. Parallel and Distributed Processing Symposium Workshops, IPDPSW 2019, Rio de Janeiro, Brazil, May 20-24, 2019. pp. 26–35. IEEE (2019). <https://doi.org/10.1109/IPDPSW.2019.00014>
21. Onodera, A., Komatsu, K., Fujimoto, S., *et al.*: Optimization of the Himeno benchmark for SX-Aurora TSUBASA. In: Wolf, F., Gao, W. (eds.) Benchmarking, Measuring, and Optimizing - Third BenchCouncil Int. Symposium, Bench 2020, Virtual Event, Nov. 15-16, 2020, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12614, pp. 127–143. Springer (2020). https://doi.org/10.1007/978-3-030-71058-3_8
22. Pechan, I., Fehér, B., Bérces, A.: FPGA-based acceleration of the AutoDock molecular docking software. In: Proc. of the 6th Conf. on Ph.D. Research in Microelectronics Electronics. pp. 1–4. IEEE (2010), <https://ieeexplore.ieee.org/document/5587139>

23. Santos-Martins, D., Solis-Vasquez, L., Tillack, A.F., *et al.*: Accelerating AutoDock4 with GPUs and Gradient-Based Local Search. *J. Chem. Theory Comput.* 17(2), 1060–1073 (2021). <https://doi.org/10.1021/acs.jctc.0c01006>
24. Solis, F.J., Wets, R.J.B.: Minimization by Random Search Techniques. *Math. Oper. Res.* 6(1), 19–30 (1981). <https://doi.org/10.1287/moor.6.1.19>
25. Solis-Vasquez, L.: Accelerating Molecular Docking by Parallelized Heterogeneous Computing - A Case Study of Performance, Quality of Results, and Energy-Efficiency using CPUs, GPUs, and FPGAs. Ph.D. thesis, Technical University of Darmstadt, Germany (2019). <https://doi.org/10.25534/tuprints-00009288>
26. Solis-Vasquez, L., Koch, A.: A performance and energy evaluation of opencl-accelerated molecular docking. In: McIntosh-Smith, S., Bergen, B. (eds.) *Proc. of the 5th Int. Workshop on OpenCL, IWOCL 2017*, Toronto, Canada, May 16-18, 2017. pp. 3:1–3:11. ACM (2017). <https://doi.org/10.1145/3078155.3078167>
27. Solis-Vasquez, L., Koch, A.: A Case Study in Using OpenCL on FPGAs: Creating an Open-Source Accelerator of the AutoDock Molecular Docking Software. In: *Proc. of the 5th Int. Workshop on FPGAs for Software Programmers (FSP)*. pp. 1–10. VDE Verlag (2018), <https://ieeexplore.ieee.org/document/8470463>
28. Takizawa, H., Shiotsuki, S., Ebata, N., Egawa, R.: An OpenCL-like offload programming framework for SX-Aurora TSUBASA. In: *20th Int. Conf. on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2019*, Gold Coast, Australia, Dec. 5-7, 2019. pp. 282–288. IEEE (2019). <https://doi.org/10.1109/PDCAT46702.2019.00059>
29. Wang, Z., Sun, H., Yao, X., *et al.*: Comprehensive evaluation of ten docking programs on a diverse set of protein–ligand complexes: the prediction accuracy of sampling power and scoring power. *Phys. Chem. Chem. Phys.* 18(18), 12964–12975 (2016). <https://doi.org/10.1039/C6CP01555G>
30. Zeiler, M.D.: ADADELTA: An Adaptive Learning Rate Method. *arXiv abs/1212.5701* (2012)

First Experience of Accelerating a Field-Induced Chiral Transition Simulation Using the SX-Aurora TSUBASA

Shinji Yoshida¹, Arata Endo², Hirono Kaneyasu³, Susumu Date²

© The Authors 2021. This paper is published with open access at SuperFri.org

An analysis method based on the Ginzburg-Landau equation for the superconductivity is applied to the field-induced chiral transition simulation (FICT). However, the FICT is time consuming because it takes approximately 10 hours on a single SX-ACE vector processor. Moreover, the FICT must be repeatedly performed with parameters changed to understand the mechanism of the phenomenon. The newly emerged SX-Aurora TSUBASA, the successor of the SX-ACE processor, is expected to provide much higher performance to the programs executed on the SX-ACE as is. However, the SX-Aurora TSUBASA processor has changed its architecture of compute nodes and gives users three different execution models, which leads to users' concerns and questions in terms of how three execution models should be selectively used. In this paper, we report the first experience of using the SX-Aurora TSUBASA processor for the FICT. Specifically, we have developed three implementations of the FICT corresponding to the three execution models suggested by the SX-Aurora TSUBASA. For acceleration of the FICT, improvement of the vectorization ratio in the program execution and the efficient transfer of data to the general purpose processor as the vector host from the vector processor as the vector engine is explored. The evaluation in this paper shows how acceleration of the FICT is achieved as well as how much effort of users is required.

Keywords: SX-Aurora TSUBASA, OS Offload, VH Call, VEO, vectorization ratio.

Introduction

The simulation based on the Ginzburg-Landau equation [4] is the standard way to analyze the superconductivity in an external magnetic field. The analysis method is also applied to study a field-induced chiral phenomenon in the superconductivity [6, 7, 12], that is, the field-induced chiral transition simulation (FICT). In the chiral superconductivity, the superconducting electron pairs have the orbital magnetization [14], which makes the spontaneous intrinsic fields.

The chiral orbital magnetization paramagnetically couples with the external magnetic field, and it induces a field-induced chiral transition with generating the paramagnetic supercurrent [6, 7]. The field-induced chiral phenomena do not occur in the non-chiral state which generates the screening supercurrents diamagnetic to the external field with breaking the superconducting electron pairs. Thus, the field-induced chiral phenomena appear by the paramagnetic coupling of the orbital magnetization with the external field, as the chiral nature.

Such field-induced chiral phenomena are expected in candidates of chiral superconductors [5, 9, 16], such as the ruthenium oxide Sr₂RuO₄ (SRO) [11]. The field-induced chiral phenomena are theoretically clarified in the inhomogeneous superconductivity arising in SRO near interfaces of the micro-meter size Ru-metal inclusions embedded in the eutectic SRO-Ru [10], by using the field-induced chiral transition simulation (FICT) [6, 7].

However, the FICT is time consuming and furthermore has to be repeatedly performed with its parameters changed. Moreover, reducing the differential width in the Ginzburg-Landau equation is needed to analyze the dependence of the superconducting order parameter and the vector potential on distances in more detail, and then more computation is required. In addition

¹Graduate School of Information Science and Technology, Osaka University, Osaka, Japan

²Cybermedia Center, Osaka University, Osaka, Japan

³Department of Science, University of Hyogo, Hyogo, Japan

to this, the lowering of temperature makes the superconductivity spread far away from the interface in the FICT. It needs the increase of mesh-number for the distance to analyze the chiral stability in the lower temperatures. At this point, the reduction of real time by the acceleration is effective for the analysis extended to the lower temperature. Thus, the acceleration makes it possible to clarify the field dependence of chiral phenomena in the lower temperatures in detail.

Until now, the SX-ACE system [3] has been used for performing the FICT, because the SX-ACE system gives a higher performance in comparison with general purpose processors. However, the newly-emerged SX-Aurora TSUBASA [2, 8], the successor of the SX-ACE, has changed the architecture of the compute node. The system architecture of the SX-Aurora TSUBASA consists of vector engines (VEs), which accelerate computation using vectorization and a vector host (VH), which is a standard x86 server and executes the operating system's tasks. This system architecture provides researchers with three types of execution models which differ in terms of how a program is executed on VE and VH. The FICT is accelerated on the SX-Aurora TSUBASA compute node by modifying the FICT that conforms to the best model that makes the FICT the fastest. However, the best execution model for the FICT is not clarified. Also, what operations are needed to modify the FICT that conforms to each execution model has not been clarified. From the perspective above, we report the first experience of accelerating the FICT using the SX-Aurora TSUBASA. We believe that the contribution of this paper is to present a case study of comparing the FICT based on the three execution models.

The rest of this paper is organized as follows. Section 1 shows our approach to accelerate the FICT after briefly explaining the FICT and the SX-Aurora TSUBASA. Section 2 presents our methods for accelerating the FICT. In Section 3, we evaluate the FICT accelerated with our acceleration approach on the SX-Aurora TSUBASA. In Conclusion, this paper is concluded.

1. Approach to Accelerate the FICT

This section first describes an overview of the FICT and the SX-Aurora TSUBASA. Secondly, we present our approaches to accelerate the FICT on the SX-Aurora TSUBASA.

1.1. Overview of the FICT

The chiral superconducting state is denoted with two components of the superconducting order parameter, while the non-chiral state is represented with one component [14]. The chiral transition corresponds to the transition to the two components state by yielding the second component in the addition to the one component. Thus, the field-induced chiral transition occurs by inducing the second component in the application of the external field [6, 7].

The FICT computes both the two order parameter components and the vector potential. Moreover, paramagnetic and screening supercurrents are calculated from the order parameter and the vector potential which are obtained as results of the equation. In the FICT, the strength of an external magnetic field is increased to simulate how the strength of the external magnetic field affects the superconducting state at the fixed temperatures respectively. The superconducting is computed at each strength of the external magnetic field at the fixed temperatures. That is, the dependence of two order parameter components and the vector potential on the distance is calculated iteratively under varying parameters for an external magnetic field and a temperature.

Figure 1 shows a flow chart of the FICT. When the FICT starts, the initial parameters regarding the superconducting order parameter are given. Next, the Ginzburg-Landau equation is computed, and then the result of the equation is written to a file, which repeats until the strength of the external magnetic field reaches a set value. After that, the parameters regarding the superconducting order parameter are initialized to the initial parameters and the strength of the temperature increases. When the strength of the external magnetic field and that of the temperature reach their values which we set, the FICT finishes.

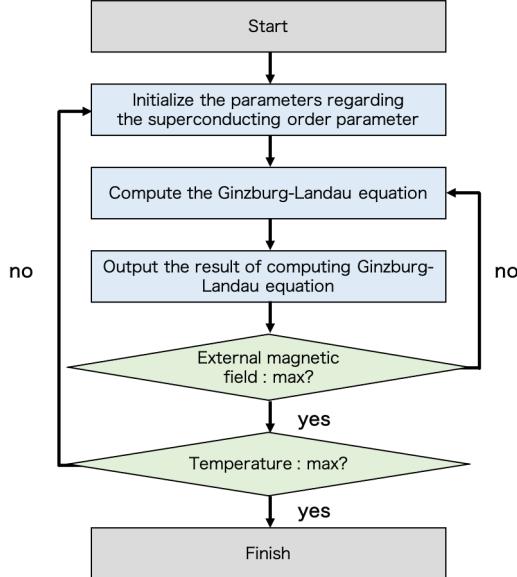


Figure 1. Flow chart of the FICT

1.2. Overview of the SX-Aurora TSUBASA

Figure 2 shows an example system architecture of the SX-Aurora TSUBASA system. A compute node where the SX-Aurora TSUBASA processor is hosted is composed of vector host (VH) and vector engine (VE). In the inside of the compute node, VE and VH are connected to a PCI Bus as shown in Fig. 2. In this architecture, the computation is performed on VE and VH works supplementarily with VE. For example, when VE requires I/O-related system calls, VH performs such system calls instead of VE.

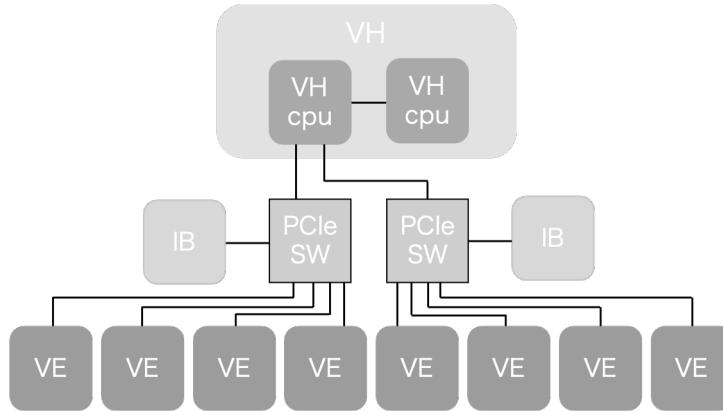


Figure 2. An example system architecture of the SX-Aurora TSUBASA system (A300-8)

Accompanied the change in the architecture, the SX-Aurora TSUBASA offers three execution models. In this paper, we refer to these three models: standard (OS Offload) execution model, VH Call execution model, and Vector Engine Offload (VEO) execution model. Figure 3 compares these three execution models.

- In the standard execution model, a program is started on VH. After that, however, it is mostly executed on VE. Only when the program needs to accomplish operating system tasks, system calls are automatically offloaded to VH from VE. The advantage of this execution model is that when researchers write the program, they do not need to know the system architecture of the SX-Aurora TSUBASA. Furthermore, most of programs which are executed on the SX-ACE system can be easily ported to the SX-Aurora TSUBASA system without much modification. On the other hand, the disadvantage might be that researchers have no way of explicitly using VH when researchers use this model.
- In the VH Call execution model, a program is initially started on VH. After that, it is mainly executed on VE. Basically, this execution model is the same as the standard execution model explained above but it allows users to explicitly invoke the VH Call for requesting the processing of some program portion on VH. Thus, the system calls are automatically offloaded to VH. In addition, some portion of calculations can be offloaded explicitly by users to VH from VE. The merit of the VH Call execution model is that a program containing scalar-friendly calculations is executed faster in the VH Call execution model than in the standard execution model. On the other hand, the disadvantage of this execution model is that researchers must have a fairly detailed knowledge of the SX-Aurora TSUBASA composed of VH and VE as well as the program semantics regarding this execution model.
- In the VEO execution model, a program is initially started and mainly executed on VH. Researchers can explicitly offload some portion of the program to VE from VH. The advantage and disadvantage of the VEO execution model are almost the same as the VH Call execution model. Regarding the disadvantage, the program to be executed on VH must be written in C unlike the VH Call execution model.

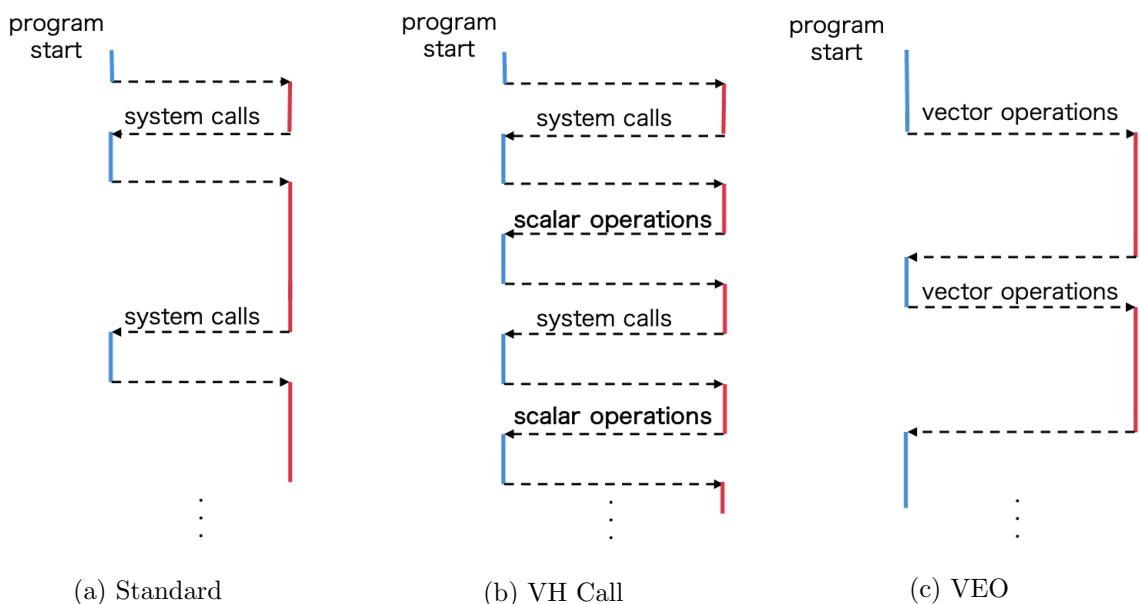


Figure 3. Three execution models of the SX-Aurora TSUBASA

1.3. Computational Characteristics of the FICT

We first describe the computational characteristics of the FICT as obtained through the program execution analysis. Next, we analyze the code that outputs the result of the Ginzburg-Landau equation.

1.3.1. FICT execution analysis information

Analysis information on program execution is obtained with a performance analysis tool named PROGINF [13], which is shipped with the SX-Aurora TSUBASA. Table 1 contains Real Time (the program execution time), Vector Time (the total time taken to execute vector operations in the program execution) and V. Op. Ratio (the vectorization ratio which is the ratio of vector operations in the program to all operations in the program) obtained as the analysis information on the FICT execution.

Table 2 shows the analysis information on each function composing the FICT with a performance analysis tool named FTRACE [13] for the SX-Aurora TSUBASA, which shows Frequency (the call count of each function in program execution), EXCLUSIVE TIME (the execution time of each function), V. Op. Ratio (the vectorization ratio), REQ. B/F (the B/F rate (the memory bandwidth per the performance) needed by the program) and PROC.NAME (the function name). In the case that SYS. B/Y (the rate of the highest observed memory bandwidth of the system to the observed peak performance of the system) is smaller than REQ. B/F of the program, the performance of the system does not achieve the peak performance of the system. Although SYS. B/F of SX-Aurora TSUBASA is under REQ. B/F of the FICT, SYS. B/F of SX-Aurora TSUBASA is higher than SYS. B/F of scalar processors [2]. Therefore, the FICT is suited for SX-Aurora TSUBASA. Also, according to the results, the vectorization ratio of the GRAD subroutine that computes the Ginzburg-Landau equation is lower than the MAIN function. Also, the execution time of the GRAD subroutine itself is approximately five times longer than the execution time of the MAIN program.

Table 1. FICT execution analysis information from PROGINF

Topic	Value
Real Time (sec)	1413.805725
Vector Time (sec)	418.584454
V. Op. Ratio (%)	94.480496

Table 2. FICT execution analysis information from FTRACE

Frequency	EXCLUSIVE TIME [sec] (%)	V. Op. Ratio	REQ. B/F	PROC.NAME
474965635	1416.492 (82.9)	92.43	1.86	GRAD
1	291.970 (17.1)	98.22	1.52	MAIN
474965636	1708.462 (100.0)	94.00	1.76	total

1.3.2. I/O characteristic of the FICT

Figure 4 shows an I/O-intensive portion contained in the FICT. The FICT is written in Fortran. When `WRITE` at line 10 is executed, the data written at lines 11–14 is transferred to VH from VE (data transfer) and then outputted to a file on VH. Data transfer occurs every time `WRITE` is executed. When the FICT is executed, a loop at lines 1–17 is executed 2 times, a loop at lines 3–16 is executed 51 times and a loop at lines 5–15 is executed 240 times. Therefore, when the FICT is executed, data transfer occurs 24,480 times.

```

1      DO nt=ntmin,ntmax,ndt
2          ... etc. ...
3      DO nh=nhmin,nhmax,ndh
4          ... computing the Ginzburg-Landau equation ...
5      DO nx=1,N
6          x=dx*0.5d0*(dbe(nx)+dbe(nx-1))
7          ... etc. ...
8          ua=0.5d0*(ay(nx)+ay(nx-1))
9          da=(ay(nx)-ay(nx-1))/dx
10         WRITE(1110000000+100000*nt+nh,*)
11         & x,x0,x1
12         & ,ev(nx-1),et(nx-1),dabs(ev(nx-1)),dabs(et(nx-1))
13         ... etc. ...
14         & ,DABS(g*(-k4*uv*dt))
15     END DO
16     END DO
17   END DO

```

Figure 4. A portion of I/O operations in the FICT code

2. Acceleration Approach

To understand which execution model is effective for the acceleration of the FICT, we have modified this original FICT code to conform to the three execution models suggested by the SX-Aurora TSUBASA architecture. In this section, we summarize how we attempted to accelerate the original FICT code, by focusing just on the improvement of the vectorization ratio and the efficiency improvement of I/O operations from VE to VH.

2.1. Improving the Vectorization Ratio

The vectorization ratio is an important criterion in achieving higher performance on vector processors. Figure 5 is the portion of the original GRAD subroutine. As shown, the loop from lines 1–4 and the loop from lines 7–10 seem to be easily combined as a single loop. According to the developer of the FICT, lines 5 and 6 remained on purpose to keep the readability and maintainability of the FICT code. There sometimes exists code structures that prevent the acceleration of execution codes in actual users' programs and sometimes researchers as users do not want the code to be modified because the code itself could lose the readability and maintainability.

In the FICT code, from a developer's perspective, two loop structures exist that prevent vectorization. In this paper, each of the two loop structures was combined so that the vectorization ratio was improved. This acceleration approach was commonly conducted in all our three different implementations based on the standard, VH Offload and VEO execution models.

```

1 DO nx=1,N
2   fv(nx)=0.0d0
3   ... etc. ...
4 END DO
5 uv=0.5d0*(ev(1)+ev(0))
6   ... etc. ...
7 DO nx=1,N
8   uv=0.5d0*(ev(nx)+ev(nx-1))
9   ... etc. ...
10 END DO

```

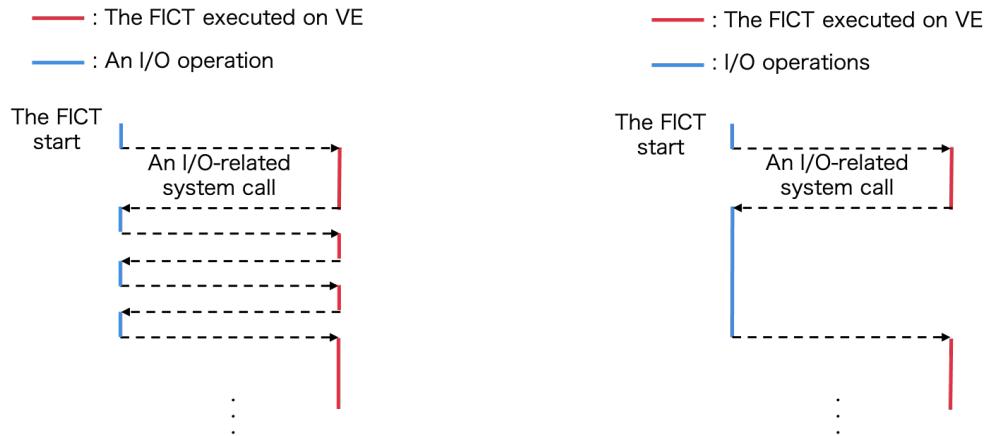
Figure 5. A portion of the original GRAD subroutine code

2.2. Efficiency Improvement of Data Transfer

Under the SX-Aurora TSUBASA system architecture composed of VH and VE, the I/O operations heavily affect the total performance of the application execution. This is because the vector processor as VE is expected to be the main processor for the application execution and the general purpose processor as VH is considered a helper processor. More specifically, I/O operations from the code running on VE trigger I/O-related system calls in the operating system running on VH. At the same time, VE must wait for the completion of system calls on VH. For this reason, performing the efficient I/O operations from the code running on VE by reducing the number of I/O operations helps developers shorten the total execution time of their application codes.

A possible approach to realize such efficient I/O operations for acceleration is to simply reduce the number of data transfers from VE to VH. In Fig. 7, which shows a portion of the original FICT code, an I/O operation at lines 10–13 triggers an I/O-related system call (Fig. 6a). We must modify this procedure as shown in Fig. 6b so that many I/O operations are not triggered by an I/O-related system call. For this modification, we moved the I/O operations part at lines 10–13 to the inside of the loop at lines 1–17 as shown in Fig. 4.

In the following subsections, how the reduction of data transfers is realized is explained in each execution model in addition to the above modification.



(a) A portion of the FICT code before modification (b) A portion of the FICT code after modification

Figure 6. Procedures of I/O operations

```

1      DO nt=ntmin,ntmax,ndt
2          ... etc. ...
3      DO nh=nhmin,nhmax,ndh
4          ... computing the Ginzburg-Landau equation ...
5          DO nx=1,N
6              x=dx*0.5d0*(dble(nx)+dble(nx-1))
7              ... etc. ...
8              ua=0.5d0*(ay(nx)+ay(nx-1))
9              da=(ay(nx)-ay(nx-1))/dx
10             WRITE(1110000000+100000*nt+nh,*) 
11             & ,x
12             ... etc. ...
13             & ,DABS(g*(-k4*uv*dt))
14         END DO
15     END DO
16     CLOSE(1110000000+100000*nt+nh)
17 END DO

```

Figure 7. A portion of the FICT code before modification

```

1      DO nt=ntmin,ntmax,ndt
2          ... etc. ...
3      DO nh=nhmin,nhmax,ndh
4          ... computing the Ginzburg-Landau equation ...
5          DO nx=1,N
6              x=dx*0.5d0*(dble(nx)+dble(nx-1))
7              ... etc. ...
8              ua=0.5d0*(ay(nx)+ay(nx-1))
9              da=(ay(nx)-ay(nx-1))/dx
10             w(1,nx,nh)=x
11             ... etc. ...
12             & ,w(63,nx,nh)=DABS(g*(-k4*uv*dt))
13         END DO
14     END DO
15     ... outputting the results ...
16 END DO

```

Figure 8. A portion of the FICT code after modification

2.2.1. Modification in the standard execution model

The OS Offload execution model is the standard execution model of the SX-Aurora TSUB-ASA architecture. The merit of this standard execution model is that it is designed to be executed as the as-is code running on the SX-ACE system without requiring developers to be aware of the complex architecture composed of VH and VE. However, the demerit of this execution model is that it does not allow developers to directly touch the underlying low-level I/O mechanisms. In the standard execution model, further efforts could not be carried out.

2.2.2. Modification in the VH Call execution model

To utilize the VH Call programming semantics, the developers must write each function to be executed on VH and VE in a separate file. Also, the developers must write such functions on VH and VE conforming to the program semantics regarding the VH Call execution model. A

suite of dedicated functions as shown in lines 15–18 in Fig. 10 are prepared so that the developers utilize the VH Call execution model. The files executed on VE and written in Fortran must be complied with *nfort*, and the ones on VH with *gfortran*. Figure 9 shows an example of Makefile for this execution model.

Figure 10 is the code fragmentation to be executed on VE, and Fig. 11 on VH. In lines 15–18 in Fig. 10, results of computing the Ginzburg-Landau equation are transferred to VH from VE. After that, these results are output to files on VH (Fig. 11). The result of executing this code portion is the same result when lines 10–13 in Fig. 7 are executed.

```
all: VE VH
VE: a fortran file name executed on VE
    nfort -o $@ $^ -lvhcall_fortran -w -report-all
VH: a fortran file name executed on VH
    gfortran -o $@ $^ -fpic -shared
```

Figure 9. Makefile in the VH Call execution model

```
1      ... configuration for using VH Call ...
2      DO nt=ntmin,ntmax,ndt
3          ... etc. ...
4      DO nh=nhmin,nhmax,ndh
5          ... computing the Ginzburg-Landau equation ...
6          DO nx=1,N
7              x=dx*0.5d0*(dble(nx)+dble(nx-1))
8              ... etc. ...
9              da=(ay(nx)-ay(nx-1))/dx
10             w(1,nx,nh)=x
11             ... etc. ...
12             w(63,nx,nh)=DABS(g*(-k4*uv*dt))
13         END DO
14     END DO
15     ir=fvhcall_args_set(ca,fvhcall_intent_inout,1,w)
16     ir=fvhcall_args_set(ca,fvhcall_intent_inout,2,nt)
17     ir=fvhcall_invoke_with_args(sym, ca)
18     CALL fvhcall_args_clear(ca)
19 END DO
```

Figure 10. A portion of the FICT code on VE in the VH Call execution model

```
1      SUBROUTINE vh_write(w,nt)
2      DO nh=nhmin,nhmax,ndh
3          DO nx=1,N
4              WRITE(1110000000+100000*nt+nh,*)
5              & ,w(1,nx,nh)
6              ... etc. ...
7              & ,w(63,nx,nh)
8          END DO
9          CLOSE(1110000000+100000*nt+nh)
10     END DO
11 END SUBROUTINE vh_write
```

Figure 11. A portion of the FICT code on VH in the VH Call execution model

2.2.3. Modification in the VEO execution model

Likewise in the case of the VH Call execution model, to utilize the VEO programming semantics, the developers must write each function to be executed on VH and VE in a separate file. Unlike the VH Call execution model, the files to be executed on VH must be written in C. Also, the developers must write such functions on VH and VE conforming to the program semantics regarding the VEO execution model. A suite of dedicated functions as shown in lines 4–8 in Fig. 13 are prepared so that the developers utilize the VEO execution model. The files executed on VE and written in Fortran must be complied with *nfort*, and the ones on VH and in C with *gcc*. Figure 12 shows an example of Makefile for this execution model.

Figure 13 is the code fragmentation to be executed on VH, and Fig. 14 is on VE. In lines 4–8 in Fig. 13, parameters which are required to compute the Ginzburg-Landau equation and store results of this computation are transferred to VE. After that, the Ginzburg-Landau equation is calculated on VE (Fig. 14). The result of executing this code portion is the same result when lines 10–13 in Fig. 7 are executed.

```
all: vefortran fortran
vefortran: a fortran file name executed on VE
    nfort -c -o libvefortran.o $^ /opt/nec/ve/bin/mk_veorun_static -o $@
        libvefortran.o
fortran: a C file name executed on VH
    gcc -o $@ $^ -I/opt/nec/ve/veos/include -L/opt/nec/ve/veos/lib64 -Wl,-
        rpath=/opt/nec/ve/veos/lib64 -lveo
```

Figure 12. Makefile in the VEO execution model

```
1 int main(){
2     ... configuration for using VEO ...
3     for(nt=ntmin; nt<=ntmax; nt=nt+ndt){
4         veo_args_set_stack(argp,VEO_INTENT_OUT,0,(char *)&w,sizeof(w));
5         veo_args_set_stack(argp,VEO_INTENT_IN,1,(char *)&nt,sizeof(nt));
6         id = veo_call_async_by_name(ctx,handle,"veo_",argp);
7         veo_call_wait_result(ctx,id,&retval);
8         veo_args_clear(argp);
9         ... outputting the results ...
10    }
11 }
```

Figure 13. A portion of the FICT code on VH in the VEO execution model

2.3. Approach Summary

We have applied seven implementations of the FICT based on the above approach by focusing on the improvement of the vectorization ratio and the efficiency improvement in I/O operations. Table 3 shows what tuning efforts are done in each implementation. OR indicates the original code of the FICT.

```

1      SUBROUTINE veo(w,nt)
2      ... etc. ...
3      DO nh=nhmin,nhmax,ndh
4          ... computing the Ginzburg-Landau equation ...
5          DO nx=1,N
6              x=dx*0.5d0*(dble(nx)+dble(nx-1))
7              ...etc. ...
8              w(1,nx,nh)=x
9              ... etc. ...
10             w(63,nx,nh)=DABS(g*(-k4*uv*dt))
11         END DO
12     END DO
13 END SUBROUTINE veo

```

Figure 14. A portion of the FICT code on VE in the VEO execution model**Table 3.** The FICT codes we have implemented

Execution model	ID	Vectorization improvement (Section 2.1)	I/O improvement (Section 2.2)
Standard	OR	no	no
	A1	yes	no
VH Call	B1	yes	no
	B2	no	yes
	B3	yes	yes
VEO	C1	yes	no
	C2	no	yes
	C3	yes	yes

3. Evaluation

As summarized in Section 2.3, we have developed seven implementations of the FICT based on three execution models suggested by the SX-Aurora TSUBASA by focusing on the improvement of the vectorization ratio and the efficiency of the I/O operations. In this section, we compare these three execution models in terms of performance and ease of programming. In all evaluations, we used a single core of processors because the maximum loop length in the FICT, 240, was smaller than the vector length of SX-Aurora TSUBASA, 256.

3.1. Experimental Environments

For the evaluation purpose, the SX-Aurora TSUBASA system the detail specification of which is shown in Tab. 4 and Tab. 5 was used. “Vector length” is the number of elements that VE can perform per instruction. The system is composed of 2 Intel Xeon Gold 6126 [15] with 96 GB memory as VH and SX-Aurora TSUBASA processor Type 10B as 8 VEs. Figure 2 illustrates the inside architecture of the system (A300-8). The Intel processors as VH and VEs are connected with PCI (Gen 3)16. In this experiment, a pair of VH and VE were used.

Table 4. Spec. of VE

VE type	Type 10B
Vector length	256
Number of cores	8
Frequency	1.4 GHz
Performance / core (DP)	268.8 GFlops
Performance / core (SP)	537.6 GFlops
LLC capacity	16 MB
Memory bandwidth	1.2 TB/s
Memory capacity	48 GB

Table 5. Spec. of VH

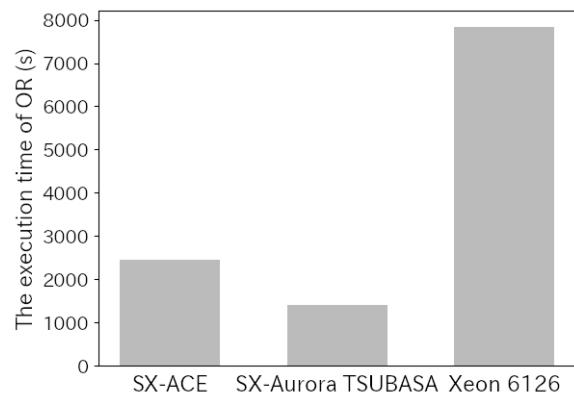
Processor	Intel Xeon Gold 6126 × 2 socket
Number of cores	12 × 2
Frequency	2.6 GHz
Performance / core (DP)	83.2 GFlops
Performance / core (SP)	166.4 GFlops
LLC capacity	19.25 MB × 2
Memory bandwidth	128 GB/s × 2
Memory capacity	96 GB × 2

3.2. Performance of the SX-Aurora TSUBASA

First, we measured the execution time of the original FICT code (OR) on the SX-ACE system, the SX-Aurora TSUBASA system and a compute node of the OCTOPUS system (the Xeon 6126 system) [1], which has the same processors as the VH of the SX-Aurora TSUBASA system. The specification of the SX-ACE is shown in Tab. 6. Figure 15 shows the execution time of the FICT on the SX-ACE system, the SX-Aurora TSUBASA system and the Xeon 6126 system. The graph indicates that the execution time of OR on the SX-Aurora TSUBASA system was 1413.81 seconds, on the SX-ACE system the execution time was 2451.61 seconds and on the Xeon 6126 system was 7828.43 seconds. According to this result, the SX-Aurora TSUBASA system delivers 1.7 times higher performance to the FICT than the SX-ACE system. Moreover, this result indicates that SX-Aurora TSUBASA is more suited for the FICT than scalar processors. Although the performance per core of SX-Aurora TSUBASA is almost four times that of SX-ACE, the execution time of the FICT is reduced by only about 40 % on the SX-Aurora TSUBASA system. As stated in Section 1.3.1, since SYS. B/F of SX-Aurora TSUBASA is smaller than REQ. B/F of the FICT, the performance of SX-Aurora TSUBASA does not achieve the theoretical performance of SX-Aurora TSUBASA.

Table 6. Spec. of the SX-ACE system

Vector length	256
Number of cores	4
Performance / core (DP)	64 GFlops
Memory bandwidth	256 GB/s
Memory capacity	64 GB

**Figure 15.** The execution time of OR on the SX-ACE system, the SX-Aurora TSUBASA system and the Xeon 6126 system

3.3. Effect of Improving the Vectorization Ratio

Next, we observed and compared how the vectorization ratio and the execution time were improved with the improvement of the vectorization ratio. Figures 16 and 17 show the vectorization ratio and the execution time of OR and A1. The vectorization ratio of the FICT increased from 94.4 % to 99.4 %. The execution time of A1 was reduced to 447.02 seconds. This result indicates that the improvement of the vectorization ratio reduced the execution time of A1 by 68.4 % compared to the original code of the FICT (OR).

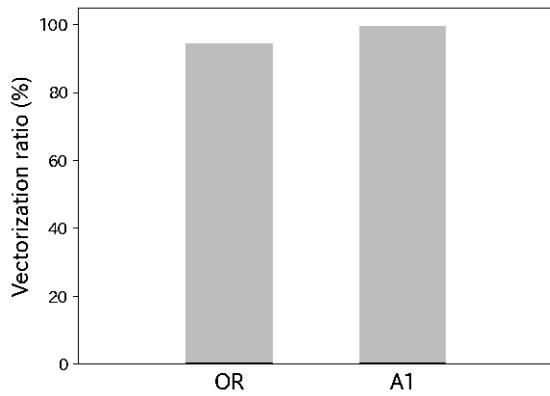
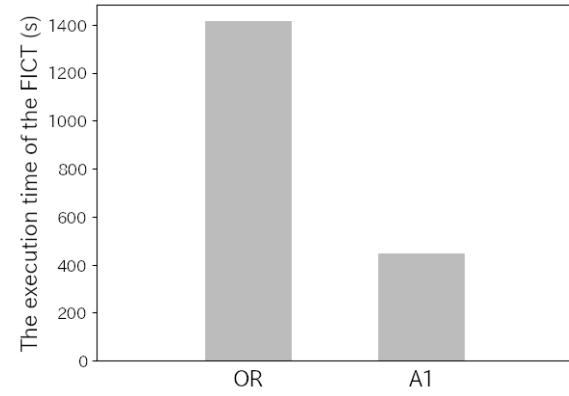


Figure 16. The vectorization ratio of the FICT **Figure 17.** The execution time with improvement of the vectorization ratio



3.4. Effect of the Efficiency Improvement of Data Transfer

Next, we observed and compared how the number of data transfers was reduced and the execution time was improved with the approach summarized in Section 2.2. Table 7 shows the number of data transfers of OR, B2 and C2. The number of data transfers of OR was 24,480 and that of B2 and C2 was 2. Figure 18 shows the execution time of OR, B2 and C2. The execution time of B2 was 1372.26 seconds and that of C2 was 1320.19 seconds. This result indicates that reduction of data transfers in the VH Call and VEO execution models slightly reduced the execution time of the FICT compared to that of the original FICT on the standard execution model.

Table 7. The number of data transfers

ID	The number of data transfers
OR	24480
B2	2
C2	2

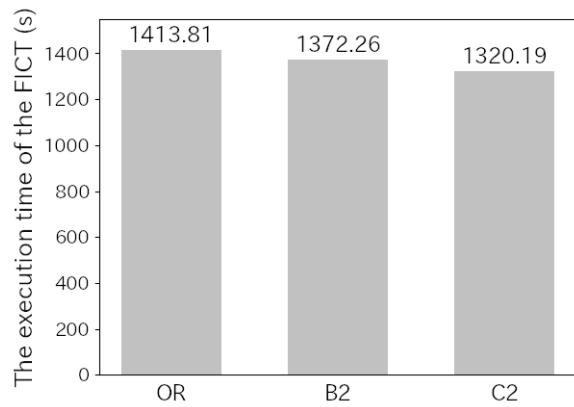


Figure 18. Execution time with reduction of data transfers

3.5. Performance Acceleration of the Three Execution Models

Next, we measured the execution time of OR, A1, B3 and C3 to investigate the three implementations based on the three execution models by focusing on the improvement of vectorization and I/O operational efficiency. Figure 19 shows the result of this measurement. The execution time of A1 was reduced to 447.02 seconds. On the other hand, the execution time of B3 was 446.07 seconds and that of C3 was 443.29 seconds. This result indicates that the implementations of the FICT based on the VH Call and VEO execution models were slightly superior in performance to that based on the standard execution model as the result of our tuning effort that focused on the improvement of the vectorization ratio and the I/O operational efficiency.

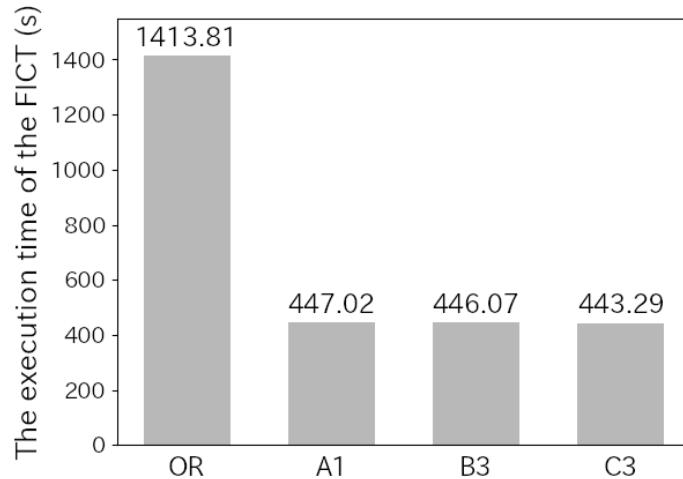


Figure 19. The execution time of the FICT with acceleration approaches

3.6. Code Modification for Acceleration

We investigated how many lines must be modified to implement seven FICT codes in Tab. 8. The number of lines modified to implement A1, B1 and C1 is the same. In the VH Call and VEO execution models, the number of lines modified to implement B2 and C2 was approximately 4.2 times larger than A1. To implement C3, we had to rewrite the code written in Fortran to be executed on VH to C.

Table 8. Modification for the FICT code

Execution model	ID	The number of modified lines	Programming language modification
Standard	OR	0	no
	A1	42	no
VH Call	B1	42	no
	B2	175	no
	B3	217	no
VEO	C1	42	no
	C2	180	no
	C3	222	yes

Conclusion

In this paper, we reported the experience of accelerating the FICT using the newly emerged SX-Aurora TSUBASA. Specifically, we applied seven implementations focusing on the three execution models which the SX-Aurora TSUBASA suggested as well as the improvement of vectorization and I/O operations as tuning methods. Hopefully, this will alleviate the FICT users' concerns and questions about how the three execution models are to be selected. Our evaluation showed that the execution time of the implementation conforming to the VEO execution model was 0.83 % shorter in comparison with the implementation conforming to the standard execution model (OS Offload), which we suggested as the standard execution model. On the other hand, we also showed that the VEO and VH Call execution models require us to write or modify the code by recognizing the structure of the SX-Aurora TSUBASA system, while the standard execution model allows us to easily accelerate the FICT running on the SX-ACE system, which focuses on the improvement of the vectorization and I/O operations. In the case of the FICT, the VEO execution model is the best if researchers respect performance. However, the standard execution model would be the best if researchers respect readability and maintainability of the FICT.

Acknowledgements

We thank the Cybermedia Center (CMC) of Osaka University which provided the SX-ACE system and the SX-Aurora TSUBASA system. This research is partly supported by the Joint Usage and Research (No. jh200032) of JHPCN. Also, we thank the technical team of the NEC at CMC for useful advice and direction. We are grateful to Kouki Otsuka for valuable discussions on the theoretical aspects of chiral superconductivity. This work is supported by the JSPS Core-to-core program No. JPJSCCA20170002.

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. Cybermedia Center of Osaka University: OCTOPUS. <http://www.hpc.cmc.osaka-u.ac.jp/octopus/>, accessed: 2021-08-08
2. Egawa, R., Fujimoto, S., Yamashita, T., *et al.*: Exploiting the Potentials of the Second Generation SX-Aurora TSUBASA. In: 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). pp. 39–49. IEEE (2020). <https://doi.org/10.1109/PMBS51919.2020.00010>
3. Egawa, R., Komatsu, K., Momose, S., *et al.*: Potential of a modern vector supercomputer for practical applications: performance evaluation of SX-ACE. The Journal of Supercomputing 73(9), 3948–3976 (2017). <https://doi.org/10.1007/s11227-017-1993-y>
4. Ginzburg, V.L., Landau, L.D.: On the theory of superconductivity. Journal of Experimental and Theoretical Physics 20, 1064–1082 (1950), English translation in: L.D. Landau:

- Collected papers. Oxford: Pergamon Press, 546–568 (1965). <https://doi.org/10.1016/B978-0-08-010586-4.50078-X>
5. Grinenko, V., Ghosh, S., Sarkar, R., *et al.*: Split superconducting and time-reversal symmetry-breaking transitions in Sr_2RuO_4 under stress. *Nature Physics* 17(6), 748–754 (2021). <https://doi.org/10.1038/s41567-021-01182-7>
 6. Kaneyasu, H., Enokida, Y., Nomura, T., *et al.*: Features of Chirality Generated by Paramagnetic Coupling to Magnetic Fields in the 3 K-Phase of Sr_2RuO_4 . In: JPS Conference Proceedings 30, 011039-1-6 (2020). <https://doi.org/10.7566/JPSCP.30.011039>
 7. Kaneyasu, H., Enokida, Y., Nomura, T., *et al.*: Properties of the H-T phase diagram of the 3-K phase in eutectic Sr_2RuO_4 -Ru: Evidence for chiral superconductivity. *Physical Review B* 100(21), 214501 (2019). <https://doi.org/10.1103/PhysRevB.100.214501>
 8. Komatsu, K., Momose, S., Isobe, Y., *et al.*: Performance Evaluation of a Vector Supercomputer SX-Aurora TSUBASA. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 685–696. IEEE / ACM (2018). <https://doi.org/10.1109/SC.2018.00057>
 9. Luke, G.M., Fudamoto, Y., Kojima, K.M., *et al.*: Time-reversal symmetry-breaking superconductivity in Sr_2RuO_4 . *Nature* 394(6693), 558–561 (1998). <https://doi.org/10.1038/29038>
 10. Maeno, Y., Ando, T., Mori, Y., Ohmichi, E., *et al.*: Enhancement of Superconductivity of Sr_2RuO_4 to 3 K by Embedded Metallic Microdomains. *Physical Review Letters* 81(17), 3765–3768 (1998). <https://doi.org/10.1103/PhysRevLett.81.3765>
 11. Maeno, Y., Hashimoto, H., Yoshida, K., *et al.*: Superconductivity in a layered perovskite without copper. *Nature* 372(6506), 532–534 (1994). <https://doi.org/10.1038/372532a0>
 12. Matsumoto, M., Sigrist, M.: Quasiparticle States near the Surface and the Domain Wall in a $p_x \pm ip_y$ -Wave Superconductor. *Journal of the Physical Society of Japan* 68, 994–1007 (1999). <https://doi.org/10.1143/JPSJ.68.994>
 13. NEC: PROGINF/FTRACE Users Guide. https://www.hpc.nec/documents/sdk/pdfs/g2at03e-PROGINF_FTRACE_User_Guide_en.pdf, accessed: 2020-12-07
 14. Sigrist, M., Ueda, K.: Phenomenological theory of unconventional superconductivity. *Reviews of Modern Physics* 63(2), 239–311 (1991). <https://doi.org/10.1103/RevModPhys.63.239>
 15. 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). <https://doi.org/10.14529/jsfi190201>
 16. Xia, J., Maeno, Y., Beyersdorf, P.T., *et al.*: High Resolution Polar Kerr Effect Measurements of Sr_2RuO_4 : Evidence for Broken Time-Reversal Symmetry in the Superconducting State. *Physical Review Letters* 97(16), 167002 (2006). <https://doi.org/10.1103/PhysRevLett.97.167002>

Evaluating the Performance of OpenMP Offloading on the NEC SX-Aurora TSUBASA Vector Engine

Tim Cramer¹ , **Boris Kosmynin¹** , **Simon Moll²**, **Manoel Römmер¹**,
Erich Focht² , **Matthias S. Müller¹** 

© The Authors 2021. This paper is published with open access at SuperFri.org

The NEC SX-Aurora TSUBASA vector engine (VE) follows the tradition of long vector processors for high-performance computing (HPC). The technology combines the vector computing capabilities with the popularity of standard x86 architecture by integrating it as an accelerator. To decrease the burden of code porting for different accelerator types, the OpenMP specification is designed to be single parallel programming model for all of them. Besides the availability of compiler and runtime implementations, the functionality as well as the performance is important for the usability and acceptance of this paradigm. In this work, we present LLVM-based solutions for OpenMP target device offloading from the host to the vector engine and vice versa (reverse offloading). Therefore, we use our source-to-source transformation tool sotoc as well as the native LLVM-VE code path. We assess the functionality and present the first performance numbers of real-world HPC kernels. We discuss the advantages and disadvantage of the different approaches and show that our implementation is competitive to other GPU OpenMP runtime implementations. Our work gives scientific programmers new opportunities and flexibilities for the development of scalable OpenMP offloading applications for SX-Aurora TSUBASA.

Keywords: HPC, OpenMP, offloading, reverse offloading, vector computing, performance.

Introduction

Nowadays, computer simulations form together with theory and experiment the third pillar of scientific research. The resulting on-growing demand for large compute capabilities led to wide use and acceptance of accelerator technologies. The NEC SX-Aurora TSUBASA vector engine (VE) is one promising solution for the acceleration of compute-intensive simulation codes. The technology integrates long vector computing into a x86 environment as a PCIe card.

However, in order to make this compute power accessible for scientific applications, support for a great range of scalable parallel programming paradigms is required. OpenMP [24] is one of these powerful solutions, which is in addition very convenient due to the compiler directive approach, which allows incremental application porting. The NEC compiler is a specialized cross-compiler for the VE, it can only produce VE code that runs natively on the device and lacks support for x86_64 compilation. It comes with native OpenMP 4.5 support for the VE but lacks support for OpenMP target device offloading. Since not all parts of an application might deliver a good performance on a vector engine (e.g. file IO, data initialization), a programmer might want to offload only the compute-intensive and vectorizable parts of the code, which is in general supported by the OpenMP target device constructs. In order to enable this functionality we presented a first implementation in [12]. Although this approach is functional and shows a good performance (as we will show in this paper), there are some disadvantages. Thus, we will also present a LLVM-VE path which uses a native VE backend which is integrated into LLVM.

Even in large real-world HPC applications there are typically only a couple of hotspots which consume most of the time. When comparing runtime profiles on different architectures, a performance engineer might identify different hotspots for the same application and data set. On the

¹IT Center, RWTH Aachen University, Germany

²NEC Corporation, Stuttgart, Germany

VE this is especially true for those code regions or functions which do not vectorize. This means that a single not-well suited function can limit the overall performance of an application significantly. To avoid such cases and give the programmer the opportunity to run poorly vectorized functions on the x86 vector host (VH), we also present the possibility of reverse offloading. This means we start the entire program on the VE and only parts are offloaded back to the VH. All our implementations and documentations are open source and freely available [4].

To summarize, in this paper we describe the following contributions:

- We present a native LLVM-VE path to enable OpenMP target device offloading to the VE.
- We present the first OpenMP reverse offloading from the VE to the VH.
- We discuss the advantages and disadvantages of the different approaches.
- We evaluate the functionality of all approaches.
- We assess the performance of OpenMP offloading kernels to the VE with relevant benchmark suites and compare them to current GPU implementations.

The paper is organized as follows: the remaining section gives a brief overview on the SX-Aurora TSUBASA vector engine and the basic concepts of OpenMP target device offloading. Section 1 explains the basic concepts of the different approaches. These concepts are evaluated in detail in Section 2 in terms of functionality and performance. Section 3 gives an overview on related work, before we conclude our work.

SX-Aurora TSUBASA Vector Engine

The Vector Engine (VE) was released in 2018 in the form factor of a PCIe card containing a processor with on-chip memory built of six HBM2 stacks, four or eight layers high, with a total of 24 or 48GB RAM and a total memory bandwidth of either 750 GB/s (24 GB model) or 1.2 TB/s [29]. The second generation VE20 released in 2020 has an increased memory bandwidth of 1.53 TB/s. The VEs re-implement the long vector processors ISA of the NEC SX architecture that combines a 2048 bit SIMD unit with an 8 cycle deep pipeline resulting in vector registers with a length of $256 * 64$ bits = 16384 bits. Unlike classical SIMD units the VE ISA contains a vector length register that controls how many elements of a vector register will be processed in a vector instruction. The vector processor has either eight or ten cores with a scalar processing unit (SPU) and a vector processing unit (VPU) featuring 64 architectural vector registers, three FMA, two integer ALU and a SQRT/DIV vector pipeline. The cores share a common 16 MB four-way skewed associative last level cache that acts as a vector cache, while each core has private L1 and L2 caches dedicated for their SPUs. The clock frequency of the VE is either 1400 or 1600 MHz resulting in the peak performance of 3 TFLOPS in double precision and 6 TFLOPS in single precision with ten cores. For this paper we used the vector engine models VE10B.

The VEs are hosted in x86_64 servers fitting up to eight accelerator cards. They can be linked to large clusters by EDR Infiniband interconnects which are used by the VEs in a PeerDirect manner. The vector host (VH) runs Linux while the VE Operating System (VEOS) functionality is offloaded to the host and implemented as a user space daemon. The core programming models are: (1) native VE programs written in C, C++, Fortran and running entirely on the VEs while offloading their system calls to the VH; (2) native VE programs executing parts of the program on the VH through reverse offloading; (3) host programs offloading kernels to the VE; (4) hybrid MPI programs running processes on both VEs and VH. Fine grained parallel execution on the VE is achieved through vectorization, at coarse, core level programmers can use OpenMP, pthreads or multiple MPI processes.

OpenMP Target Device Offloading

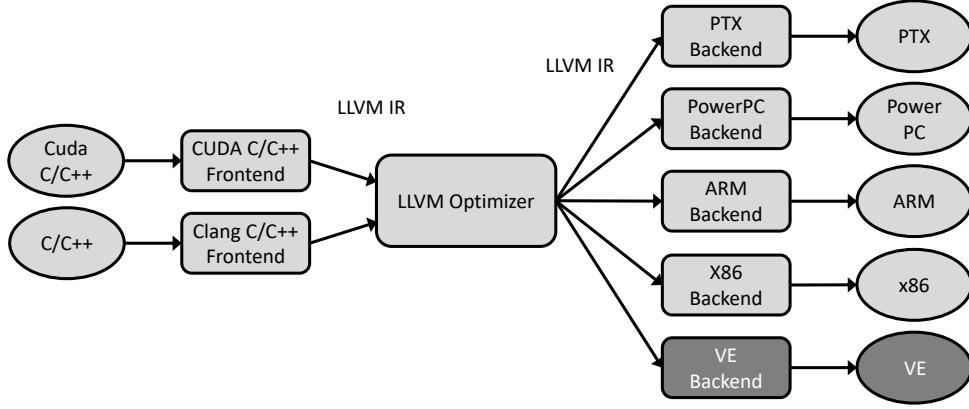
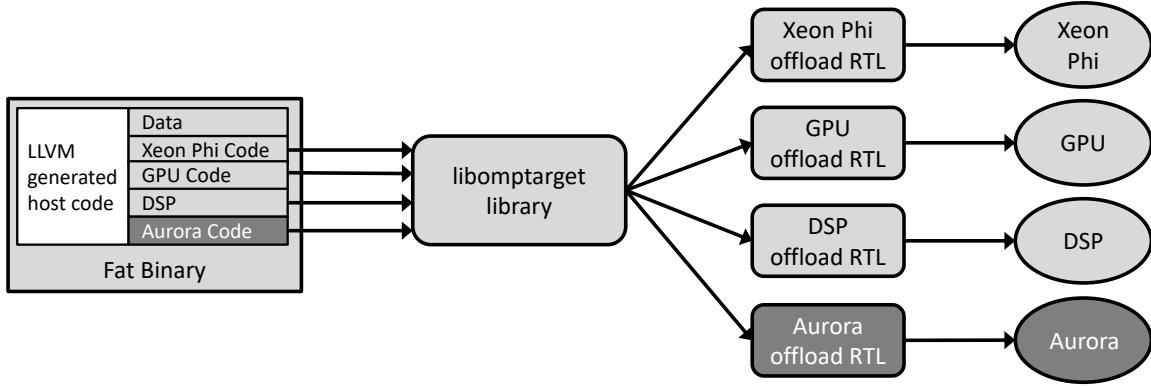
OpenMP [24] is known as de-facto standard for shared memory parallel programming. In addition, target device offloading for compute-intense code parts to accelerators is possible since version 4.0. In order to be generic, the OpenMP specification does not define the concrete hardware architecture for such target devices. Thus, a target device may be a GPU, a DSP, an Intel Xeon Phi, a x86 system, a vector engine or a logical execution engine running on the same physical processor. Although threads can not migrate between devices, a target device may or may not share the hardware resources like the memory or cores. An OpenMP implementation requires both compiler and runtime library support.

Offloading regions are explicitly expressed by `target` constructs in the code (i.e., user directives). Since the target device may have a different instruction set architecture (ISA), all static variables used and all functions called within the scope of target region have to be declared by using `declare target` directives. Within a target device region, all other OpenMP directives are allowed (e.g., for parallel regions). In order to address the specific hardware architecture hierarchy of typical target devices like GPUs, OpenMP offers the `teams` construct, which creates multiple teams of threads. The `distribute` construct allows scheduling a set of teams for execution of a loop. While the threads within a thread team can be synchronized by `barrier` constructs, no such primitive exists to synchronize multiple thread teams (similar to CUDA threadblocks in NVIDIA GPUs). Since the performance for target devices like the SX-Aurora TSUBASA is driven by vector instructions, OpenMP offers the `simd` construct in order to signal the compiler that the corresponding loop is data parallel and can be vectorized. The iterations of two or more nested loops can be collapsed into one larger logical iteration space with the `collapse` clause, which might increase the performance due to a bigger vector length. For convenience reasons, OpenMP defines a set of combined constructs as shortcuts for specifying one construct nested inside another one (e.g. the `target teams distribute parallel for simd` directive). A standard-compliant OpenMP implementation has to implement all these combined constructs.

To ensure data consistency between target host and the target device data environment, the `map` clause can be added to different target-related constructs. Furthermore, the `target data` construct maps variables to the data environment without executing any user code. Data transfers to the target device memory and allocations in the target device memory are issued according to a reference count mechanism. Corresponding map-type-modifiers or constructs like `target update` can make those data operations explicit. The `target enter data` and `target exit data` constructs explicitly map variables to the target device data environment without using a data region (i.e. a scope).

1. OpenMP Target Device Offloading Designs

In [12] we presented a first solution realizing OpenMP Target Device Offloading from a vector host (VH) to the SX-Aurora TSUBASA vector engine (VE) by leveraging the LLVM infrastructure. One of the main concepts of this approach is to make use of the Clang compiler frontend of the base language (e.g., C/C++) and the x86 backend in order to generate the code parts for the VH. Furthermore, the frontend can pass the LLVM Intermediate Representation (LLVM IR) to the VE backend in order to generate the code parts for the VE (see Fig. 1). In general, the frontend parses the code, generates LLVM IR, optimizes the IR and passes the code to a corresponding backend. This backend generates the code for the target platform. For OpenMP Target Device

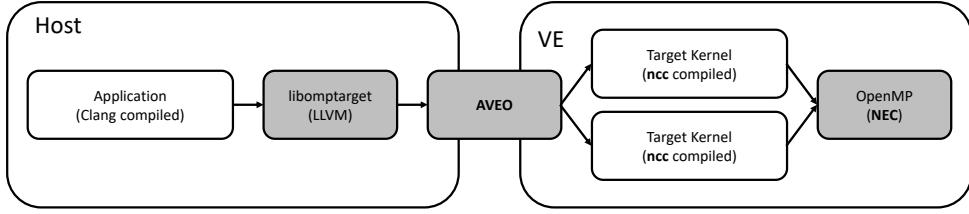
**Figure 1.** High level view of the LLVM toolchain**Figure 2.** The LLVM Offloading Infrastructure, based on [7, 12]

Offloading this process is more complex, because the driver needs to invoke different tool chains for the same set of input files [7]. Parts of the codes will be processed multiple times in order to generate the code for all OpenMP target devices and the host device. Furthermore, corresponding code for all dependencies to required data structures, types and functions will be generated for each device. All these partial outputs of each tool chain will be integrated as separated code paths into the same fat binary (see Fig. 2). In addition to the compiler support, a runtime library is required in order to execute the integrated code on the target device. In LLVM this is handled by the libomptarget library, which selects at runtime a target device for the execution of the offloaded code. We developed a corresponding plugin based on the NEC VE Offloading (VEO [15]) interface. However, our current implementation benefits from the new AVEO [16] implementation, which shows a better performance compared to our original library.

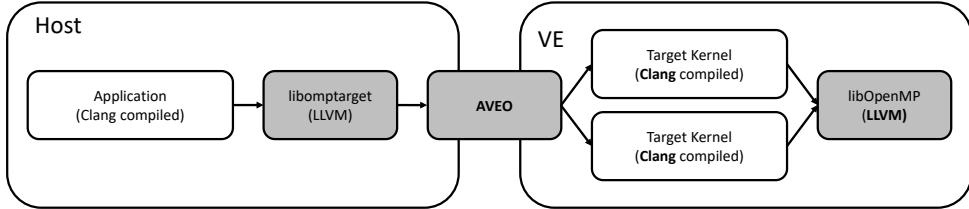
Since the development of compiler backends is a complex task, no LLVM backend for VEs was available at first. In order to enable OpenMP Target Device Offloading early our first approach was a source-to-source transformation technique (see Section 1.1). Meanwhile, a LLVM backend for VE is available, which enables also OpenMP target device offloading on a native path for the first time (see Section 1.2). Furthermore, we discuss an approach for reverse offloading from VE to a x86 VH (see Section 1.3).

1.1. Source-To-Source Transformation with sotoc

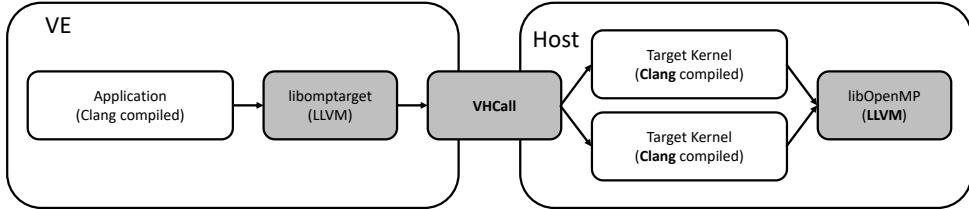
For target devices which do not have an LLVM backend available, an additional compiler is required for the code generation. In our source-to-source approach [12] we outline all OpenMP



(a) VH to VE offloading with sotoc



(b) LLVM-VE code path for VH to VE offloading



(c) LLVM-VE code path for VE to VH offloading

Figure 3. Comparison of the different offloading opportunities

target regions as well as all functions, types and global variables required on the target device by using our Clang-based tool *sotoc*. An example for the function outlining is shown in Fig. 4. These outlined code fragments are passed to the external compiler (e.g., the NEC compiler) such that only the required parts are compiled for the target device. Figure 3a provides an overview of an application that was compiled with *sotoc* offloading. The target kernels are extracted with *sotoc* from the original source code and compiled with the official NEC C compiler (*ncc*). The VE code uses the proprietary OpenMP implementation of NEC that comes with the compiler. The x86 host application itself is compiled with Clang.

One advantage of this approach is that it is very generic and completely vendor-independent. In general, it can be applied for any target device type which has a C compiler and a native OpenMP runtime available. Thus, the existing infrastructure is leveraged and a corresponding LLVM toolchain with a LLVM backend is not required. Furthermore, this approach benefits from the optimization capabilities of the existing compiler and delivers good code performance.

A disadvantage of this approach is that it does not fit well into the LLVM workflow. Since our source-to-source tool relies on the internal abstract syntax tree representation (AST) which is not fully exposed to external tooling via a stable interface, it is error-prone and might break with any LLVM internal update. In this context, we encountered one limitation in our code transformation. C11 allows so-called anonymous enums and structs, which means that we can never refer to these anywhere else in the code by their type name. However, we need to pass data to the outlined function as shown in Fig. 5, for which we need the data's type name. Due to the Clang's AST

```
#pragma omp declare target
int n = 10240;
#pragma omp end declare target
void saxpy(){
    float a = 42.0f; float b = 23.0f; float *x, *y;
    // Allocate and init x, y
    // ...
#pragma omp target map(to:x[0:n], a) map(tofrom:y[0:n])
#pragma omp parallel for
for (int i = 0; i < n; ++i){
    y[i] = a*x[i] + y[i];
}
}
```



```
int n = 10240;
void __omp_ofld_b73b_saxpy_14(int n, float * y, float * __sotoc_var_a, float * x) {
    float a = *__sotoc_var_a;
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    y[i] = a * x[i] + y[i];
}
__sotoc_var_a = a;
}
```

Figure 4. Basic function outlining with our source code transformation technique [12]

```
void foo() {
    enum { VAL1 = 1, VAL2, VAL3, VAL4} scalar_enum = VAL1;
#pragma omp target map(tofrom: scalar_enum)
{
    scalar_enum = VAL4;
}
printf("%d",scalar_enum);
}
```



```
enum { VAL1 = 1, VAL2, VAL3, VAL4} scalar_enum = VAL1;
void __omp_ofld_b73b_foo_15(<ANOM_ENUM_TYPE> * __sotoc_var_scalar_enum)
{
    enum <ANOM_ENUM_TYPE> scalar_enum = __sotoc_var_scalar_enum;
    scalar_enum = VAL4;
    __sotoc_var_scalar_enum = scalar_enum;
}
```

Figure 5. Limitation in our source-to-source approach: Anonymous enum

representation, the fact that we have to hand over the data as pointer and that we can not name `<ANOM_ENUM_TYPE>`, we do not have a solution to generate valid C-code in this case. Although, a programmer can work around this limitation by naming the struct or enum, not all codes will compile out of the box. Another minor disadvantage is that the compilation time might be slightly slower, because we add the source-to-source transformation and the compiler analysis in two different compiler frontends. Due to these downsides, LLVM-VE code generation is preferable in the long term, when the performance is as good as for the source-to-source approach.

1.2. LLVM-VE Code Generation

The standard way to implement OpenMP target device offloading with LLVM is to use LLVM IR and a regular LLVM backend for target code generation. This is what we call the LLVM-VE code path in contrast to the source-to-source sotoc path described in Section 1.1. The LLVM-VE path also relies on the open source OpenMP runtime of LLVM for constructs running on the target, such as *parallel for*. Both code paths use the VE plugin of libomptarget.

Figure 3b shows an overview of an OpenMP target application that is compiled with the LLVM-VE native code path. All code is generated by LLVM, and the LLVM OpenMP runtime is running on the device. Only AVEO, the actual offloading library of NEC, is not part of the LLVM stack.

The LLVM-VE code path has two core advantages: First, it relies on LLVM IR and so any frontend of LLVM, for example the upcoming Flang [2] Fortran frontend, can immediately use it for target offloading. Second, the LLVM OpenMP runtime is used by many different targets. This means the implementation of the OpenMP constructs have a lot of exposure to testing, which makes the implementation very compliant. We show this quantitatively on the OMPVV test suite in Section 2.1.

The main disadvantage of LLVM-VE is that LLVM is not tuned for the VE. This shows the comparison to the official NEC compilers for VE, which are specifically tailored for this architecture.

1.3. Reverse Offloading

The VE ecosystem provides offloading from the VE to the host machine with the VHCall [3] reverse offloading library. We provide this as a standard LLVM offloading path with another libomptarget plugin for reverse offloading. This setup is contrary to the usual offloading paradigm, in reverse offloading the application runs on the accelerator and kernels are offloaded to the host machine. However, regarding OpenMP this is standard-compliant, because the specification does not define the concrete heterogeneous architecture. In this case the host machine becomes the *target device* with the accelerator being the *host device*.

Figure 3c gives an overview of the OpenMP target reverse offloading structure. All code is generated by LLVM, and the LLVM OpenMP runtime is running on the device. Only VHcall, the actual offloading library is not part of the LLVM stack. In essence, except for its use of VHCall instead of AVEO, this is the same as in Fig. 3b with the use of VH and VE swapped.

2. Evaluation

In this section we evaluate the performance of the SX-Aurora TSUBASA and validate the completeness and correctness of the source-to-source, the LLVM-VE and the reverse offloading approaches. We perform comparative measurements on three systems. All VE measurements have been executed on a NEC SX-Aurora TSUBASA Type 10B with 8 cores running at 1.4 GHz and a memory bandwidth of 1.2 TB/s. Those measurements are hereafter referred to as SX. The accompanying base system consists of two Intel Xeon Silver 4108 CPUs at 1.8 GHz with 16 cores in total. Measurements referring to x86 were performed on the same system, with the exception of runs including a GPGPUs. For those measurements, referred to as V100, a dual-socket system with two Intel Xeon Platinum 8160 CPUs at 2.1 GHz with 48 cores in total, and two Nvidia Volta V100 GPUs was used. One of those V100s is actually used for the benchmarks.

2.1. OMP Validation & Verification Suite

The OMP Validation & Verification Suite [13, 14] is used to evaluate the completeness and OpenMP specification adherence of the source-to-source, the LLVM-VE and reverse offloading approach for OpenMP code offloading. The suite was developed by researchers at the University of Delaware and the Oak Ridge National Laboratory, and published in 2018 as part of the Exascale project. Since its publication it continues to be well maintained and to receive additions and fixes [5]. Even though the suite is comprised of C, C++ and Fortran tests, only C and C++ tests were used for the following evaluation, as Fortran code is not supported by either implementation.

Table 1 shows the results of the evaluation, as a sum total of tests passed and tests failed with errors at compile and runtime, separated for the C and C++ tests. We can see that both x86→SX paths behave comparably. The LLVM native path, however, has no compile errors, as this path allows for syntax-agnostic transformation of the code. As sotoc does not support C++ code, none of the tests compile. The native LLVM-VE path, however, does support C++ code compilation and runs all but one C++ tests. The *test_enter_data_classes_inheritance* test fails. Clang warns at compile time that incorrect mapping may occur in this test because the mapped object is a class with a non-trivial copy constructor. This SOLLVE test may be unsound.

In the following we discuss the main roots of the failures. Due to the limited scope of this publication, we will focus on all runs pertaining to the SX-Aurora. For the source-to-source implementation we can ascribe all errors to one of three main reasons, not counting the lack of C++ support. The first one, which covers all compile-time errors, are the so called anonymous structs. Those are unnamed structs, and our implementation is unable to properly process them, as we describe in Section 1.1 and one can see in Fig. 5. Since we do not have a solution for this issue, it will stay as a limitation for the source-to-source approach. The second reason, responsible for most of the runtime errors, is memory miss-management. This fault occurs mainly when dealing with multiple devices, and/or with asynchronous code execution. However, multi-device and asynchronous execution support is available and does generally work properly. This problem is also present in the LLVM-VE native path, which suggest an issue with the OpenMP plugin, rather than any specific implementation. The third reason is limited construct or clause support in OpenMP 4.5. As per the method described in Section 1.1, we split combined target constructs up and discard the target. This method leaves, in very rare cases, a vital construct or clause unable to be used any further, as some constructs are not allowed to be used in an stand-alone fashion. Furthermore, clauses pertaining to those constructs also have to be discarded.

2.2. EPCC Syncbench Microbenchmark

All parallel programming paradigms introduce some additional overhead in terms of compute cycles (e.g., for the communication, synchronization or data management). In order to achieve good performance and scalable applications, it is important to reduce this overhead to a minimum. The overhead introduced with OpenMP constructs does not rely on the paradigm itself, but more on the quality of the compiler and runtime implementation and the optimization for the target architecture. In order to assess the state of the different runtime implementations of selected key OpenMP constructs, we used the `syncbench` benchmark, which is part of the EPCC OpenMP Microbenchmark Suite [9] and measures those constructs requiring synchronization. We focused on the `syncbench`, because these would show limited scalability and include the most common used constructs. In addition to the benchmarks in the original suite we used an extension as presented

Table 1. Test summary for the OMP Validation and Verification Suite (total of 109 C and 14 C++ tests), where “ $A \rightarrow B$ ” means offloaded from target host A to target device B . The source-to-source path is marked in brackets. All other tests are using the native LLVM / LLVM-VE path

		x86→SX (sotoc)	x86→SX	x86→V100	x86→x86	SX→x86
C	Passed	91	98	105	97	101
	Compile Error	7	0	0	0	0
	Runtime Error	11	11	4	12	8
C++	Passed	0	13	14	13	12
	Compile Error	14	0	0	1	0
	Runtime Error	0	1	0	0	2

```

start = omp_get_wtime();
#pragma omp target
int j;
for (j=0; j<innerreps; j++){
    delay(delaylength);
}
t_ref = (omp_get_wtime() - start);

```

(a) Reference time t_{ref} of `innerreps` executions with a `delay`

```

start = omp_get_wtime();

int j;
for (j=0; j<innerreps; j++){
    #pragma omp target
    {
        delay(delaylength);
    }
}
t_ofld = (omp_get_wtime() - start);

```

(b) Offloading time t_{ofld} of `innerreps` target regions with a `delay`

Figure 6. EPCC-like kernels to determine the overhead of a `target` construct

in [11] in order to assess the overhead of the `target` construct. Here, we applied the same procedure as it is done for the other constructs. Basically, we compare the measured reference time t_{ref} of an offloaded run including a `delay` function with the measured time t_{ofld} of multiple offloaded functions with the same delay (see Fig. 6). The overhead O of a target region is determined as

$$O = \frac{t_{ofld} - t_{ref}}{innerreps}. \quad (1)$$

Furthermore, we ported the benchmarks into an offload version in order to measure the overheads of the selected constructs nested into a target region. Here, the expectation is that the overhead between the offloaded and original version is not significant for the same OpenMP runtime implementation.

Table 2 shows the results, where we used the median of 20 repetitions. Depending on the target device, a different number of threads has been used: 8 threads on the SX-Aurora, 16 on the x86 system and the implementation default on the V100. Since the expected overhead increases with growing numbers of threads, we only make qualitative comparison between the different devices and implementations. However, this can be done best by using all available cores of the underlying hardware, because this is the most typical use case. As expected, we can see that the overheads of the NEC OpenMP runtime for a `parallel for` ($6.77\mu s$ vs. $7.08\mu s$), a `barrier` ($3.74\mu s$ vs. $3.79\mu s$) and a `reduction` ($7.01\mu s$ vs. $7.51\mu s$) are comparable when executing on the SX-Aurora with and without a target region (s. rows `SX (NEC)` and `x86→SX (NEC)`).

The same holds for the LLVM OpenMP runtime on x86 (*x86* vs. *x86*→*x86*) and the LLVM OpenMP runtime on SX-Aurora (*SX (LLVM)* vs. *x86*→*SX(LLVM)*), especially considering the order of magnitude (microseconds). This shows that the runtime implementations do not introduce additional overhead due to the nesting of OpenMP constructs into target regions.

Table 2. EPCC Syncbench and **target** construct overhead in μs . Columns without a “→” show the results of the original benchmarks without any target region. Columns with a “→” show the results for the modified version with constructs nested into a target region, where “*A* → *B*” means offloaded from target host *A* to target device *B*. The measured OpenMP runtime implementation is denoted in brackets

		target	parallel for	barrier	reduction
SX	(NEC)	-	6.77	3.74	7.01
SX	(LLVM)	-	724.4	309.8	608.5
SX→x86	(LLVM)	173.47	14.34	4.36	6.83
x86	(LLVM)	-	7.27	1.87	7.50
x86→x86	(LLVM)	96.45	7.97	2.47	8.94
x86→SX	(NEC)	163.34	7.08	3.79	7.51
x86→SX	(LLVM)	124.99	815.24	339.07	663.55
x86→V100	(LLVM ^a)	130.18	4242.64	2.35	34.73

Furthermore, we see that the NEC OpenMP runtime on SX-Aurora is very competitive, compared to the LLVM OpenMP runtime on x86. However, the overheads of the LLVM OpenMP runtime on the SX-Aurora are two orders of magnitudes higher than the overheads of the NEC OpenMP runtime. This clearly shows that the LLVM runtime was optimized for x86 architectures, but not for vector engines. The main reason here is that the LLVM OpenMP runtime internally uses fast user-space locking (**futex**). On the VE this is executed as a system call. Due to the fact that the VE does not run an operating system on the device, a call back to the VH has to be done, which is expensive. To fix this performance issue, one has to replace each **futex** by a mechanism which uses hardware synchronization registers instead. This limits the performance especially for small regions, because the overhead is constant for a given number of threads. Since the influence for bigger regions is smaller, the LLVM OpenMP runtime can still provide a good performance for many applications.

The comparison to the LLVM OpenMP runtime on a Nvidia V100 shows that the NEC runtime has comparable overheads for **barrier** constructs and **reduction** clauses. The overhead for a **parallel for** construct is about three orders of magnitudes higher ($\sim 4242\mu\text{s}$ vs. $\sim 7\mu\text{s}$) which limits the performance for OpenMP programs using this combined construct. In contrast to the LLVM runtime on the SX-Aurora, the LLVM runtime on V100 is not just a cross-compiled version of the original runtime, but a special implementation which is integrated into LLVM. The comparison of the overhead for the **target** constructs shows that all runtime implementations are comparable. Furthermore, one has to keep in mind that the order of magnitude is small given the fact that a context switch between host and target device is done. The overhead for the first **target** construct might be much higher in all cases. However, for typical OpenMP

^aSpecial OpenMP runtime implementation for the GPU which is integrated into LLVM.

applications with either multiple target regions or one long running target region this will not limit the performance significantly.

2.3. SPEC Accel Benchmarks

The SPEC Accel Benchmark Suite [17, 18] provides a set of benchmarks for hardware accelerators using different programming paradigms like OpenCL, OpenACC and OpenMP for C/C++ and Fortran. We selected the benchmarks written in C and OpenMP in order to assess

1. the functionality of real-world kernels in addition to the pure construct evaluation as presented in Section 2.1;
2. the performance of different implementations on different systems.

We measured the offloading configurations listed in Tab. 3 with the appropriate number of threads and systems as mentioned in Section 2. Since the LLVM-VE implementation is still under development and thus not competitive in terms of performance at the moment, we do not consider this approach here.

Functionality All benchmarks but the *504.polbm* run successfully when offloading from x86 to x86. Since this implementation is meant as reference implementation only (i.e., uses a logical unit as target device on the same host) the usability is not limited in general. However, it still shows some interesting results. Furthermore, the table clearly shows full functionality for our source-to-source transformation-based approach and the LLVM GPU implementation. All seven benchmarks run successfully. For benchmark *554.pcg* to compile and run successfully with the GNU implementation on the Nvidia V100, the size of the arrays had to be included in the *map* clause. This was done by defining the `SPEC_NEED_EXPLICIT_SIZE` compatibility macro. Although, the benchmark *514.pomriq* compiles successfully, it crashes during the runtime (GNU).

The SPEC Accel Benchmark Suite is not a typical use case for reverse offloading from the SX-Aurora back to the x86 vector host, because the most compute-intensive parts of the codes are offloaded from the vector engine to the vector host. However, the results show that this feature is also usable for real-world kernels. To the best of our knowledge this is the only available implementation which enables such a functionality with OpenMP. Codes which vectorize, but for some exceptions (e.g., IO code parts), can benefit from this convenience.

Performance For all the results we used different configuration files in order to get the best performance for the given compiler, OpenMP runtime and target device. For instance we set the `USE_INNER SIMD` compatibility macro when using the NEC compiler. This flag ensures that the innermost loop gets vectorized as shown in Fig. 7. Here, a different (combined) constructs without simd is used on the outer loop. Both code snippets are semantically identical, but the performance differs, because in some of the more complex loops the NEC compiler will not vectorize the code with the outer SIMD directive and thus only delivers scalar performance. The performance for *503.postencil* and *557.pcsp* is increased by one order of magnitude by applying this modification.

The comparison between the source-to-source transformation-based approach (which uses the NEC compiler for the target device code) shows good performance results in comparison to the V100 GPU. Five of the benchmarks reach a better or similar performance on the SX-Aurora. The *504.polbm* benchmark is at least twice as fast as on a V100 and the *570.pbt* about one order of magnitude. However, the benchmarks *514.pomriq*, *552.pep* are one to two orders of magnitude

Table 3. Execution time of the SPEC Accel benchmarks in seconds, where “ $A \rightarrow B$ ” means offloaded from target host A to target device B and runtime errors are marked with “RE”. The used compiler version is denoted in brackets, or LLVM upstream otherwise

Benchmark	x86→SX (NEC 3.0.8)	x86→V100 (LLVM 12)	x86→V100 (GCC 9)	x86→x86	SX→x86
503.postencil	21.3 ¹	16.8	145	103	137
504.polbm	18.8	36.7	60.1	RE	94.4
514.pomriq	321 ¹	31.4	RE	11822	11976
552.pep	1923	71.1	140	639	667
554.pcg	91.6	88.6	80.2	124	240
557.pcsp	81.6	101	232	167	253
570.pbt	19.6	486	122	114	154

```
#pragma omp target teams distribute\
    parallel for simd
for (...) {
    for (...){
        // Parallel code
    }
}
```

```
#pragma omp target teams distribute\
    parallel for
for (...) {
    #pragma omp simd
    for (...){
        // Parallel code
    }
}
```

(a) Outer SIMD

(b) Inner SIMD

Figure 7. Example of inner SIMD usage

slower. The reason for that is the fact that *552.pep* does not vectorize due to dependencies in the innermost loops. In one of the hotspots of the code three inner loops are nested into an outer parallel loop. In the first inner (short) loop we have in addition to the dependencies between the loop iterations a function call and some conditionals which complicate the vectorization further. In the second inner loop we have also unresolvable loop dependencies. The third inner loop at least vectorizes partially. However, we have a non-consecutive access depending on a conditional to an array structure which makes a scatter instruction necessary. The *514.pomriq* uses an array of structures with four single precision scalar values such that the vectorization is also not optimal due to the memory access pattern. As consequence, these two benchmarks do not fit to the SX-Aurora architecture, because vectorization is the key for a good performance.

The comparison between the two different compiler / runtime implementations for the V100 shows, that the LLVM compiler outperforms the GNU compiler in most cases. For *503.postencil* this is even an order of magnitude. However, *570.pbt* is a factor of 4 faster with the GNU compiler. As before the x86 to x86 measurements are meant as a reference only. However, although the comparison of two Intel Xeon Silver CPU to a V100 or SX-Aurora are not completely fair, we see that especially *503.postencil*, *514.pomriq* and *570.pbt* benefit from OpenMP offloading to a GPU or vector engine.

¹Measured with a deviating configuration: NEC 2.5.1

3. Related Work

Besides OpenMP target device offloading, other approaches exist in order to execute compute-intensive code parts on a SX-Aurora TSUBASA vector engine. The direct use of the low-level APIs VEO [15], AVEO [16] or VHCall [3] gives the programmer full control of the data transfers and the kernel execution. Noack et al. [23] built on top of the portable Heterogeneous Active Messages (HAM) a high-level C++-only framework for SX-Aurora TSUBASA offloading. While all of the previous approaches are non-standard, Takizawa et al. present a OpenCL-like [26] programming framework [27]. Ke et al. recently presented a first SYCL implementation [19] for SX-Aurora TSUBASA, which also allows kernel offloading with a single-source programming model.

OpenMP Target device offloading infrastructures, prototypes and implementations exist also for other devices like the Intel Xeon Phi [22], the Texas Instruments Keystone II [21], Nvidia GPUs [8] or AMD GPUs [1]. Parts of the LLVM infrastructure work presented by Bertolli et al. [8] form the basis for parts of our results. In [25], Sommer et al. presented an implementation for OpenMP offloading to FPGA accelerators. Their proof-of-concept implementation is similar to our approach, although the technical realization slightly differs. Furthermore, we also support all kinds of target-related constructs (e.g., `teams` and combined directives), while their prototype focuses on the `target` construct. Another FPGA prototype implementation has been presented by Knaust et al. [20]. In their approach they are using the OpenCL backend for the bitstream generation instead of function outlining or IR code generation. Álvarez et al. [6] present an infrastructure which allows to embed the source code in addition to the device-specific code in the fat binary. This work describes an alternative offloading methodology, which only requires little support from the host compiler similar to our approach.

In a recent experience report Tian et al. [28] presented the idea of a portable GPU runtime in order to have support for Nvidia and AMD GPUs. This replacement library can be shipped in Linux distributions LLVM packages, which lowers the entry barrier for OpenMP offloading, because no vendor-specific SDKs are required. Although implementations for reverse offloading for heterogeneous systems are available [10], we presented, to the best of our knowledge, the first OpenMP implementation which gives the programmer full flexibility for target device offloading from the host system to the accelerator card or vice versa. The OpenMP Offloading evaluation suite presented in the work of Diaz et al. [13] was a great support for us in order to improve and validate our offloading implementations for SX-Aurora TSUBASA.

Conclusion

The heterogeneity trend in modern supercomputers is driven by the requirement for large compute capabilities and led to a broader range of accelerator types. From the users perspective performance portability of HPC applications is mandatory for the usability and acceptance of those different types. Due to the portability and convenient use, OpenMP is known as the de-facto standard for shared memory parallel programming. For the same reasons it can also become more popular for applications which require target device offloading to different kinds of accelerators. However, this assumes a broad support and availability of corresponding OpenMP implementations for many accelerator architectures.

In this paper, we presented three different LLVM-based approaches which enable OpenMP target device offloading from the x86 vector host to the SX-Aurora TSUBASA vector engine or vice versa. The source-to-source approach shows already a very convenient usability and a very

good performance on the VE for many real-world kernel applications. Furthermore, the approach is very competitive compared to available GPU OpenMP target device offloading implementations. This approach has the potential to be generic and compiler-independent for other devices with a C compiler available. The native LLVM-VE approach is still under development and requires some more performance improvements. However, we have shown that it already has a very good usability, because it generates correct code for most of the tests. Furthermore, this approach can overcome the current limitations of our source-to-source solution. Especially, it already enables the usage of C++ programs and will allow Fortran code generation in future. The third approach is the first full functional OpenMP implementation which allows reverse offloading from the VE to x86 VH.

Since all of these approaches are available open source and as pre-compiled packages, we believe that we provide flexible solutions for scientists who want to use the SX-Aurora TSUBASA vector engine with OpenMP target device offloading. As a future step we will improve the implementations in order to complete the offloading infrastructure and make it even more reliable, efficient and portable to arbitrary target device architectures.

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. AOMP GitHub repository. <https://github.com/ROCM-Developer-Tools/aomp>, accessed: 2021-06-24
2. Flang GitHub repository. <https://github.com/flang-compiler/f18-llvm-project>, accessed: 2021-06-24
3. Getting Started with VH Call - libsysve. https://www.hpc.nec/documents/veos/en/libsysve/md_doc_VHCall.html, accessed: 2021-06-24
4. NEC & RWTH Aachen University GitHub repositories. <https://github.com/sx-aurora-dev>, <https://github.com/RWTH-HPC>, <https://rwth-hpc.github.io/sx-aurora-offloading>, accessed: 2021-06-24
5. Sollve_vv GitHub repository. https://github.com/SOLLVE/sollve_vv, accessed: 2021-06-24
6. Álvarez, Á., Ugarte, I., Fernández, V., Sánchez, P.: OpenMP Dynamic Device Offloading in Heterogeneous Platforms. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) OpenMP: Conquering the Full Hardware Spectrum. Lecture Notes in Computer Science, vol. 11718, pp. 109–122. Springer (2019). https://doi.org/10.1007/978-3-030-28596-8_8
7. Antao, S.F., Bataev, A., Jacob, A.C., et al.: Offloading Support for OpenMP in Clang and LLVM. In: Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC, Salt Lake City, UT, USA, Nov. 14, 2016. pp. 1–11. LLVM-HPC, IEEE (2016). <https://doi.org/10.1109/LLVM-HPC.2016.006>

8. Bertolli, C., Antao, S.F., Bercea, G.T., *et al.*: Integrating GPU Support for OpenMP Offloading Directives into Clang. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. ACM (2015). <https://doi.org/10.1145/2833157.2833161>
9. Bull, J.M.: Measuring Synchronisation and Scheduling Overheads in OpenMP. In: Proc. of the 1st European Workshop on OpenMP. pp. 99–105. Lund, Sweden (1999)
10. Chen, C., Yang, W., Wang, F., *et al.*: Reverse Offload Programming on Heterogeneous Systems. IEEE Access 7, 10787–10797 (2019). <https://doi.org/10.1109/ACCESS.2019.2891740>
11. Cramer, T., Schmidl, D., Klemm, M., an Mey, D.: OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison. In: Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University. pp. 38–44 (2012)
12. Cramer, T., Römmер, M., Kosmyrin, B., *et al.*: OpenMP Target Device Offloading for the SX-Aurora TSUBASA Vector Engine. In: Wyrzykowski, R., Deelman, E., Jack Dongarra, K.K. (eds.) Parallel Processing and Applied Mathematics: 13th International Conference, PPAM 2019. Theoretical Computer Science and General Issues, vol. 12043, pp. 237–249. Springer (2020). https://doi.org/10.1007/978-3-030-43229-4_21
13. Diaz, J.M., Pophale, S., Friedline, K., *et al.*: Evaluating Support for OpenMP Offload Features. In: Proceedings of the 47th International Conference on Parallel Processing Companion. pp. 31:1–31:10. ICPP ’18, ACM (2018). <https://doi.org/10.1145/3229710.3229717>
14. Diaz, J.M., Pophale, S., Hernandez, O., *et al.*: OpenMP 4.5 Validation and Verification Suite for Device Offload. In: Evolving OpenMP for Evolving Architectures, IWOMP 2018. Lecture Notes in Computer Science, vol. 11128, pp. 82–95. Springer (2018). https://doi.org/10.1007/978-3-319-98521-3_6
15. Focht, E.: VEO and PyVEO: Vector Engine Offloading for the NEC SX-Aurora Tsubasa. In: Resch, M.M., Kovalenko, Y., Bez, W., *et al.* (eds.) Sustained Simulation Performance 2018 and 2019. pp. 95–109. Springer (2020). https://doi.org/10.1007/978-3-030-39181-2_9
16. Focht, E.: Speeding Up Vector Engine Offloading with AVEO. In: Resch, M.M., Wossough, M., Bez, W., *et al.* (eds.) Sustained Simulation Performance 2019 and 2020. pp. 35–47. Springer (2021). https://doi.org/10.1007/978-3-030-68049-7_3
17. Juckeland, G., Brantley, W.C., Chandrasekaran, S., *et al.*: SPEC ACCEL: A standard application suite for measuring hardware accelerator performance. In: Jarvis, S.A., Wright, S.A., Hammond, S.D. (eds.) High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation - 5th International Workshop, PMBS 2014. Lecture Notes in Computer Science, vol. 8966, pp. 46–67. Springer (2014). https://doi.org/10.1007/978-3-319-17248-4_3
18. Juckeland, G., Hernandez, O.R., Jacob, A.C., *et al.*: From Describing to Prescribing Parallelism: Translating the SPEC ACCEL OpenACC Suite to OpenMP Target Directives. In: Taufer, M., Mohr, B., Kunkel, J.M. (eds.) High Performance Computing. ISC High Performance 2016. Lecture Notes in Computer Science, vol. 9945, pp. 470–488. Springer (2016). https://doi.org/10.1007/978-3-319-46079-6_33

19. Ke, Y., Agung, M., Takizawa, H.: NeoSYCL: A SYCL Implementation for SX-Aurora TSUBASA. In: The International Conference on High Performance Computing in Asia-Pacific Region. p. 50–57. HPC Asia 2021, ACM (2021). <https://doi.org/10.1145/3432261.3432268>
20. Knaust, M., Mayer, F., Steinke, T.: OpenMP to FPGA Offloading Prototype Using OpenCL SDK. In: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 387–390. IEEE (2019). <https://doi.org/10.1109/IPDPSW.2019.00072>
21. Mitra, G., Stotzer, E., Jayaraj, A., Rendell, A.: Implementation and Optimization of the OpenMP Accelerator Model for the TI Keystone II Architecture. In: Using and Improving OpenMP for Devices, Tasks, and More, IWOMP 2014. Lecture Notes in Computer Science, vol. 8766, pp. 202–214. Springer (2014). https://doi.org/10.1007/978-3-319-11454-5_15
22. Newburn, C.J., Dmitriev, S., Narayanaswamy, R., *et al.*: Offload Compiler Runtime for the Intel® Xeon Phi Coprocessor. In: 2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, May 20-24, 2013. pp. 1213–1225. IEEE (2013). <https://doi.org/10.1109/IPDPSW.2013.251>
23. Noack, M., Focht, E., Steinke, T.: Heterogeneous Active Messages for Offloading on the NEC SX-Aurora TSUBASA. In: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 26–35. IEEE (2019). <https://doi.org/10.1109/IPDPSW.2019.00014>
24. OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 5.0 (2018)
25. Sommer, L., Korinth, J., Koch, A.: OpenMP device offloading to FPGA accelerators. In: 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP), Seattle, WA, USA, July 10-12, 2017. pp. 201–205. IEEE (2017). <https://doi.org/10.1109/ASAP.2017.7995280>
26. Stone, J.E., Gohara, D., Shi, G.: OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. Computing in Science Engineering 12(3), 66–73 (2010). <https://doi.org/10.1109/MCSE.2010.69>
27. Takizawa, H., Shiotsuki, S., Ebata, N., Egawa, R.: An OpenCL-Like Offload Programming Framework for SX-Aurora TSUBASA. In: 2019 20th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT). pp. 282–288. IEEE (2019). <https://doi.org/10.1109/PDCAT46702.2019.00059>
28. Tian, S., Chesterfield, J., Doerfert, J., Chapman, B.: Experience Report: Writing a Portable GPU Runtime with OpenMP 5.1 (2021)
29. Yamada, Y., Momose, S.: Vector Engine Processor of NEC’s Brand-New Supercomputer SX-Aurora TSUBASA. Hot Chips Symposium on High Performance Chips (2018), accessed: 2021-06-24

Performance and Power Analysis of a Vector Computing System*

Kazuhiko Komatsu¹  **Akito Onodera²,** **Erich Focht³**  **Soya Fujimoto⁴,** **Yoko Isobe⁴,** **Shintaro Momose⁴,** **Masayuki Sato²**  **Hiroaki Kobayashi²** 

© The Authors 2021. This paper is published with open access at SuperFri.org

The performance of recent computing systems has drastically improved due to the increase in the number of cores. However, this approach is reaching the limitation due to the power constraints of facilities. Instead, this paper focuses on a vector processing with long vector length that has a potential to realize high performance and high power efficiency. This paper discusses the potential through the optimization of two benchmarks, the Himeno and HPCG benchmarks, for the latest vector computing system SX-Aurora TSUBASA. The architecture of SX-Aurora TSUBASA owes the high efficiency to making good of its long vector length. Considering these characteristics, various levels of optimizations required for a large-scale vector computing system are examined such as vectorization, loop unrolling, use of cache, domain decomposition, process mapping, and problem size tuning. The evaluation and analysis suggest that the optimizations improve the sustained performance, power efficiency, and scalability of both benchmarks. Therefore, it is clarified that the SX-Aurora TSUBASA architecture can achieve higher power efficiency due to its high sustained memory bandwidth paired with the long vector computing.

Keywords: SX-Aurora TSUBASA, optimization, vector computing, power efficiency, Himeno benchmark, HPCG.

Introduction

The performance of recent high-performance computing (HPC) systems has been remarkably improved. One of the main factors is the increase in the number of nodes. A large number of nodes are clustered into an HPC system. For example, Supercomputer Fugaku, the top 1 system in the TOP500 ranking as of November 2020, is equipped with 158,976 computing nodes and 7,630,848 cores [5]. The large number of nodes brings the improvement of the peak performance. The other factor is the improvement of a core in a processor. The improvement of a core performance is mainly due to the improvement of vector processing. Vector processing has been adopted by various recent processors in shape of SIMD units, AVX-512 instruction architecture (ISA) for Intel Xeon, AVX-2 ISA for AMD EPYC. GPUs from NVIDIA and AMD support vectorization in the SIMT manner, while Fujitsu A64FX implements the ARM SVE as SIMD units with a vector ISA. The NEC SX dedicated vector processors implement a long vector ISA combining SIMD with pipelining.

The performance growth comes with a considerable increase in the power consumption of HPC systems. Due to the limitation of the power supply capacity of each system, the conventional approach to improve the performance by simply increasing the number of nodes is reaching the limit. For the design of future HPC systems, a paradigm shift to another new approach is essential to maximize performance within limited power constraints.

*This paper is an extended version of the following two papers, A. Onodera, et al., “Optimization of the Himeno Benchmark for SX-Aurora TSUBASA,” Proceedings of International Symposium on Benchmarking, Measuring and Optimizing (Bench20), 2020, and K. Komatsu, et al., “Performance Evaluation of a Vector Supercomputer SX-Aurora TSUBASA,” Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC18), 2018, by adding optimizations of HPCG and evaluations of the Himeno and HPCG benchmarks on large vector computing systems.

¹Cyberscience Center, Tohoku University, Miyagi, Japan

²Graduate School of Information Sciences, Tohoku University, Miyagi, Japan

³NEC Deutschland GmbH, Germany

⁴NEC Corporation, Japan

This paper focuses on a computing system that uses a long vector ISA, which is one of the most promising technologies for high power efficiency. To exploit the potential of the computing system, this paper takes an approach to the enhancement of the sustained performance by the optimizations for not only a single node but also multiple nodes on a vector computing system. So far, there have been many efforts for the single node optimizations for a long vector ISA to accelerate HPC applications [10, 13, 19, 20]. Especially, focusing on a high data supply capability to cores in a vector processor, many memory-intensive HPC applications such as computational fluid dynamics simulation [22] have been accelerated. This paper examines NEC's latest vector system named *SX-Aurora TSUBASA* that adopts commodity interconnects such as InfiniBand for inter-node communication [24, 30]. Since multiple node optimizations as well as the single-node optimizations are important, this paper optimizes, two benchmark programs, the Himeno [1] and HPCG [2, 9, 17] benchmarks by applying not only the single optimizations such as vector optimizations, loop unrolling, and efficient use of the cache, but also the multiple node optimizations such as appropriate process mapping, tuning of domain decomposition. By performance evaluation and analysis in terms of sustained performance, scalability, and power consumption, the power efficiency of a large-scale SX-Aurora TSUBASA is investigated.

The contributions of this paper are the following.

1. The potential of a large-scale vector computing system SX-Aurora TSUBASA is investigated. By applying optimizations to the Himeno and HPCG benchmarks, the effectiveness of the optimizations is discussed using evaluations on two generations of SX-Aurora TSUBASA.
2. The power efficiency of the vector computing system SX-Aurora TSUBASA is quantitatively discussed by detailed power analysis.

The rest of this paper is organized as follows. Section 1 explains a large-scale vector computing system of SX-Aurora TSUBASA. Section 2 describes optimizations of the Himeno and HPCG benchmarks for SX-Aurora TSUBASA. Section 3 discusses the effectiveness of the optimization and investigates the power efficiency of SX-Aurora TSUBASA through evaluation. Section 4 introduces related work. Section 4 describes the conclusions of this paper.

1. Overview of Vector Computing Systems

1.1. SX-Aurora TSUBASA Vector Computing System

NEC SX is a series of vector supercomputer systems that has been continuously developed since 1983. SX-Aurora TSUBASA is the latest vector computing system based on not only the long term experience and accumulated knowledge but also new strategies that improve the flexibility and usability.

Figure 1 shows two generations of SX-Aurora TSUBASA. Figure 1a sketches the first generation of SX-Aurora TSUBASA called an A300-8 model. One *Vector Host (VH) node* consists of one VH and eight Vector Engines (*VEs*). Eight of the first generation VEs are connected to one VH through two 16-lane PCI Express (PCIe) generation 3.0 switches. The eight VEs are divided into two *VE groups*. Four VEs in each VE group and one InfiniBand EDR HCA are connected to each PCIe switch.

Figure 1b depicts the second generation of SX-Aurora TSUBASA called a B401-8 model. Eight of the second generation of VEs are used in the B401-8 model. The eight VEs are divided

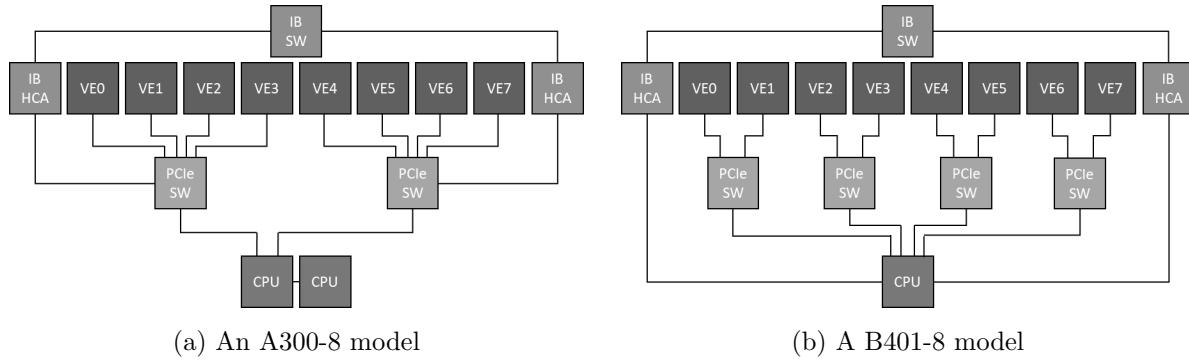


Figure 1. Two generations of SX-Aurora TSUBASA

into four VE groups. Two VEs in a VE group are connected to a PCIe generation 3.0 switch. The two InfiniBand HDR HCAs are connected to a CPU through 16-lane PCIe generation 4.0.

There are four paths to communicate among processes on SX-Aurora TSUBASA: communication within a VE node, communication within a VE group, communication among VE groups, and communication among VHs. Since each path has different bandwidth and latency, it is necessary to optimize parallel computing considering the difference of communication paths.

A VH is a common x86 Linux node equipped with an x86 processor such as Intel Xeon and AMD EPYC. A VE is a dedicated vector processor attached to a VH through PCI Express. An application is basically executed on a VE as a primary processor responsible for main calculations. A VH mainly manages VEs and performs OS-related tasks such as system calls from a VE. The OS-related tasks are transparently offloaded to a VH from a VE. This transparent offload enables a programmer to use the vector processor without any special effort such as the specifications of computational kernels and OS-related tasks. Furthermore, this execution model of SX-Aurora TSUBASA can reduce frequent data transfers between a VE and a VH. Such data transfers become one of the major bottleneck factors on an ordinary accelerator.

Moreover, SX-Aurora TSUBASA supports two explicit offload mechanisms: *VH call* and *VEO (VE offload)*. VH call can offload scalar-friendly computations such as serial computation and system calls to a VH from a VE by explicitly specifying a part of an application to be offloaded. On the other hand, VEO is used for programs executed on a VH as a primary processor. By VEO, a part of an application is offloaded to a VE, and the VE acts as a secondary processor, which is close to the execution model of an ordinary accelerator. Using VEO, vector-friendly computations such as main computations are explicitly offloaded to a VE from a VH. These offloading mechanisms allow the SX-Aurora TSUBASA to support various execution models. This flexibility contributes to improvements of usability and effective usage of the computational resources by considering characteristics of applications and processors.

1.2. Vector Engine

A VE is a vector processor that mainly contributes to the system performance based on its vector computing capability. Figure 2 shows an architecture of a VE. The architecture of two generations of VE is the same. The VE is equipped with eight vector cores. As the core performance of VE Type 10B called VE 10B is 537.6 Gflop/s for single-precision (SP) floating-point calculations, the socket performance reaches 4.30 Tflop/s. In the case of the second generation of VE Type 20B called VE 20B, the core performance is 614.4 Gflop/s (SP), resulting in the socket performance of 4.92 Tflop/s. The vector length of each vector core is 256 double-precision

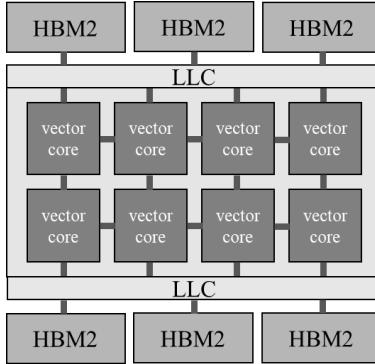


Figure 2. Architecture of a VE

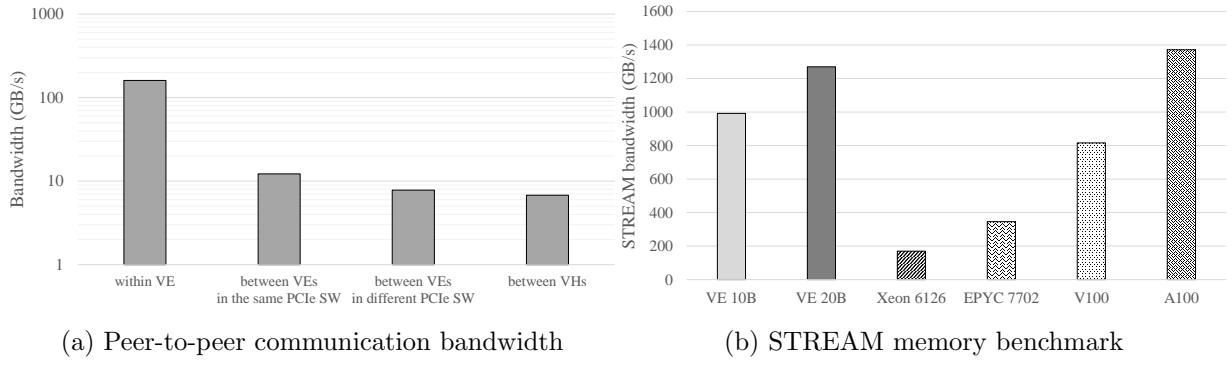
floating-point elements. It is much longer than that of recent x86 processors which have the SIMD length of 8 double-precision words in 512-bit SIMD units. The eight vector cores share a total 16 MB last level cache (LLC). Each core and the LLC are connected by a two-dimensional mesh network. Furthermore, six High Bandwidth Memory (HBM) modules work together as the main memory [18]. VE 10B and VE 20B use HBM2 [7] and HBM2E [26], respectively. As a result, their memory bandwidths reach 1.22 TB/s and 1.53 TB/s, which are much higher than those achieved with conventional DDR memory modules.

An optional configuration mode of the VE is the *partitioning mode*. In partitioning mode, vector cores, LLC, and main memory are virtually partitioned into two same capability segments. A VE can be treated as two independent partitioning nodes. Since these nodes are isolated from each other, conflicts of the communication between LLC and vector cores can be reduced. Thus, the partitioning mode is useful for an application whose bottleneck is the LLC bandwidth. On the other hand, one vector core can use only the half of memory bandwidth and capacity of the VE processor. Because of this trade-off, the partitioning mode should be used considering characteristics of target applications.

1.3. Multiple Levels of Bandwidths of SX-Aurora TSUBASA

In order to examine the various memory levels bandwidths of SX-Aurora TSUBASA, preliminary evaluations are conducted. Figure 3a shows the peer-to-peer network bandwidths of the four communication paths using the *osu_bw* kernel of the OSU Micro-Benchmarks [3]. The vertical axis shows the bandwidth when the message size is 512 KB. The horizontal axis shows the communication paths on the B401-8 systems. This figure shows that the network bandwidth is fast in the order of communication within a VE, within a VE group, with different VE groups, and between a VH node. Therefore, to efficiently exploit the potential of a system, it is necessary to take care of the differences in the communication bandwidths. Since the bandwidths are different among various paths, the communication that requires high bandwidth should use high bandwidth paths, e.g., by localizing communication as much as possible.

Figure 3b shows the memory bandwidth using the *triad* kernel of the STREAM benchmark [4]. The vertical axis shows the stream memory bandwidth. The horizontal axis shows the tested processor types. VE 10B, VE 20B, two sockets of Intel Xeon Gold 6126, called Xeon 6126, and two sockets of AMD EPYC 7702 are used as x86 processors. Nvidia TESLA V100 and A100 are used as GPUs. This figure shows that VE 20B and A100 achieve the highest memory bandwidths. Although the memory bandwidths of VE 10B and V100 are lower than those of VE 20B and A100, they are higher than those of two sockets of Xeon 6126 and two sockets of EPYC

**Figure 3.** Bandwidth of SX-Aurora TSUBASA

7702. The main reason for the differences in the bandwidth comes from the memory subsystems that each processor adopts. VE 20B and A100 are HBM2E memory modules, VE 10B and V100 are the HBM2 memory modules, and Xeon 6126 and EPYC 770 are the DDR4 memory DIMMs. These differences affect the stream memory bandwidths. To take a close look at the figure, the stream memory bandwidth of A100 is about 8.0 % higher than that of VE 20B. The efficiency to the peak memory bandwidth of A100 is higher than that of VE 20B. The efficiencies of A100 and VE 20B are 88.2 % and 82.7 %, respectively. The operational frequency of HBM2E and the efficiencies lead to the differences in the memory bandwidths between VE20B and A100. As a result of the preliminary evaluation, it is essential for the optimization of applications to fully exploit the various bandwidths considering the characteristics of a single node and multiple nodes.

2. Optimization Techniques for a Vector Computing System

To exploit the potential of a vector computing system, optimizations for a single node and multiple nodes are essential. For target computations, this paper chooses important kernels frequently used in memory-bound HPC applications: a stencil computation and a conjugate gradient (CG) computation. There are two famous benchmark programs including these types of the kernels, the Himeno and HPCG benchmarks. In this section, by briefly investigating the characteristics of the benchmarks, optimizations such as vectorization, exploitation of memory and LLC bandwidths, domain decomposition, and process mapping are applied to these benchmark programs.

2.1. Optimizations for Stencil Computations

Stencil computations are one of the important kernels in the field of HPC and data sciences. The Himeno benchmark is one of the benchmark programs that measures the performance of the stencil computations. The Himeno benchmark solves the Poisson equation by the Jacobi method in the incompressible fluid analysis [1]. The main kernel named Jacobi requires high memory bandwidth because it performs stencil calculations that continuously update grid points using the values of adjacent grid points. In the Jacobi kernel, 19-point stencil calculations are performed for an array p of the pressure term. By updating the array p in a triple loop in the i , j , and k directions, 19 references of array p occur in one iteration.

First, to understand the characteristics of the Himeno benchmark, its code is briefly analyzed using four Bytes/Flop (B/F) ratios: *required B/F*, *actual B/F*, *memory B/F*, and *LLC B/F* [11,

24]. The required B/F ratio is defined as the ratio of bytes of the number of load and store instructions to the number of floating-point operation instructions. The required B/F ratio of the Himeno benchmark is 3.33. The actual B/F ratio is calculated from the number of actual memory accesses that take into account the actual behavior of LLC divided by the number of actual floating-point operation instructions. The actual B/F ratio of the Himeno benchmark is 2.24 in VE 10B and VE 20B. The memory and LLC B/F ratios are defined as the ratios of the peak memory and LLC bandwidths to the peak computing performance. The memory B/F ratios of VE 10B and 20B are 0.28 and 0.31, respectively. The LLC B/F ratios of VE 10B and 20B are 0.62 and 0.61, respectively. By comparing with the four B/F ratios, the Himeno benchmark is judged as a memory bandwidth-bound application even on vector computing systems equipped with high memory bandwidth.

To exploit the bandwidth of SX-Aurora TSUBASA, the optimizations for improving utilization of the LLC, loop unrolling, domain decomposition, and process mapping are applied [27]. The first optimization is the efficient use of the LLC. Since each element in an array p is used 19 times in the Jacobi kernel, 18 times of memory accesses can be reduced if the element is stored in the LLC. For the 19-point stencil calculation, three planes need to be stored in LLC to reuse an element 18 times if the size of three planes can fit the LLC. Therefore, the priority of the cache retention for array p sets to be high by using a dedicated compiler directive.

The next optimization is loop unrolling to reduce the loop overhead. As the Jacobi kernel is the triple nested loop and the number of loop iterations is large, the cost of controlling the loop such as loop condition tests and increments of loop indices cannot be ignored, especially on vector computing systems. By applying loop unrolling, the overhead is reduced. As the innermost loop is used for the vectorization and the outermost loop is used for parallelization, the second loop is unrolled. As VE has more vector registers than a general-purpose processor, the number of unrolls can be large, which is more effective in general. This paper selects the best parameter of the number of unrolls by the brute-force search in the range of 2^0 to 2^6 .

The third optimization is the tuning of the domain decomposition. For the MPI version of the Himeno benchmark, it is necessary to decompose the three-dimensional domain for parallel processing. To keep a sufficiently large vector length for the computation, the innermost loop should be carefully selected. Thus, the length in the k direction should be at least 256. Moreover, the decomposed domain in the j direction should be smaller than that in the i direction as the memory accesses to the i direction are sequential. The appropriate domain decomposition is searched by brute-force as the patterns of the domain decomposition are not so large.

The last optimization is process mapping. A halo communication between two processes that calculate adjacent domains is one of the most bandwidth-bound communication parts in the Himeno benchmark. As there is a bandwidth difference of each communication path shown in Fig. 3a, adjacent processes are carefully assigned in the same VE and the same VE group rather than the VH-VH communication and the VH groups communication. Furthermore, considering the balance of the communication load in each communication path, the process assignment is equally distributed.

2.2. Optimizations for the Conjugate Gradient Method

The CG method is one of algorithms to solve linear equations. The CG method is generally used for large sparse systems that are difficult to handle by direct numerical methods. The HPCG benchmark [2, 9, 17] is one of the benchmark programs that measures the performance

of the CG computation. The HPCG solves a linear equation $Ax = b$ with a symmetric sparse matrix discretized by the finite element method using a multi-grid preconditioned conjugate gradient (CG) method with a symmetric Gauss-Seidel smoother. According to the CG method, the linear equation $Ax = b$ results in finding x that minimizes $f(x) = \frac{1}{2}x^T Ax - b^T x + c$. The CG method solves simultaneous linear equations by the iterative method. The method is often used in large-scale sparse matrix coefficients that would require a huge number of calculations and memory in direct methods like Gauss elimination. The required B/F ratio of the reference version of the HPCG benchmark is 8.31, making it a very realistic benchmark for memory-bound applications and an ideal candidate for exploiting the large memory bandwidth of the VE.

By using the B/F ratios, the characteristics of the HPCG benchmark is briefly analyzed. The required and actual B/F ratios of the HPCG benchmark are 7.62 and 5.80, respectively. The memory B/F ratios of VE 10B and 20B are 0.28 and 0.31, respectively. The LLC B/F ratios of VE 10B and 20B are 0.62 and 0.61, respectively. By comparing with the four B/F ratios, it is clarified that the HPCG benchmark is an LLC bandwidth-bound program.

This paper uses the code optimized for SX-Aurora TSUBASA [16]. The code is based on a vectorized version of the reference algorithm that has achieved 11.2 % efficiency for the SX-ACE processors [12, 14, 21] by the following important optimizations: ELLPACK data format for the sparse matrix, hyperplanes or level scheduling ordering for vectorization, and cache retention control for variables in the SX-ACE advanced data buffer (ADB) for data reuse. Instead of the cache retention control for ADB on SX-ACE, the priority of the cache retention for the LLC on VE is controlled by the dedicated compiler directive.

Besides vectorization and optimal data access, the highest impact on the performance is reformulating the Gauss-Seidel smoother implementation to significantly reduce the number of operations. When storing the matrix A in three parts, strictly lower and upper matrices L and U as well as the diagonal D , a symmetric Gauss-Seidel step can be expressed as follows.

$$(L + D)\mathbf{x}^{(k+1/2)} = \mathbf{b} - U\mathbf{x}^{(k)} \quad (\text{forward substitution}) \quad (1)$$

and

$$(U + D)\mathbf{x}^{(k+1)} = \mathbf{b} - L\mathbf{x}^{(k+1/2)} \quad (\text{backward substitution}) \quad (2)$$

with $\mathbf{x}^{(k)}$ being the k^{th} iteration of \mathbf{x} . After computing the temporary vector $\mathbf{r} = U\mathbf{x}^{(k)}$ through a sparse matrix-vector multiplication (SpMV), the value of $\mathbf{x}^{(k+1/2)}$ is computed through a triangular solve (TRSV) operation by fulfilling the following equation.

$$(L + D)\mathbf{x}^{(k+1/2)} = \mathbf{b} - \mathbf{r}. \quad (3)$$

Thus, the right-hand side of (2) can be computed as follows.

$$\mathbf{b} - L\mathbf{x}^{(k+1/2)} = \mathbf{r} + D\mathbf{x}^{(k+1/2)}, \quad (4)$$

which leads to compute $\mathbf{x}^{(k+1)}$ from Eq. 2 as the result of a backward triangular solve. A matrix forward and backward substitutions are replaced by the forward and backward substitutions of only the $L + D$ and $U + D$ matrices while saving almost half of the loads and operations with the expense of one fast SpMV, fast vectorizable element-wise product $D\mathbf{x}$, and vector additions/subtractions. Additionally, operations in the first, fine-grained smoother step of the V-shaped multi-grid are saved by using the zero initial guess $\mathbf{x}^{(0)} = 0$ that leads to $\mathbf{r} = 0$. This

algorithm is also applied by other architectures such as Intel optimized HPCG provided with the MKL library and GPU HPCG implementations like rocHPCG [25, 28, 29].

In the MPI parallelized version, the matrix can be decomposed into a purely local part and a halo matrix containing domain boundary elements. This separation allows for some extent of overlap between computation and communication that improves scalability.

Furthermore, the matrix size should be appropriately specified. To perform efficient vector computing by keeping an enough long vector length, the y-axis and z-axis sizes need to be long. As the matrix size affects the convergence of the calculation results, the matrix size should be carefully selected considering the residual. This paper searches for the optimum matrix size. To reduce the search space, the matrix size suitable for a single node is searched. For a single node, $(nx, ny, nz) = (56, 216, 376)$ achieves the highest performance. Then, based on the suitable matrix size for a single node, the size for multiple nodes is searched. By fixing the value of nz to 376 in order to keep the vector length, nx and ny are searched.

This paper also uses the partitioning mode to further exploit the potential of LLC. As the HPCG benchmark is an LLC bandwidth-bound program, the partitioning mode that reduces the contention to LLC is more suitable than the normal mode.

3. Evaluation

3.1. Evaluation Environments

Table 1. Computing systems used for the evaluations

Systems	A300-8	B401-8	Xeon	EPYC	V100	A100
Host	2×Xeon 6126	EPYC 7402P	2×Xeon 6126	2×EPYC 7702	2×Xeon 6126	
Accelerator	VE 10B	VE 20B	-	-	V100	A100
# of nodes	8 VE nodes	576 VE nodes	1 node	68 nodes	1 node	1 node
Compiler	NEC 3.2.1	NEC 3.2.0	Intel 19.1.3.304	Intel 19.1.2.254	PGI 21.2-0	

For the evaluation, six various computing systems, SX-Aurora TSUBASA A300-8, SX-Aurora TSUBASA B401-8, Xeon, EPYC, V100, and A100, are used as shown in Tab. 1. The specifications of the processors used in the systems, VE 10B, VE 20B, two sockets of Xeon 6126, two sockets of EPYC 7702, V100, and A100, are shown in Tab. 2.

For the compiler, the proprietary NEC compiler for VEs is used. The compile option “-O4 -msched-block” is used, which allows the compiler to automatically optimize and schedule the instruction in a basic block. For the general-purpose processors such as Xeon and EPYC, the Intel compiler collection is used. For V100 and A100, the PGI compiler is used.

For the Himeno benchmark, the MPI versions are used. For the evaluation on multiple nodes, the weak scale version is developed. For Xeon 6126 and EPYC 7702, only the optimization for the domain decomposition is applied to the reference codes. For V100 and A100, the parameter tuning of system parameters is optimized [23].

For the HPCG benchmark on Xeon 6126 and EPYC 7702, only the optimization for the size tuning is applied to the reference codes. For A100, the HPCG code in the NVIDIA HPC-Benchmark 21.2 is used.

To measure the power consumption of processors, Vector Engine MMM-Command, Intel SoC Watch, and NVIDIA SMI are used. For the power consumption of the whole system,

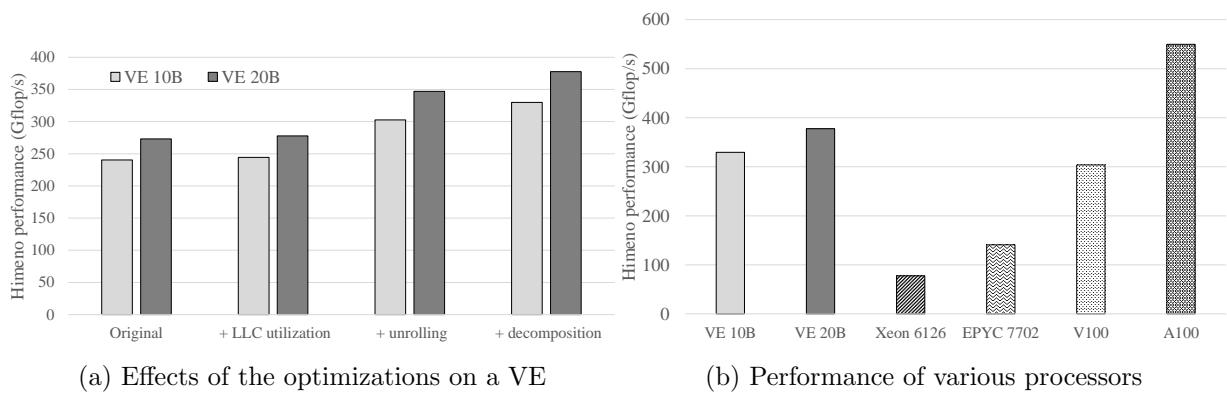
Table 2. Specification of processors used for evaluation

	VE 10B	VE 20B	Xeon 6126	EPYC 7702	V100	A100
Number of cores	8	8	12	64	5120	6912
Peak SP (Tflop/s)	4.30	4.92	1.766	4.096	14	19.5
Peak DP (Tflop/s)	2.15	2.46	0.883	2.048	7	9.7
Memory	$6 \times$ HBM2	$6 \times$ HBM2E	$6 \times$ DDR4	$8 \times$ DDR4	$4 \times$ HBM2	$6 \times$ HBM2E
Mem. BW (GB/s)	1228	1536	128	204.8	900	1555
Mem. Cap. (GB)	48	48	192	256	32	40
LLC BW (TB/s)	2.66	3.00	-	-	2.70	6.88
LLC Cap. (MB)	16	16	19.25	256	6	40

Supermicro IPMICFG is used. The execution times of the Himeno and HPCG benchmark are set to 10 minutes and two minutes, respectively.

3.2. Evaluation Results

3.2.1. Evaluation of the Himeno benchmark

**Figure 4.** Performance of the Himeno benchmark on a single node

First, to examine the effects of the optimizations on a single VE node, Fig. 4a shows the performance on VE 10B and VE 20B. The vertical axis represents the Himeno performance. The horizontal axis represents each optimization. “+LLC utilization”, “+unrolling”, and “+decomposition” indicate that each optimization is applied in the order of LLC utilization, loop unrolling, and tuning of decomposition parameters from “Original”. In the original code, the decomposition parameter is set to $(i, j, k) = (2, 2, 2)$.

Figure 4a shows that each optimization improves the Himeno performance. In particular, the loop unrolling has a great impact on the performance improvement. The loop unrolling achieves about 23.9 % and 24.9 % performance improvements on VE 10B and 20B, respectively. This is because the loop unrolling reduces loop overheads that is one of the large overheads. Moreover, the tuning of the decomposition parameters improves about 9.0 % and 8.7 % on VE 10B and VE 20B, respectively. The sequential memory access along the i direction by the tuning of the domain decomposition contributes to the performance improvement. As a result, the single-node optimizations achieve about 37.3 % and 38.3 % sustained performance improvements on VE 10B and VE 20B compared to the original code, respectively. Figure 4a also shows that the performances of VE 20B are higher than those of VE 10B. This performance improvement is

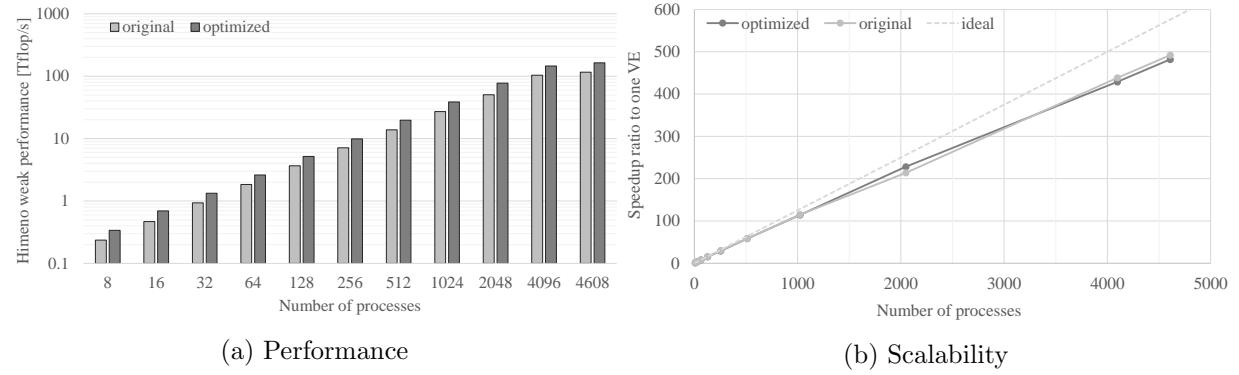


Figure 5. Weak scale performance of the Himeno benchmark on the B401-8 system

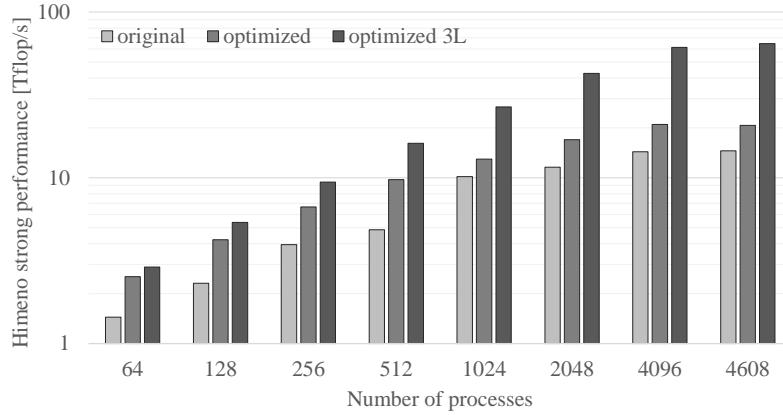
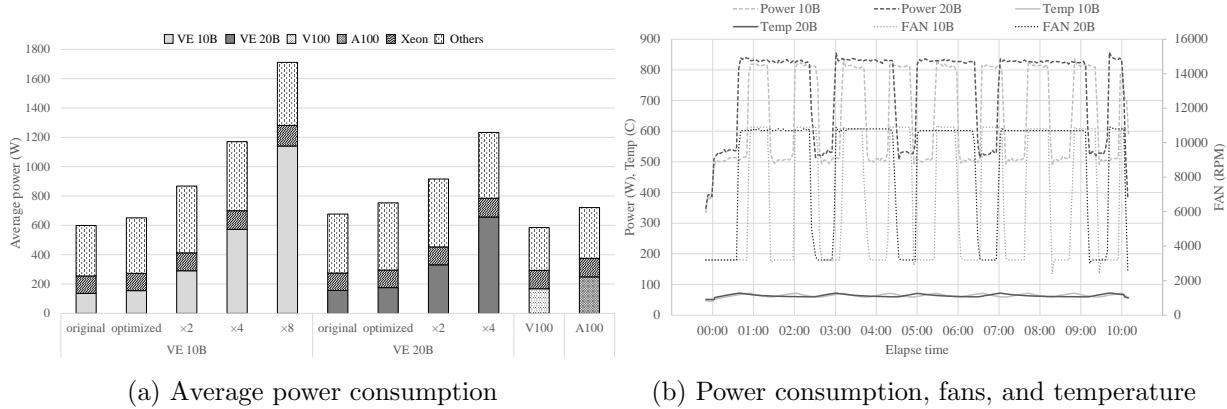
mainly brought by the improvement of the computational capability and the memory bandwidth of VE 20B.

To compare the Himeno performance of VE 10B and VE 20B with other processors, Fig. 4b shows the Himeno performance on various processors. This figure shows that A100 achieves the highest performance. A100 achieves about 43.3 % higher performance than VE 20B even though the peak memory bandwidth of VE 20B and A100 are almost the same. One of the reasons is that the reduction operation is heavy for vector computing. Since the vector length of a VE is long, the cost for the vector reduction becomes large. The other reason is that the high bandwidth of VE cannot be exploited due to the single-precision floating-point data. A packed memory load operation that treats two single-precision floating-point elements in a one load operation is not efficiently performed. Thus, A100 achieves higher performance compared with VE 20B. Compared with Xeon, EPYC, and V100, VE 10B and VE 20B achieve high performance. Since the memory and LLC bandwidths of VEs are the highest among them, VEs can achieve the highest performance.

In the aspect of the efficiency, the ratio of the sustained performance to the peak performance, VE 10B, VE 20B, Xeon 6126, EPYC 7702, V100, and A100 are 7.7 %, 7.7 %, 2.2 %, 1.7 %, 2.2 %, and 2.8 %, respectively. VE 10B and VE 20B achieve the highest efficiencies. Since VEs are carefully designed considering a balance between the sustained memory performance and sustained computational performance for memory-intensive applications, VEs can achieve the highest performance.

Next, the performance and weak scalability on a large-scale SX-Aurora TSUBASA are examined. Figure 5a shows the Himeno performance of the weak scaling on the B401-8 system. The problem size assigned to each process is fixed to the L size of $256 \times 256 \times 512$. This size is the maximum size for the memory capacity when eight processes are assigned to a VE. The vertical axis shows the sustained performance in the log scale. The maximum number of processes is 4608 that is equivalent to 576 VE nodes or 72 VH nodes. This figure shows that the optimized version achieves higher performance than that of the original version. About 43 % on average and about 53 % at maximum performance improvements are obtained by the optimizations. These results indicate that the proposed optimizations are essential to exploit performance on the large-scale vector computing system.

Figure 5a also shows that the increase in the number of processes improves the weak-scale performance. To examine the weak scalability in detail, Fig. 5b shows the weak scalability of the Himeno benchmark on the B401-8 system. The vertical axis shows the speedup ratio to a single VE node. The horizontal axis shows the number of processes. This figure shows that

**Figure 6.** Strong scale performance of the Himeno benchmark on the B401-8 system**Figure 7.** Power consumption of the Himeno benchmark

good scalabilities are obtained in the original and optimized versions. Although the collective communication for residual in the Himeno benchmark leads to the decrease in scalabilities, the parallel efficiencies of the original and optimized versions reach 85.4 % and 83.7 % even when the number of processes is 4608.

The strong scalability on a large-scale SX-Aurora TSUBASA is examined. Figure 6 shows the Himeno performance of the strong scaling on the B401-8 system. The problem size is the XL size of $512 \times 512 \times 1024$ and the 3L size of $1024 \times 1024 \times 2048$. The vertical axis shows the sustained performance. This figure shows that the optimized version achieves higher performance than the original version. This is because the optimizations contribute to the performance improvements. In particular, when the number of processes is large, multi-node optimizations such as domain decomposition and process mapping impact the sustained performance.

Moreover, the performance of the 3L size is much higher than that of the XL size, especially when the number of processes is large. This is because the parallelisms for vectorization and parallelization in the XL size are not enough for the large-scale execution. As the number of processes increases, the performance differences between the XL size and the 3L size become large. Even the 3L size is not enough when the number of processes is large. To exploit the full potential of the system, the larger size needs to be selected according to the system size.

To clarify the power efficiency, the power consumption of SX-Aurora TSUBASA is examined. Since one of the limiting factors in the scale of computing systems is power consumption of each system, power efficiency and/or sustained performance per power is very important to-

day and future. Figure 7 shows the power consumption when the weak scale performance of the Himeno benchmark is measured. Figure 7a shows the breakdown of the average power consumption of VEs, Xeon, GPUs, and “Others.” “Others” includes the power consumption of the cooling fans, the memory modules of a VH, the power units, and the other server components. “Others” is calculated by subtracting the power consumption of the processor from the total power consumption. The horizontal axis represents the various processors of the original and optimized versions.

First, taking look at single VE cases, the power consumptions of the optimized version increases by comparing the original and optimized versions on VE 10B and VE 20B. This is because the cores and memory in VEs are fully operated by the optimizations. Moreover, VE 20B consumes about 13.4 % more power than VE 10B. This is due to the difference in the operating frequency of VE 20B and VE 10B. As the operating frequency of VE 20B is 1.6 GHz while it is 1.4 GHz in VE 10B, VE 20B is running at about 14.2 % faster frequency than VE 10B. As the power consumption is in proportion to the frequency, VE 20B consumes more power than VE 10B.

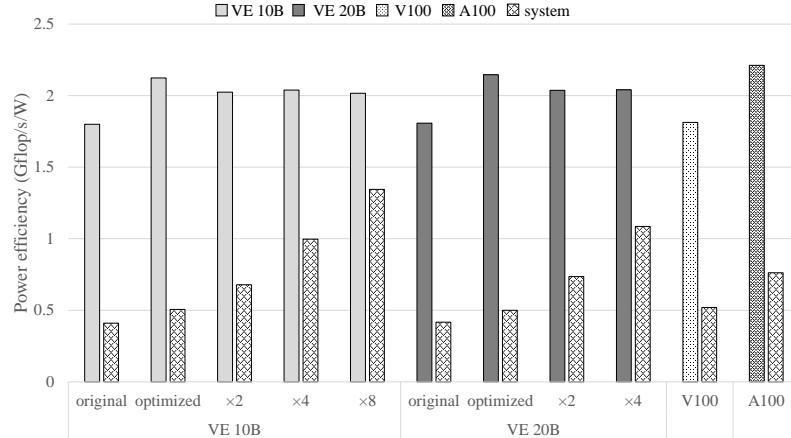
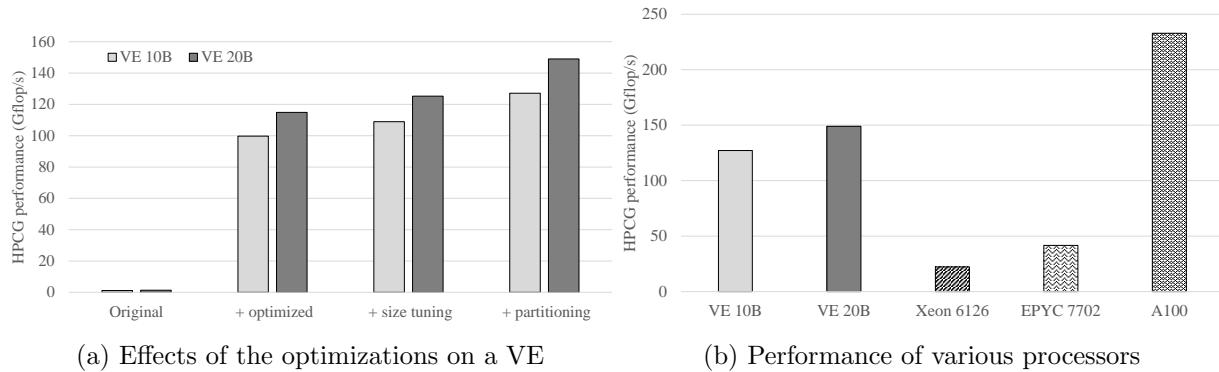
Compared with A100, the power consumptions of VE 10B and VE20B are low. However, the total power consumption of the A100 system is lower than those of the VE 10B and VE 20B systems. This is because the fine-grain fan control can be performed in the A100 system. Thus, the power consumption by the cooling fan in the A100 system becomes low.

The power consumption of “Others” in cases of a single VE occupies about 60 % of the total power consumption. One of the power consuming components is the cooling fans, especially when the fans rotate at very high speed when the temperature of VEs increases. The power consumption of “Others” on VE 20B is higher than that on VE 10B. This is because the temperature of VE 20B easily increases compared with VE 10B due to the high frequency of VE 20B.

To investigate the relationship between the power consumption and the cooling fans, Fig. 7b shows the total power consumption, the number of rotations of fans, and the temperature on VE 10B and VE 20B. The vertical axis in the left shows the power consumption and the temperature of VEs. The vertical axis in the right shows rotations per second of the cooling fan. The horizontal axis shows the elapsed time. This figure suggests that the cooling fan runs at the high speed from when the temperature of VEs rises to 70 degrees until when it drops to 60 degrees. This figure also shows that the fan of VE 10B often runs high rotations compared to VE20. This result implies that VE10 is easier to be cold enough to stop the fans than VE20 because the operating frequency of VE 10B is lower than that of VE 20B.

Second, in the cases of multiple VEs, the power consumption of VEs almost proportionally increases as the number of VEs increases in both cases of VE 10B and VE 20B. Since most computation is performed on VEs, the power consumption increases according to the number of VEs while those of Xeon and others slightly increase.

Figure 8 shows the power efficiency that divides the sustained performance by the average power. “Processor” indicates the power efficiency of the processor, i.e., the performance divided by the average power of only the processor. “System” indicates the power efficiency of the whole system. First, taking look at single node cases, this figure shows that the optimizations contribute to the power efficiencies as well as the sustained performance. The efficiencies of VE 10B and VE 20B are improved by about 18.0 % and 18.7 %, respectively. Since the increase in the power consumption can be amortized by the increase in the sustained performance, the power

**Figure 8.** Power efficiency**Figure 9.** Performance of the HPCG benchmark on a single node

efficiencies can be improved. Moreover, the power efficiencies of VE 10B, VE 20B, and A100 are similar. Even the previous generation of VE 10B achieves high power efficiency. Although the performances of VE 10B and VE 20B are lower than that of A100, the power consumption of VE 10 and VE 20B are lower than that of A100. As a result, VE 10B and VE 20B achieve a similar power efficiency to A100.

In the cases of multiple VEs, the power efficiencies of “processors” in VE 10B and VE 20B gradually decrease as the number of VEs increases. This is because the sustained performance does not ideally scale according to the number of VEs, although the power consumption increases according to the number of VEs. On the other hand, the power efficiencies of “system” increase as the number of VEs increases. As the total power consumption does not proportionally increase to the number of VEs, the increase in the total power consumption can be amortized by the increase in the sustained performance.

The process technologies used in VE 10B and VE 20B are TSMC 16 nm while the process technology of A100 is TSMC 7 nm. Even though VEs use the two-generation old process technology, the power efficiencies of VEs are similar to A100 that uses the latest process technology. If VEs use the same process technology, the power efficiencies should be much higher than that of GPUs. This result clarified that the vector architecture is a power efficient architecture that is one of important features as the power constraints become stricter in the future.

3.2.2. Evaluation of the HPCG benchmark

To examine the effects of the optimizations, Fig. 9a shows the performance of the optimizations on VE 10B and VE 20B. The vertical axis represents the sustained performance of the HPCG benchmark. The horizontal axis represents each optimization. “Original” indicates the reference version of HPCG. “+ optimized” indicates the version for the vector optimizations. “+ size tuning” indicates tuning of the problem size to $(56 \times 216 \times 376)$ from the initial problem size of $(104 \times 104 \times 104)$. “+ partitioning” uses the partitioning mode. The optimizations are applied in the order of the vector optimizations, the tuning of the matrix size, and the partitioning mode from left to right in the figure.

This figure shows that each optimization improves the HPCG performance. The optimized version that applies the ELL data format, hyperplanes or level scheduling ordering for vectorization, and cache retention for the LLC, and reduction in the instructions significantly improve the performance. The main reason is due to the increase in the vectorization ratio and the average vector length. By the optimizations, the vectorization ratio is improved to 99.2 % from 73.7 % and the average vector length is drastically improved to 236.2 from 27.9. Therefore, the performance improves by 86.8 times compared with the original version.

Figure 9a also shows that the tuning of the matrix size further improves the performance by about 9 %. As the matrix size affects the size of hyperplane slices, the average vector length is improved to 241.2.

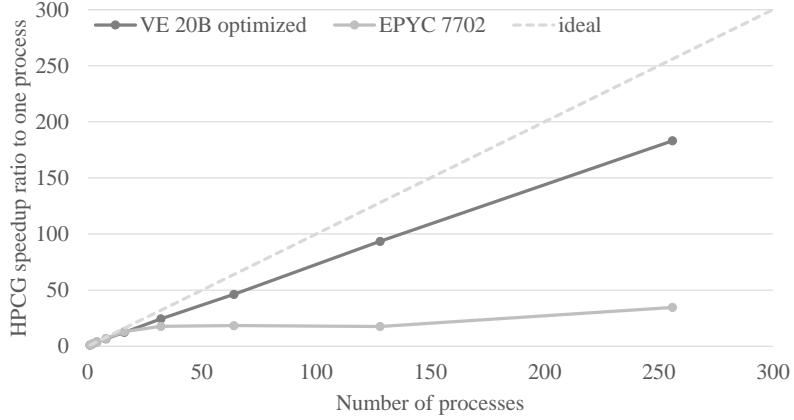
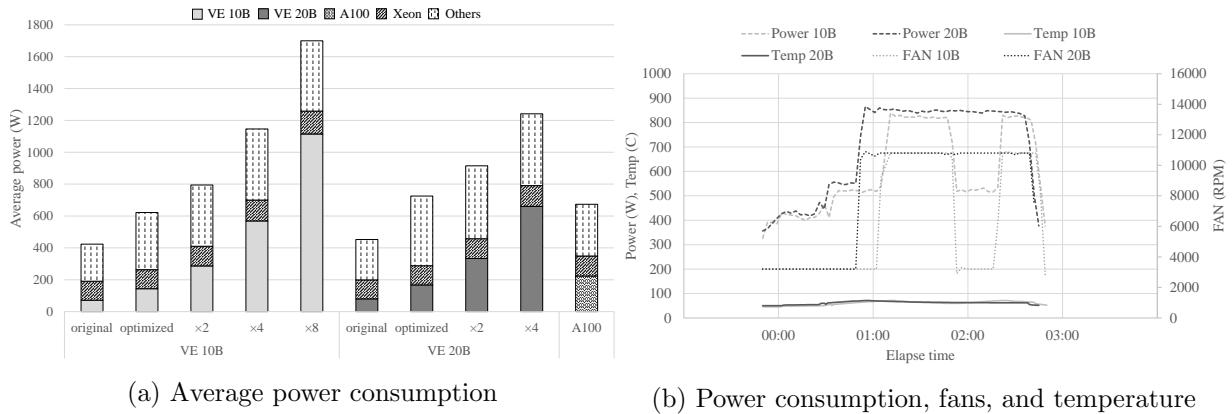
Furthermore, the partitioning mode further improves the performance. By the partitioning mode, the execution time of the load instruction becomes about 10 % shorter than that of the normal mode. The short execution time of the load instruction by the reduction in the network conflicts between LLC and the memory further results in about 19 % performance improvement.

Moreover, Fig. 9a shows that the performances of VE 20B are about 15 to 17 % higher than those of VE 10B. The main reason for the improvement is that the LLC bandwidth of VE 20B is higher than that of VE 10B. Since the LLC bandwidth of VE 20B is improved by about 12.8 %, it contributes to the higher performance.

Compared with the other processors, Fig. 9b shows the performance on VE 10B, VE 20B, two sockets of Xeon 6126, two sockets of EPYC 7702, and A100. The horizontal axis represents processors. This figure shows that A100 achieves the highest performance. The performance of A100 is 1.56 times faster than that of VE 20B. Compared with Xeon and EPYC, VE 10B and VE 20B achieve much high performance. One of the reasons is that the LLC bandwidth of A100 is higher than that of VE 20B. The theoretical LLC bandwidth of A100 is 6.88 TB/s [8], while it is 3.00 TB/s on VE 20B. As the LLC bandwidth of A100 is more than 2.29 times higher, the LLC bandwidth improves the HPCG performance on A100. The other reason is that VE 20B cannot fully exploit its high memory bandwidth due to the memory latency by the indirect memory accesses. As a result, the difference of the HPCG performance between VE 20B and A100 becomes larger than that in the case of the stream memory bandwidth.

On the other hand, the efficiencies of VEs are the highest. The efficiencies of VE 10B, VE 20B, Xeon 6126, EPYC 7702, A100 are 5.9 %, 6.1 %, 1.3 %, 1.0 %, and 2.4 %, respectively. Due to the balanced architecture of SX-Aurora TSUBASA for bandwidth-bound applications, VE 10B and VE 20B achieve higher efficiencies than the other processors.

Figure 10 shows the scalability of the HPCG benchmark on the B401-8 and EPYC systems. The vertical axis represents the speedup ratio to single process performance. The horizontal axis represents the number of processes. This figure shows that a good scalability of VE 20B

**Figure 10.** Scalability of the HPCG benchmark**Figure 11.** Power consumption of the HPCG benchmark

is obtained. When the number of processes is 256, the parallel efficiency is about 71.6 %. As the HPCG benchmark is weak scaling, the performance scales fine even when the number of processes is large.

In the case of EPYC 7702, the scalability is not good compared with VE 20. The memory bandwidth of EPYC 7702 scales up to 32 processes, and then, the memory bandwidth is saturated when the number of processes is 64 or more. As the HPCG performance on EPYC 7702 is limited by the memory bandwidth, the scalability of EPYC 7702 also becomes saturated when the number of processes is 64 or more.

To examine the power consumption, Fig. 11a shows the breakdown of the average power consumption. This figure shows that the power consumption of a VE 10B and a VE 20B increases by the optimizations. As the optimized version with the partitioning mode efficiently uses VEs, the power consumption increases. On the other hand, as the original version cannot exploit the performance of VEs, the power consumption is low.

Furthermore, the power consumption of A100 is higher than those of VE 10B and VE 20B although the total power consumption of the A100 system is lower than those of the VE 10B and VE 20B systems. This is because of the difference of the fan control mechanism between the A100 system and the VE systems, which is also discussed in the Himeno benchmark. As the fine-grain fan control can be performed in the A100 system, the power consumption by the cooling fan becomes low.

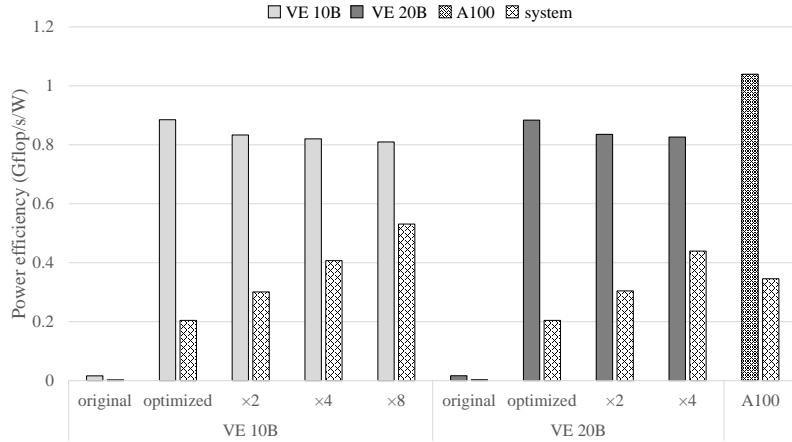

Figure 12. Power efficiency

Figure 11a also shows the power consumption on VE 20B is higher than those on VE 10B. To investigate the reason, Fig. 11b shows the total power consumption, the number of rotations of fans, and the temperature on VE 10B and VE 20B. The vertical axis in the left shows the power consumption and the temperature of VEs. The vertical axis in the right shows rotations per second of the cooling fan. The horizontal axis shows the elapsed time. This figure shows that the total power consumption of VE 20B becomes high because the cooling fans of VE 20B rotate more often than that of VE 10B. This is because VE10 is easier to be cold enough to reduce the number of rotations of the fan than VE20. Furthermore, the fan in the HPCG benchmark runs at the high rotation more often than that in the Himeno benchmark. Since the characteristics of the benchmarks differ from each other, it affects the frequency of the high rotations of the fan.

Figure 12 shows the power efficiency of the HPCG benchmark. The power efficiencies of a VE 10B and a VE 20B are almost the same even though the performance of VE 10B is lower than that of VE 20B even in the HPCG benchmark as well as the Himeno benchmark. As the power consumption of VE 10B is lower than that of VE 20B, the power efficiency of VE 10B equals to that of VE 20B. Moreover, the power efficiency of A100 is higher than those of VE 10 and VE 20B. Although the power consumption of VE 10 and VE 20B is lower than that of A100, the sustained performance of A100 is much higher than those of VE 10B and VE 20B. As a result, A100 achieves higher power efficiency than VE 10B and VE 20B. The reason is that the process technologies used in VE 10B and VE 20B are two-generation old compared with A100. If VEs use the same process technology, it is expected that VEs can achieve higher power efficiency than A100.

In the cases of multiple VEs, the power efficiencies of VE 10B and VE 20B gradually decrease as the number of VEs increases. This is because the sustained performance does not ideally scale according to the number of VEs, although the power consumption increases according to the number of VEs. On the other hand, the power efficiencies of “system” increase as the number of VEs increases. As the total power consumption does not increase in proportion to the number of VEs, the increase in the total power consumption can be amortized by the increase in the sustained performance.

4. Related Work

The performance optimization and evaluation of vector computing systems have been continuously conducted [11, 12, 15, 24]. Komatsu et al. have evaluated the first generation of SX-

Aurora TSUBASA using benchmarks including the Himeno benchmark. From the evaluation, it is clarified that the first generation of SX-Aurora TSUBASA has advantages of memory-intensive benchmark compared with Xeon Skylake and SX-ACE. However, the performance on multiple nodes is not evaluated. This paper extends the optimization of the Himeno benchmark for multiple nodes such as domain decomposition and processing mapping considering the bandwidth and clarifies the performance and scalability on multiple nodes of SX-Aurora TSUBASA.

Furthermore, the performance evaluation on the second generation of SX-Aurora TSUBASA has been reported [11]. It clarifies that the performance and power efficiencies of HPCG and HPL of VE 20B are higher than those of Xeon and EPYC. However, the detailed analysis of performance and power efficiency is not conducted. This paper further optimizes the HPCG benchmark for the large-scale vector computing systems by the size tuning. Moreover, this paper deeply evaluates and analyzes the Himeno and HPCG benchmark performances on the large-scale vector computing systems in terms of the effects of optimizations, scalability, average power consumption, power efficiency. As a result of the evaluation and deep analysis, it can be clarified that the power efficiency of a vector architecture is high and promising for the future HPC systems.

Hartwig et al. have evaluated the memory bandwidth of the A100 and the performance of sparse and batched computations [6]. This paper evaluates the performance and the power efficiency of V100 and A100 with the Himeno benchmark. It shows higher performance of A100 than that of V100. By comparing the performance of GPUs and VEs, this paper clarifies the characteristics of these processors.

Conclusions

The peak performance of recent HPC systems has been remarkably improved by the increase in the number of nodes. This approach to improve the performance has also brought the increase in the power consumption of the HPC systems. However, due to the limitation of the power budget for each facility, the conventional approach of simply increasing the number of nodes to improve the performance is not realistic in near future. Therefore, a paradigm shift to a new approach is essential to keep improving the performance within the limited power constraints for the design of future HPC systems.

This paper focuses on a vector computing system adopting a long vector processing that has a potential to realize high performance with high power efficiency under the strict power constraints. To achieve high power efficiency, this paper improves the sustained performance by the optimizations for vector computing system. As the target programs, this paper selects two benchmark programs, the Himeno and HPCG benchmarks, and applies vector optimizations for a vector computing system in aspect of a single node and multiple nodes.

By deep analysis through the performance evaluation, the sustained performance, the scalability, and the power consumption, the power efficiency of a large-scale SX-Aurora TSUBASA are clarified. For the Himeno benchmark, VE 10B and VE 20B achieve about 37.3 % and 38.3 % performance improvements by a single node optimizations compared to the original code, respectively. VE 10B and VE 20B achieve about 7.7 % efficiency, which is the highest efficiency among various processors. For the HPCG benchmark, VE 10B and VE 20B can achieve about 112 and 113 times performance improvements by single node optimizations compared to the reference code, and about 5.9 % and 6.1 % efficiencies, respectively, which also are the highest efficiencies among various processors. Furthermore, it is clarified that SX-Aurora TSUBASA

could achieve the highest power efficiencies among the latest processors such as an Intel processor, an EPYC processor, and a GPU even though VEs adopt the previous generation of the process technology. This fact suggests that the vector computing with a long vector length can achieve a high power efficient computing and the vector architecture could be most efficient if it used the latest process technology. Therefore, this paper clarified that the vector computing is one of the promising ways to survive in the design of the future computing system with the strict power constraints.

Acknowledgements

The authors would like to thank Christie Alappat from the University of Erlangen-Nuremberg for the assistance and inspiring discussions on HPCG algorithm tuning. The authors also thank large-scale HPC challenge of Cyberscience Center, Tohoku University for the large-scale executions of the supercomputing systems. This research was 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,” Grants-in-Aid for Scientific Research (A) #19H01095, Grants-in-Aid for Scientific Research (C) #20K11838, and Japan-Russia Research Cooperative Program between JSPS and RFBR, Grant number JPJSBP120214801.

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. Himeno benchmark. <http://i.riken.jp/en/supercom/documents/himenobmt/>, accessed: 2021-05-31
2. HPCG benchmark. <https://www.hpcg-benchmark.org/>, accessed: 2021-05-31
3. MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. <http://mvapich.cse.ohio-state.edu/benchmarks/>, accessed: 2021-05-31
4. STREAM: Sustainable Memory Bandwidth in High Performance Computers. <https://www.cs.virginia.edu/stream/>, accessed: 2021-05-31
5. TOP500 Supercomputer Sites, <http://www.top500.org/>
6. Anzt, H., Tsai, Y.M., Abdelfattah, A., *et al.*: Evaluating the performance of NVIDIA's A100 ampere GPU for sparse and batched computations. In: 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). pp. 26–38. IEEE (2020). <https://doi.org/10.1109/PMBS51919.2020.00009>
7. Cho, J.H., Kim, J., Lee, W.Y., *et al.*: A 1.2V 64Gb 341GB/S HBM2 stacked DRAM with spiral point-to-point TSV structure and improved bank group data control. In: 2018 IEEE International Solid - State Circuits Conference - (ISSCC). pp. 208–210. IEEE (2018). <https://doi.org/10.1109/ISSCC.2018.8310257>

8. Choquette, J., Gandhi, W.: NVIDIA A100 GPU: Performance innovation for GPU computing. In: 2020 IEEE Hot Chips 32 Symposium (HCS). pp. 1–43. IEEE (2020). <https://doi.org/10.1109/HCS49909.2020.9220622>
9. 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 (2016). <https://doi.org/10.1177/1094342015593158>
10. Egawa, R., Komatsu, K., Takizawa, H.: Designing an open database of system-aware code optimizations. In: 2017 Fifth International Symposium on Computing and Networking (CANDAR). pp. 369–374. IEEE Computer Society (2017). <https://doi.org/10.1109/CANDAR.2017.102>
11. Egawa, R., Fujimoto, S., Yamashita, T., *et al.*: Exploiting the potentials of the second generation SX-Aurora TSUBASA. In: 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). pp. 39–49. IEEE (2020). <https://doi.org/10.1109/PMBS51919.2020.00010>
12. Egawa, R., Komatsu, K., Isobe, Y., *et al.*: Performance and power analysis of SX-ACE using HP-X benchmark programs. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER). pp. 693–700. IEEE Computer Society (2017). <https://doi.org/10.1109/CLUSTER.2017.65>
13. Egawa, R., Komatsu, K., Kobayashi, H.: Designing an HPC refactoring catalog toward the exa-scale computing era. In: Resch, M.M., Bez, W., Focht, E., Kobayashi, H., Patel, N. (eds.) *Sustained Simulation Performance 2014*. pp. 91–98. Springer (2015). https://doi.org/10.1007/978-3-319-10626-7_8
14. Egawa, R., Komatsu, K., Momose, S., *et al.*: Potential of a modern vector supercomputer for practical applications: performance evaluation of SX-ACE. *The Journal of Supercomputing* 73(9), 3948–3976 (2017). <https://doi.org/10.1007/s11227-017-1993-y>
15. Egawa, R., Momose, S., Komatsu, K., Isobe, Y., Musa, A., Takizawa, H., Kobayashi, H.: Early evaluation of the SX-ACE processor. In: The poster at International Conference for High Performance Computing, Networking, Storage and Analysis (SC14) (2014)
16. Focht, E.: HPCG Performance Efficiency on VE at 5.99%. <https://sx-aurora.github.io/posts/hpcg-tuning/> (2019), accessed: 2021-06-09
17. Heroux, M.A., Dongarra, J., Luszczek, P.: HPCG benchmark technical specification (2013). <https://doi.org/10.2172/1113870>
18. Hou, S.Y., Chen, W.C., Hu, C., *et al.*: Wafer-level integration of an advanced logic-memory system through the second-generation CoWoS technology. *IEEE Transactions on Electron Devices* 64(10), 4071–4077 (2017). <https://doi.org/10.1109/TED.2017.2737644>
19. Komatsu, K., Egawa, R., Hirasawa, S., *et al.*: Migration of an atmospheric simulation code to an OpenACC platform using the Xevolver framework. In: 2015 Third International Symposium on Computing and Networking (CANDAR). pp. 515–520. IEEE Computer Society (2015). <https://doi.org/10.1109/CANDAR.2015.102>

20. Komatsu, K., Egawa, R., Hirasawa, S., *et al.*: Translation of large-scale simulation codes for an OpenACC platform using the Xevolver framework. International Journal of Networking and Computing 6(2), 167–180 (2016). https://doi.org/10.15803/ijnc.6.2_167
21. Komatsu, K., Egawa, R., Isobe, Y., *et al.*: An approach to the highest efficiency of the HPCG benchmark on the SX-ACE supercomputer. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC15), Poster. pp. 1–2 (2015)
22. Komatsu, K., Egawa, R., Takizawa, H., *et al.*: Exploring system architectures for next-generation CFD simulations in the postpeta-scale era. Journal of Fluid Science and Technology 9(5), JFST0073–JFST0073 (2014). <https://doi.org/10.1299/jfst.2014jfst0073>
23. Komatsu, K., Kishitani, T., Sato, M., *et al.*: An appropriate computing system and its system parameters selection based on bottleneck prediction of applications. In: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 768–777. IEEE (2019). <https://doi.org/10.1109/IPDPSW.2019.00127>
24. Komatsu, K., Momose, S., Isobe, Y., *et al.*: Performance evaluation of a vector supercomputer SX-Aurora TSUBASA. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis. pp. 54:1–54:12. SC ’18, IEEE Press (2018). <https://doi.org/10.1109/SC.2018.00057>
25. Liu, Y., Yang, C., Liu, F., *et al.*: 623 Tflop/s HPCG run on Tianhe-2: Leveraging millions of hybrid cores. The International Journal of High Performance Computing Applications 30(1), 39–54 (2016). <https://doi.org/10.1177/1094342015616266>
26. Oh, C.S., Chun, K.C., Byun, Y.Y., *et al.*: 22.1A 1.1V 16GB 640GB/s HBM2E DRAM with a Data-Bus Window-Extension Technique and a Synergetic On-Die ECC Scheme. In: 2020 IEEE International Solid- State Circuits Conference - (ISSCC). pp. 330–332. IEEE (2020). <https://doi.org/10.1109/ISSCC19947.2020.9063110>
27. Onodera, A., Komatsu, K., Fujimoto, S., *et al.*: Optimization of the himeno benchmark for SX-Aurora TSUBASA. In: Wolf, F., Gao, W. (eds.) Benchmarking, Measuring, and Optimizing. Lecture Notes in Computer Science, vol. 12614, pp. 127–143. Springer (2021). https://doi.org/10.1007/978-3-030-71058-3_8
28. Park, J., Smelyanskiy, M., Vaidyanathan, K., *et al.*: Optimizations in a high-performance conjugate gradient benchmark for IA-based multi- and many-core processors. The International Journal of High Performance Computing Applications 30(1), 11–27 (2016). <https://doi.org/10.1177/1094342015593157>
29. Phillips, E., Fatica, M.: Performance analysis of the high-performance conjugate gradient benchmark on GPUs. The International Journal of High Performance Computing Applications 30(1), 28–38 (2016). <https://doi.org/10.1177/1094342015599239>
30. Yamada, Y., Momose, S.: Vector engine processor of NEC’s brand-new supercomputer SX-Aurora TSUBASA. In: International symposium on High Performance Chips (Hot Chips2018) (2018)

Distributed Graph Algorithms for Multiple Vector Engines of NEC SX-Aurora TSUBASA Systems

Ilya V. Afanasyev^{1,2}, Vadim V. Voevodin^{1,2}, Kazuhiko Komatsu³, Hiroaki Kobayashi³

© The Authors 2021. This paper is published with open access at SuperFri.org

This paper describes the world-first attempt to develop distributed graph algorithm implementations, aimed for modern NEC SX-Aurora TSUBASA vector systems. Such systems are equipped with up to eight powerful vector engines, which are capable to significantly accelerate graph processing and simultaneously increase the scale of processed input graphs. This paper describes distributed implementations of three widely-used graph algorithms: Page Rank (PR), Bellman-Ford Single Source Shortest Paths (further referred as SSSP) and Hyperlink-Induced Topic Search (HITS), evaluating their performance and scalability on Aurora 8 system. In this paper we describe graph partitioning strategies, communication strategies, programming models and single-VE optimizations used in these implementations. The developed implementations achieve 40, 6.6 and 1.3 GTEPS performance on PR, SSSP and HITS algorithm on 8 vector engines, at the same time achieving up to 1.5x, 2x and 2.5x acceleration on 2, 4 and 8 vector engines of Aurora 8 systems. Finally, this paper describes an approach to incorporate distributed graph processing support into our previously developed Vector Graph Library (VGL) framework – a novel framework for graph analytics on NEC SX-Aurora TSUBASA architecture.

Keywords: *vector computers, graph algorithms, graph framework, VGL, optimisation.*

Introduction

Developing efficient graph algorithm implementations is an extremely important problem of modern computer science, since graphs are frequently used in various real-world applications, such as social network and web-graph analysis, navigation, and many others. Due to the fact that graph algorithms belong to the data-intensive class (since they typically load from memory a large amount of data, at the same time performing almost no floating point arithmetic), modern architectures with high-bandwidth memory potentially allow solving many graph problems significantly faster compared to modern multicore CPUs. Among other supercomputer architectures, NEC SX-Aurora TSUBASA vector processors [26, 36] are equipped with high-bandwidth memory, what makes them a very promising architecture for graph processing.

In our prior research, we have proposed a Vector Graph Library (VGL) [3, 4, 6] – a novel high-performance graph-processing framework, which is so far the only known graph processing library for the NEC SX-Aurora TSUBASA architecture. As we have previously demonstrated, VGL is capable to outperform both modern Intel multicore CPUs and NVIDIA GPUs on several graph algorithms.

However, at the current moment VGL supports graph processing only within a single vector engine of NEC SX-Aurora TSUBASA system. At the same time, modern NEC SX-Aurora TSUBASA systems, such as the A300-8 (Aurora8) [1], consist of up to 8 vector engines, connected to a single vector host. Such systems are similar to well-known multi-GPU systems, such as DGX or DGX-2, thus in the following paper, we will refer to them as “multi-VE” systems. Implementing graph processing only within a single vector engine has two important drawbacks. First, (1) memory of single vector engine is limited to 48 GB, thus large-scale graphs can not be

¹Moscow Center of Fundamental and Applied Mathematics, Moscow, Russia

²Research Computing Center, Lomonosov Moscow State University, Moscow, Russia

³Tohoku University, Sendai, Miyagi, Japan

processed (without using out-of-core processing techniques, which have been shown to be rather slow on NVIDIA GPUs [15]). Second, (2) eight vector engines of the Aurora8 system working in parallel are capable of significantly accelerating many graph algorithms.

Similar to developing efficient vector graph algorithm implementations for a single vector engine of NEC SX-Aurora TSUBASA, approaches to developing efficient multi-VE implementations are not studied well enough at the moment of this writing. Due to a certain similarity of multi-VE to multi-GPU systems, some existing graph partitioning or inter-GPU communication methods can be used, but this requires a detailed verification due to the differences between vector engines and GPUs. Examples of such differences include using different graph storage formats and optimization techniques within single-GPU or single-VE, different properties of host-device interconnect, different programming models used for developing distributed multi-VE and multi-GPU implementations, and so on.

In this paper, we develop multi-VE implementations of three widely-used graph problems: Page Rank, Single Source Shortest Paths and Hyperlink-Induced Topic Search. We discuss in details which graph partitioning strategies, communication strategies, programming model, and single-VE optimizations have been applied to these implementations in order to achieve good scalability and performance. The performance and scalability of the developed implementations have been evaluated on Aurora 8 system, installed in Tohoku university. Finally, we discuss how programming multi-VE systems can be implemented in VGL framework⁴.

1. NEC SX-Aurora TSUBASA Architecture

NEC has developed vector computing systems called SX series since SX-2 released in 1983 to SX-9 [33], and SX-ACE [13]. So far, many optimizations have been conducted on NEC SX series not only for the benchmark programs [11, 25] but also for various applications such as HPC simulation codes [12, 14, 23, 24] and graph algorithms [5, 7].

The latest vector computer NEC SX-Aurora TSUBASA [26, 36] with dedicated vector processors is the primary target architecture of graph algorithm implementations, proposed in this paper. NEC SX-Aurora TSUBASA has been developed according to the design concepts of vector supercomputer based on the long-term experiences and novel innovations to achieve higher sustained performance and higher usability. Different from the previous generations of the SX vector supercomputer series, the system architecture of SX-Aurora TSUBASA mainly consists of vector engines (VE), equipped with a vector processor and a vector host (VH) of an x86 node.

The VE is used as a primary processor for executing applications, while the VH is used as a secondary processor for managing the VE and executing a basic operating system (OS) functions that are offloaded from the VE.

1.1. Vector Engine

The VE has eight powerful vector cores. As each core provides 537.6 GFlop/s of single-precision performance at 1.40 GHz frequency, the peak performance of the VE reaches 4.3 TFlop/s.

Each SX-Aurora vector core consists of three components: scalar processing unit (SPU), vector processing unit (VPU), and memory subsystem. Most computations are performed by VPUs, while SPUs provide the functionality of a typical CPU. Since SX-Aurora is not just a

⁴VGL is available for free download at vgl.parallel.ru

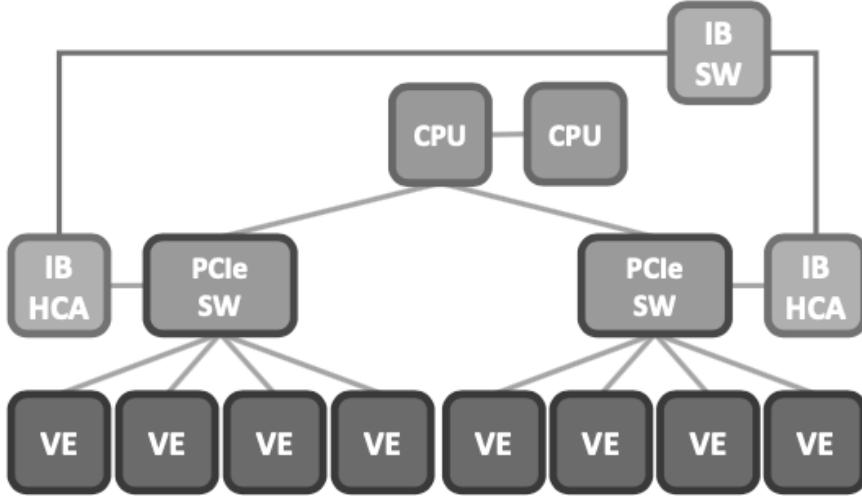


Figure 1. SX-Aurora TSUBASA A300-8

typical accelerator, but rather a self-sufficient processor, SPUs are designed to provide relatively high performance on scalar computations. VPU of each vector core has its own relatively simple instruction pipeline aimed at decoding and reordering vector instructions incoming from SPU. Decoded instructions are executed on vector-parallel pipelines (VPP). In order to store the results of intermediate calculations, each vector core is equipped with 64 vector registers with a total register capacity equal to 128 KB. Each register is designed to store a vector of 256 double-precision elements (DP). On the memory subsystem side, six HBM modules in the vector processor can deliver the 1.22 TB/s memory bandwidth with up to 48 GB total capacity.

1.2. SX-Aurora TSUBASA A300-8 System

Figure 1 is the SX-Aurora TSUBASA A300-8 model. One *VH node* consists of one Vector Host (*VH*) and eight Vector Engines (*VEs*). Four VEs are grouped into a *VE island*. Each VE island and one Infiniband EDR host channel adapter (HCA) is connected to PCI Express switch. Two PCI Express switch is connected into one of CPUs.

As shown in the figure, there are multiple hierarchies of communications. As the communication bandwidth is different among network hierarchies, it is necessary to consider the network hierarchies to exploit the potential of multiple VEs.

1.3. Programming Aurora Systems

Parallel programs for the NEC SX-Aurora Vector Engines are implemented using the OpenMP programming model, while vectorization is performed by the NEC compiler: a developer inserts compiler-specific directives, which help the compiler to perform automatic vectorization. When utilizing multiple Vector Engines of Aurora systems is required, MPI parallelization has to be implemented. At each vector engine an MPI process is started, while data transfers are implemented via MPI send, recv, gather, scatter, bcast and other functions.

2. State of the Art

In this section we will describe previously conducted research, related both to developing distributed graph algorithms and implementing graph algorithms on NEC SX-Aurora TSUBASA vector system.

2.1. Distributed Graph Algorithm Implementations

As already mentioned, NEX SX-Aurora TSUBASA systems equipped with multiple vector engines have multiple similarities to modern multi-GPU system. Developers of Gunrock [35] framework, which targets both single and multi-GPU, have recently published a comprehensive survey on approaches used for developing multi-GPU implementations [32].

There are multiple large scale distributed memory based graph processing systems for CPU clusters, such as GraphX [18], Pregel [28] or Giraph [20]. However, communication models and methods of those systems will very likely be under heavy pressures when local computation is much faster, as in the case when GPU or SX-Aurora vector engines are involved.

Regarding GPU, the most well-known multi-GPU frameworks are Enterprise [27], Medusa [37], Gunrock [35] and NVGRAPH [2]. These frameworks use important implementation techniques, referred through our paper: graph partitioning strategies (how graph vertices and edges are distributed among processors), such as 1D, 2D, Metis [21] partitioning, vertex duplication strategies (how graph remote graph vertices are stored), such as duplicate-1-hop, duplicate-all, and communication strategies (which information to transfer between GPUs), such as broadcast or selective-communicate. In addition, in these framework different programming techniques are used for inter-GPU communication, such as MPI, unified memory, peer-to-peer access.

2.2. VGL (Vector Graph Library) Framework

As previously mentioned in the introduction, we are primary developers of Vector Graph Library (VGL) – a novel graph processing framework, designed to operate on NEC SX-Aurora TSUBASA vector system. A detailed description of VGL is provided in [4]. VGL is designed to implement iterative graph algorithms, and thus uses Bulk synchronous parallel(BSP) [34] model. All graph algorithms in VGL are represented as a sequence of 4 computational abstractions: Advance, Compute, Reduce and Generate New Frontier abstractions (GNF). The Advance abstraction is responsible for processing graph edges, Compute and Reduce – graph vertices, while GNF allows to create working subsets of vertices, which can be processed by other abstractions. These abstractions operate over Graph, Frontier, VerticesArray and EdgesArray data-structures. VGL includes a large variety of optimizations required to operate efficiently within a single vector engine of NEC SX-Aurora TSUBASA, such as parallel workload balancing, using vector instructions with the maximum vector length, improving LLC usage, and many others. In addition, VGL provides an architecture-independent API, which allows it to support computations on different architectures, such as modern NVIDIA GPUs and multicore CPUs without having to change implemented algorithms. This is achieved by object oriented programming, which allows to develop different implementations of each computational abstraction, however, providing identical interfaces for each target architecture [3].

2.3. VGL Graph Storage Format

A central point of VGL framework is an optimized graph storage format, called VectCSR. This format is described in details in [4]. VectCSR is based on a combination of CSR (Compressed Sparse Row) and Sell-C-Sigma [17] formats. In the following paragraph, we will provide a brief description of VectCSR format, since understanding it is frequently required in the following subsections of the paper.

Vect CSR graph storage format scheme is provided in Fig. 3. Its main idea is based on splitting graph vertices into 3 groups based on their outgoing (or incoming) degrees. Vertices with “large degree” ($> 256 * 8$) are processed using all cores of vector engine, each vertex with “medium degree” – using a single vector core, while a group of 256 “small degree” vertices is processed collectively by a single vector core. In addition, edges of “small-degree” vertices are stored in memory in 2 different representations: CSR, which allows to process only a few vertices in sparse algorithms (for example final iterations of BFS), and representation similar to Sell-C-Sigma format, where all edges are reordered in a way shown in Fig. 3. In order to simplify splitting vertices in such groups, as well as to improve memory access pattern for power-law graphs, graph vertices are preliminary sorted based on their degree, and renumbered afterwards.

According to our experience [4], using VectCSR Graph Storage format is mandatory for achieving high performance of graph processing on a single vector engine. This is mostly related to the fact that CSR format does not allow to process graph vertices with a degree lower than vector length efficiently. At the same time VectCSR format provides enough flexibility to allow processing specific subsets of graph vertices and their adjacent edges, which is required in many sparse graph algorithms, such as BFS. Our experiments during VGL development demonstrated that using VectCSR format provides approximately 4–5 times acceleration compared to using traditional CSR format on various synthetic and real-world graphs. This forces us to investigate graph partitioning strategies taking into account the requirement of storing graph in VectCSR-like format within a single vector engine.

3. Evaluating MPI Benchmarks on Aurora8 System

In the beginning of our research, we wanted to estimate transfer speed and scalability of two communication patterns frequently used in graph processing: send/receive point-to-point and all_gather.

3.1. MPI Ping-Pong

MPI ping pong benchmark can be used to evaluate transfer speed of send/receive point-to-point communication. “Ping-pong” benchmark is based on performing a sequence of MPI_Send and MPI_Recv operation between two MPI processes. This communication pattern is used in graph algorithm when MPI processes are doing cycle exchange in order to update some vertices arrays, as will be described further in the paper. We executed “ping-pong” benchmark on MPI processes, attached to different Vector Engines of Aurora8 system (Fig. 2). As shown in Fig. 2, the transfer bandwidth slightly decreases in the case when communicating processes are attached to different switches of aurora8 system (e.g., vector engines 0 and 7). To achieve close to theoretical peak bandwidth values, sending messages of ≥ 16 MB size is needed. Finally, point-to-point communication bandwidth is relatively low due to the fact that different vector

engines are communicating through PCI express, one-directional bandwidth of which does not exceed 12 GB/s. On the contrary, multi-GPU systems equipped with NVLINK 3.0 and A100 GPU are capable of transferring data with 300 GB/s one-directional bandwidth. This means that graph partitioning and communication strategies have to be chosen much more carefully for vector engines, since MPI transfers between different Vector Engines can quickly become a significant bottleneck.

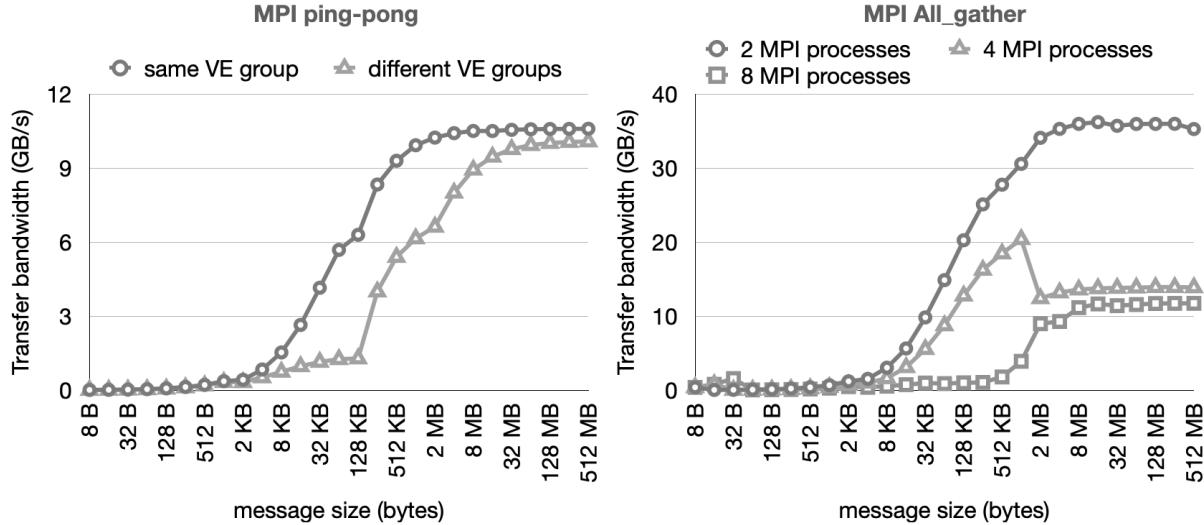


Figure 2. The transfer bandwidth of MPI ping-pong (left) and all_gather (right) benchmarks on Aurora8 system

3.2. MPI_Allgather

Next we evaluated transfer bandwidth of MPI_Allgather operation, which together with MPI_Allgatherv will be frequently used in our implementations. Transfer bandwidth of MPI_Allgather is calculated as $|N|/time$, where $|N|$ is the size of array, which is distributed among $|P|$ MPI processes. As shown in Fig. 2, MPI_Allgather communication between two adjacent vector engines (0/1, 1/2, etc.) can be up to two times faster, compared to four and eight. This is explained by the (1) increase of communication rate (amount of data transferred between all processes) and (2) that while communication of 2 and 4 vector engines is handled by one PCIe SW, but two PCIe SW are used for 8 vector engine communication. In the context of developing graph algorithms this means that the developed implementations will demonstrate the best scaling among two vector engines when MPI_Allgather is used.

4. Implementing Multi-VE Graph Algorithms

4.1. Deciding on Graph Partitioning Strategy

The first important thing to take into an account is how graph vertices and edges need to be partitioned among MPI processes. When selecting graph partitioning strategy, the following factors need to be considered:

1. each MPI process should process approximately equal amount of graph edges;
2. edge processing rate should be approximately equal among different MPI processes;

3. the amount of communications (number of vertices, information about which is required to be sent to other processors) should be minimized;
4. graph processing inside a single vector engine should include optimizations, implemented in VGL (primary VectCSR format should be supported);
5. graph partitioning process should not be very complex and should be vectorizable in order to allow graph partitioning right on vector engines (to avoid exchanging data with vector host).

Thus, we considered three possible graph partitioning strategies between different MPI processes.

Successive distribution of VectCSR edges between MPI processes

First, let us denote the total amount of graph edges as $|E|$, total amount of graph vertices as $|V|$, and total amount of MPI process (and thus vector engines, because each MPI process is bound to a separate vector engine) as $|P|$. The most basic partitioning strategy, which can be applied to VectCSR format is illustrated in Fig. 3 (left), where each MPI process stores a successive region of $|E|/|P|$ edges. This approach has the following **advantages**:

1. such partitioning allows efficient graph processing inside single vector engine, since no re-ordering of vertices data or strided memory accesses to vertices and edges arrays is required (unlike in other approaches, which we will discuss further in the section);
2. each MPI process storing approximately equal amount of edges.

However, this approach also has several crucial **disadvantages**:

1. as shown in our previous paper [4], edge processing rate among different groups of vertices in VectCSR format (“large-degree”, “medium-degree”, “small-degree”) can be different for some algorithms and input graphs. For example, on RMAT [10] graphs and shortest paths algorithm “small-degree” vertices are processed with a rate of 518 GB/s sustained bandwidth rate, while group of “medium-degree” vertices – with 350 GB/s sustained bandwidth rate. This causes uneven load balancing between different vector engines, since each engine processes equal amount of $|E|/|P|$ edges, but with different processing rate.
2. vector extension, required for fast processing of vertices with low-degree, is stored together with CSR representation. Thus several MPI processes (MPI Proc 3 in Fig. 3), which store information about vertices with low degree, are forced to store approximately twice the amount of edges, which leads to a significant difference in memory consumption between different processes.
3. finally, for many scale-free graphs with uneven distribution of vertex degrees, each MPI process does not necessary store $|V|/|P|$ vertices. In the case when communication between processes is based on sending “local” vertices, communication time will be different for each MPI process resulting in a significant imbalance of time spent on communication.

Distribution of VectCSR edges between MPI processes inside separate vertex groups Disadvantages (1) and (2) of previously discussed partitioning strategy can be relatively easily solved by doing partitioning inside each vertex group (“small-degree”, “medium-degree”, etc.) instead of the whole graph, as shown in Fig. 3 (right). According to our experiments, edge processing rates are roughly equal inside different parts of vector groups, while vector extension is distributed among all working processes, instead of only last ones. However, disadvantage (3) still exists, since low rank processes store information about lower number of vertices in scale-free graphs. In order to solve this issue to some extent, we reversed distribution of vertices

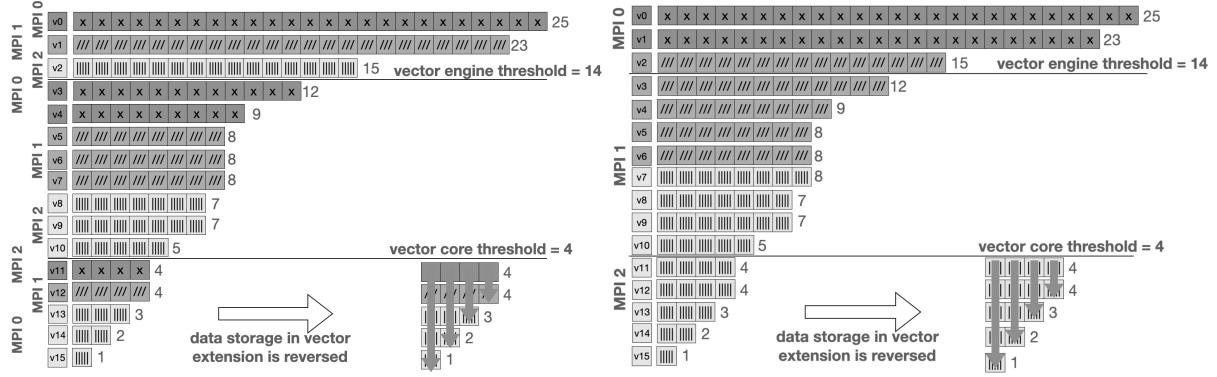


Figure 3. Two strategies of distributing VectCSR graph between MPI processes: splitting VectCSR edges between MPI processes successively and splitting graph edges inside each vector group. Three MPI processes used in this example: MPI process 0 stores graph edges are marked as “*x*”, MPI process 1 – as “*//*”, MPI process 2 – as “*||*”

inside vector core and collective groups, as shown in Fig. 3 (right): low rank processes store high-degree vertices of vector core group, and small-degree vertices of collective group.

Specific distribution of graph vertices between processes Despite previously discussed graph partitioning strategies lack significant disadvantages, strategies which allow better load-balancing and lower amount of communications between MPI processes exist. These strategies are based on distributing graph vertices and edges between different processor according to some vertex partitioning function. However, a more complex graph storage format should be used in order to incorporate VectCSR format into such approaches, due to each vector engine storing non-consecutive vertices. This format (illustrated in Fig. 4) will be further referred to as ShardedCSR format. Vertices in ShardedCSR graph are distributed between different shards (segments) using a specific partitioning function, some of which will be described in the further paragraphs. In our implementation, each vertex belongs only to a single shard, however, this can be relatively easily changed.

Vertex partitioning function is a crucial factor, which defines quality of load-balancing and the amount of communication in ShardedCSR graph. A particularly good strategy, used in Gunrock, is random partitioning of vertices between GPUs (vector engines in our case). Each vertex is assigned with an equal probability to a particular GPU, together with all its neighbouring edges. If the size of processed graph is large, this approach results in $|E|/|P|$ edges and $|V|/|P|$ vertices being stored by each VE.

Another well-known approach is using graph partitioning systems, which are designed to minimize the amount of communications between different MPI processes, such as Metis [21]. Metis provides multilevel recursive-bisection, multilevel k-way, and multi-constraint partitioning schemes. Unfortunately, Metis can not be executed on vector engines, and thus requires doing preliminary graph partitioning on vector host. According to our experiments this approach is rather disadvantageous, since (1) graph partitioning time on CPU is huge and (2) it is accompanied by a large transfer time graph VH to VE through PCIe, which results in very significant pre-processing overheads, in many situations larger than distributed graph processing time. At the same time, previously mentioned methods allow vectorized graph partitioning and pre-processing directly on vector engines, which allows them to significantly outperform Metis-based approach.

Conclusions Thus, further in the paper we will use two **most promising** graph partitioning strategies: (1) Distribution of VectCSR edges between vector engines inside separate vertex groups and (2) random distribution of graph vertices between vector engines.

4.2. Deciding on Communication Strategy

Communication strategy is another important trait of distributed graph algorithm implementations. All the discussed graph partitioning strategies are based on splitting graph edges between MPI processes to make sure that all the required for the computations edges are stored locally (on the process itself). Thus, MPI processes need to exchange **only information about vertices** (for example current distances in shortest paths, rank values in page rank, etc). Information about graph vertices can be exchanged in several different ways, depending on the properties of algorithm and graph partitioning schemes. The comparative characteristics of multiple communication strategies, such as additional space required for communication buffers, communication cost (the amount of data transferred between all processes) and pre-/post-processing complexity are provided in Tab. 1. Pre-/post-processing complexity is the total number of operations, required to prepare data for MPI communication (for example, copy information about scattered vertices into exchange buffers), and to receive it afterwards. The communication cost provided in Tab. 1 takes into account only the amount of data exchanged, however, the actual communication time additionally depends on the performance of MPI communication functions, shown in Fig. 2.

Table 1. The comparative characteristics of communication strategies. $|V|$ – number of vertices in graph, $|N|$ – number of MPI processes, $|K_i|$ – number of updated vertices on each MPI process

Strategy	Additional space for communication buffers	Communication cost	Pre-/post-processing complexity
(1)all-to-all (bcast based)	$ V * N $	$ V * N ^2$	$ V * N $
(2)all-to-all (cycle exchange based)	$ V $	$ V * N * \log N $	$ V * \log N $
(3)all-to-all recently changed only	$ V $	$(\sum_{i=1}^{ N } K_i) * N * \log N $	$(\sum_{i=1}^{ N } K_i) * \log N $
(4)all_gather	0	$ V * N $	0
(5)all_gather recently changed only	$ V $	$(\sum_{i=1}^{ N } K_i) * N $	$K_i + \sum_{i=1}^{ N } K_i$

All-to-all This communication scheme involves each MPI process broadcasting vertices array of size $|V|$ to all other processes. This can be achieved by using 2 communication algorithms. (1) Each MPI process broadcasts information about $|V|$ graph vertices to all other processes, while receiving information about $|V| * |N|$ vertices. After communication each process updates its private vertices arrays using provided in the algorithm update criteria, such as computing minimal distances to each vertex or calculating a sum of ranks for each vertex, obtained on different processes. In Tab. 1 this strategy is referred to as “all-to-all bcast based”. Another

possible strategy (2) requires each MPI process to send information about $|V|$ graph vertices to $rank+1$ process and simultaneously receive information about $|V|$ vertices from $rank-1$ process. After communication, each process updates local vertices arrays using update criteria and the received information. After that, newly calculated information is sent to $rank+2$ process, and so on using $\log|N|$ steps. In Tab. 1 this strategy is referred to as **“all-to-all cycle exchange based”**. This approach requires significantly less additional space and has lower communication cost compared to “all-to-all bcast based” strategy (1).

All-to-all recently changed only This strategy is based on slightly modified all-to-all strategy. However, information only about $|K|$ vertices, values of which have been changed during the last algorithm iteration (for example only recently updated distances or ranks) is sent to other processes. This reduces the amount of transferred data depending on the algorithm properties, however, at the cost of increased pre-processing and post-processing, since each process is now required to generate lists of recently updated vertices before communication (using `copy_if` or parallel prefix sum algorithm), and afterwards to place received data to the correct places of local vertices arrays using scatter operation. Similarly to “all-to-all” strategy, cycle exchange communication pattern can be used instead of broadcast here.

All_gather Two previously discussed strategies can be used both for pull- and push-based [9] algorithms. In pull-based algorithms each MPI process updates only its locally stored vertices, while for push-based algorithm any graph vertices can be updated. Since pull-based algorithms update only their local vertices during computations, it is possible to use communication strategy based on `MPI_all_gather` operation, when each process broadcasts only its local vertices.

All_gather recently changed only This strategy is aimed to further reduce communication cost of all_gather strategy, similar to “all-to-all recently changed only”. Each process generates a list of recently updated local vertices, and then uses `all_gatherv` operation to communicate lists of potentially smaller size, but, similarly, at the cost of additional pre- and post-processing.

Conclusions Table 1 demonstrates that each strategy can potentially be useful for different graph algorithms, due different strategies having different trade-offs: larger communicating cost or pre-/post-processing complexity, etc. In addition, all these strategies can be implemented on vector engines, since they do not require complex data structure and support all the required optimizations, applied withing a single VE. In the following section we implemented all these strategies for single source shortest paths algorithm, comparing execution time, performance and scalability of each approach, and afterwards selecting the most fitting approach for NEC SX-Aurora TSUBASA architecture.

4.3. Bellman-Ford Shortest Paths Algorithm

The single-source shortest paths problem involves finding paths between a given source vertex and all other graph vertices, such that all weights on the path between source and destination vertices are minimized. Multiple parallel shortest paths algorithms exist, including delta-stepping [29] and Bellman-Ford [16], the latter being implemented in VGL, and thus being a subject of our research. Two different variations of Bellman-Ford algorithm exist: push-based and pull-based. Pull-based variation updates distance to each vertex based on the distances to vertices, connected via incoming edges to the processed vertex. Push-based algorithm propagates distance of current vertex to its neighbouring vertices via outgoing edges. Both these variations are suitable for implementation within a single vector engine [4].

We implemented the following variations of shortest paths algorithms based on vectCSR partitioning inside separate vector groups:

1. push-based algorithm using “all-to-all” communication strategy;
2. push-based algorithm using “all-to-all recently-changed only” communication strategy;
3. pull-based algorithm using “all-to-all recently-changed only” communication strategy;
4. pull-based algorithm using “all-gather” communication strategy;
5. pull-based algorithm using “all-to-all recently-changed only” communication strategy.

In addition, we implemented two variations of algorithm based on ShardedCSR graph storage format and random vertex partitioning:

1. pull-based algorithm using “all-gather” communication strategy;
2. push-based algorithm using “all-to-all recently-changed only” communication strategy.

Scalability of these implementations is evaluated in Fig. 4. Here (as well as in the following section) synthetic RMAT [10] of scale 25 with edge factor 32 are used as input data. Thus such graph contains 33 million vertices and 1 billion edges. RMAT graphs are scale-free, which means that they have power law distribution of vertex degrees. Thus, they successfully model social networks and web graphs, which are used in various application fields. As could be expected, scalability of all-to-all implementations is very limited due to a very large amount of transferred data coupled with low bandwidth of PCIe interconnect.

The highest performance and best scalability has been achieved when using VectCSR partitioning inside separate vertices groups together with “push all to all recently changed” and “pull all gather” communication policies. However, scalability of both these implementations is still far away from the linear: only 2 and 2.6 times acceleration is achieved when using 8 vector engines. In order to further investigate these issues, we have collected profiling data, which is provided in Tab. 2 and Tab. 3. Table 2 provides information about main computational components of the developed implementations, such as execution time spent inside Advance (which processes graph edges) and Compute (which processes graph vertices) VGL abstractions, MPI pre-process and post-process functions (when sent data is prepared and received data is analysed), and MPI communication time (of MPI_send, MPI_recv, MPI_allgather, MPI_bcast functions). Table 2 demonstrates that while time, spent on processing of graph edges (Advance) scales nearly linearly, the primary reason of limited scalability of both implementations is the increase of MPI communications and MPI pre-/post-processing time. While on single VE no time is required on these activities, 16 % of program execution time is spent on communication and pre-/post-processing on 2 VE, 42 % on 4 VE and 61 % on 8 VE for push-based algorithm. Similar situation can be observed on pull-based algorithm. As demonstrated in Tab. 3, time spent on communications is roughly equal between different MPI processes.

On the final note, we would like to discuss reasons standing behind poor scalability of implementations, based on ShardedCSR format. Despite the fact it provides more equal distribution of graph edges and vertices between different processes, this format requires reordering of vertices arrays of size $|V|$, after obtaining information from other processes, since vertices inside each sharded can be sorted differently to support VectCSR format inside each shard. Despite the fact that this reordering has $O|V|$ complexity, its efficiency is lower compared to edge processing in advance, which has $O(|E|/|N|)$ complexity. With the increase of number of MPI processes used ($|N|$), and taking into the account that for many real-world graphs $|E| = C * |V|$, with $|C|$ being a constant in 8–64 range, such reordering has a comparable time with Advance execution time, and the scalability of ShardedCSR implementaion being limited by Amdahl’s law [19]. A

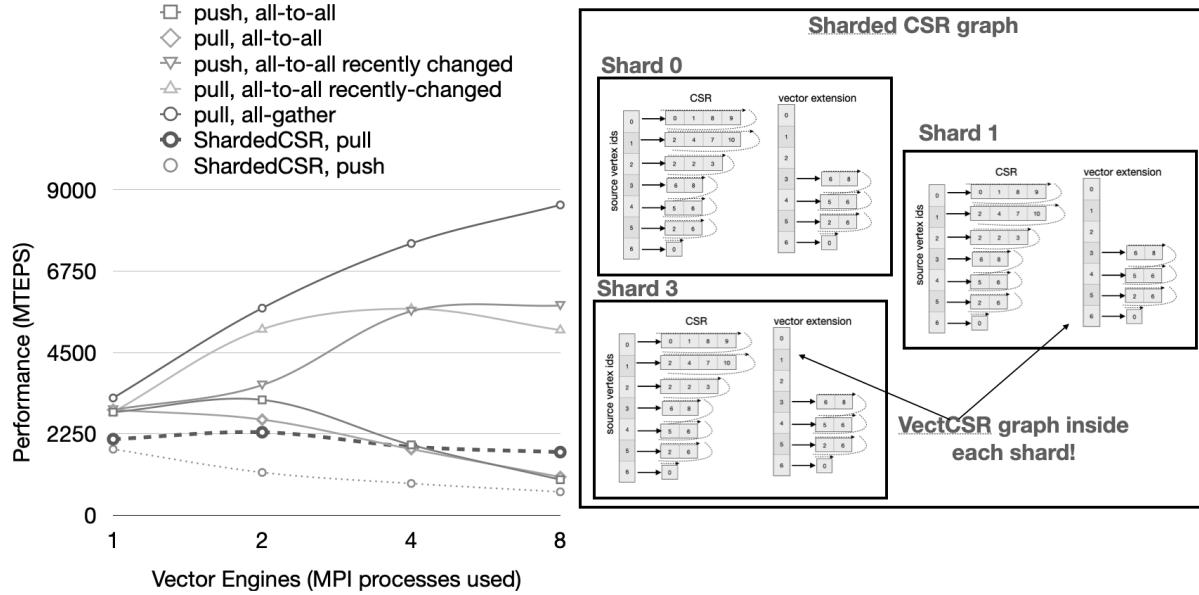


Figure 4. Scalability of the shortest paths algorithms (left), ShardedCSR graph storage format scheme (right). ShardedCSR graph consists of different shards (0, 1, 2 in this example). Each shard is a subgraph in VectCSR format, demonstrated in details in Fig. 3, which includes CSR and vector extension

Table 2. Main computational components of shortest paths implementations on RMAT graph of scale 25, multiple algorithm iterations

Activity	SSSP, push, 1 VE	SSSP, push, 2 VE	SSSP, push, 4 VE	SSSP, push, 8 VE	SSSP, pull, 1 VE	SSSP, pull, 2 VE	SSSP, pull, 4 VE	SSSP, pull, 8 VE
Advance	3540 s	1999 s	1122 s	602 s	3795 s	1467 s	737 s	387 s
Compute	23 s	31 s	29 s	32 s	28 s	30 s	32 s	32 s
MPI communication	-	81 s	224 s	139 s	-	294 s	407 s	774 s
MPI preprocess and postprocess	-	324 s	598 s	834 s	-	40 s	17 s	21 s

possible solution to this problem is using CSR format instead of VectCSR inside each shard, which removes the requirement of reordering vertices after each communication, however, these increase Advance time in 3–4 times for various graph algorithms, which is an unacceptable trade off.

4.4. Page Rank Algorithm

The page rank [31] algorithm assigns a numerical weighting to each element of a hyper-linked set of documents, (for example, web-graph), with the purpose of quantifying its relative importance within the set. Similarly to shortest paths, page rank algorithm has pull-based or push-based variations. Push-based algorithm requires using atomicAdd operations (since 2 vertices processed in parallel can possibly try to update the same adjacent vertex). Despite vector

Table 3. Profiling of shortest paths implementations using Ftrace tool

Algorithm and MPI rank	ELAPSED time	COMM TIME	COMM TIME /ELAPSED time	AVER.LEN
sssp pull, rank 0	2.229 s	1.392 s	62 %	4 MB
sssp pull, rank 1	2.227 s	1.300 s	58 %	4.1 MB
sssp pull, rank 2	2.227 s	1.047 s	47 %	4.5 M
sssp pull, rank 3	2.226 s	0.423 s	19 %	7 MB
sssp push, rank 0	2.039 s	0.340 s	16 %	3.1 MB
sssp push, rank 1	2.037 s	0.373 s	12 %	3.1 MB
sssp push, rank 2	2.037 s	0.333 s	11 %	3.0 MB
sssp push, rank 3	2.036 s	0.248 s	6 %	3.0 MB

engines having support of atomic operations, code including them can not be vectorized, thus pull-based variation must be used on NEC SX-Aurora TSUBASA.

Based on the research provided for shortest paths algorithm, for page rank algorithm we implemented vectCSR partitioning inside separate vector groups and all_gather communication model. Scalability of the developed implementation is provided in Section 5.

4.5. HITS

Hyperlink-Induced Topic Search (HITS) is also a link analysis algorithm that rates Web pages [22], however, quite differently compared to page rank. The main idea of HITS algorithm is based on each graph vertex having authority and hub scores. Both scores are consequently updated on each iteration; authority score of each node is updated based on incoming edges, while hub score – based on outgoing. This means that when using VectCSR graph storage format, two reorderings of vertices arrays are required on each iteration, when traversal direction is changed. According to the profiling data the reordering process on large graphs (vertices arrays of which do not fit into LLC cache) take up to 45 % of program execution time, while the remaining 55 % are spent on Advance (processing edges), Compute and Reduce (communications are excluded). At the same time increasing the number of vector engines reduces Advance time linearly, but does not reduce reordering time at all, since each process is required to reorder $|V|$ vertices no matter how many vector engines are used. Thus scalability of such implementation is limited due to Amdahl's law, similarly to shortest paths implementation when using shardedCSR graph partitioning. Scalability of the developed HITS implementation is provided in Fig. 5 and Tab. 4.

The provided analysis allows to make a conclusion that VectCSR-based format is not suitable for distributed graph processing when algorithm frequently switches traversal direction, such as HITS or direction-optimizing BFS [8].

4.6. Implementing Distributed Graph Processing Support in VGL

The provided research allowed us to include support of distributed graph processing inside VGL⁵. This has been achieved by implementing a special method `exchange_vertices_array`, which is aimed to update vertices on each vector engine according to remote data and is called after each graph algorithm iteration. In addition, we slightly modified `VectCSR` and `ShardedCSR` graph storage formats in order to support distributed graph storage, as well as inner representation of `Advance` abstraction. However, since all computational and data abstractions have their interfaces unchanged, many implemented in VGL algorithms can be turned into distributed versions by simply adding `exchange_vertices_array` calls into the correct places.

5. Evaluating Scalability of All Developed Implementations

Figure 5 and Tab. 4 demonstrate scalability of the developed implementations of page rank, HITS and shortest paths (pull and push) algorithms. The performance comparison of VGL-based implementations working on a single vector engine with their GPU and Intel CPU counterparts can be found in [4] or at VGL website⁶ in the “performance” section. In general, for SSSP and PR algorithms VGL-based implementations are up to 14 times faster compared to Ligra, Galois and GAPBS multicore CPU frameworks and libraries, and up to 3 times faster compared to Gunrock and NVGRAPH implementations for various synthetic and real-world graphs [4]. Thus, in this section we will only evaluate MPI scalability of the developed implementations.

During these experiments we used 1, 2, 4 and 8 vector engines of Aurora 8 systems. When using 2 and 4 vector engines, MPI processes were bound to the adjacent vector engines in the same PCIe SW group. The main performance characteristics used is TEPS (Traversed Edges Per Second) [30], equal to the amount of graph edges processed divided by algorithm execution time. For page rank and HITS algorithms the performance is calculated for a single iteration.

Table 4. Scalability of all the developed distributed graph algorithm implementations

Algorithm	1 VE	2 VE	4 VE	8 VE
page rank	1	1.53	2.0	2.38
sssp (push)	1	1.41	1.75	1.82
sssp (pull)	1	1.74	1.97	2.34
hits	1	0.89	1.00	1.07

Finally, we can compare scalability of our implementations with multi-GPU scalability of Gunrock [35]. According to the research [32] on SSSP problem Gunrock achieves 1.5 and 2.92 times acceleration when using 2 GPUs and 6 GPUs, while on PR Gunrock achieves 2.8 and 4.1 times acceleration on 2 and 6 GPU. Comparing these values to Tab. 4 indicates that VGL achieves lower scalability, which is explained by differences in bandwidth of interconnect used in modern GPUs (NVLINK, 80–300 GB/s) and NEC SX-Aurora TSUBASA (PCIe, 12 GB/s).

⁵Distributed graph processing is currently available in VGL on NEC-MPI git branch.

⁶<https://vgl.parallel.ru/performance.html>

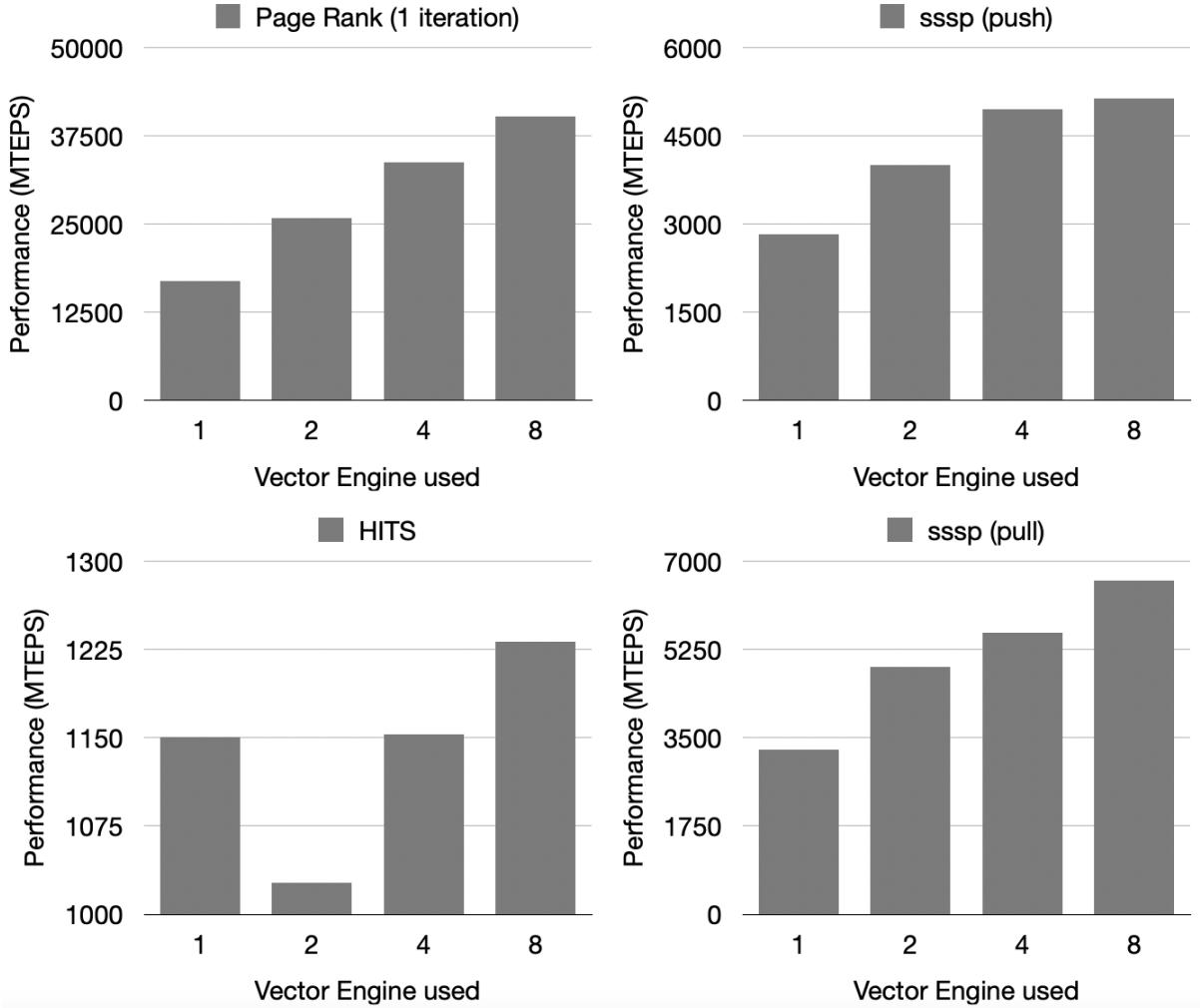


Figure 5. Scalability of the developed distributed implementations of page rank, HITS and shortest paths (pull and push) graph algorithms

We hope that future generations of NEC SX-Aurora TSUBASA architecture will be based on higher bandwidth interconnect, which can greatly improve scalability of our implementations.

Conclusion and Future Plans

In this paper we have proposed world first distributed graph algorithms for modern NEC SX-Aurora TSUBASA systems. In this paper we have discussed multiple attributes of these implementations: graph storage format, graph partitioning schemes, communication strategies, optimizations within single vector engine. Our experiments demonstrated that partitioning of VectCSR graph storage format inside separate vertex groups coupled with all_gather and all-to-all recently changed only strategies communication provide the best scalability. Our implementations achieve 40, 6.6 and 1.3 GTEPS performance on PR, SSSP and HITS algorithm on 8 vector engines, at the same time achieving up to 1.5x, 2x and 2.5x acceleration on 2, 4 and 8 vector engines of Aurora 8 systems. In addition, the developed distributed implementations allowed us to process 8 times larger graphs using Aurora 8 system at the same time obtaining reasonable (up to 2.6 times acceleration), which is crucial in many real-world applications.

Finally, using the example of HITs algorithm we have demonstrated that algorithms, which require switching between push and pull traversal on each iteration do not scale well with the proposed approaches on Aurora systems. Solving this problem is an important direction of our future work. Our other future plans include developing distributed versions of graph algorithms, which require working with sparse frontiers of active vertices, such as breadth-first search.

Acknowledgements

The research is carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University and the computational resources of Cyberscience Center at Tohoku University.

The reported study presented in all sections excluding 5 was funded by RFBR and JSPS according to the research project No. 21-57-50002 and Grant number JPJSBP120214801. The work presented in Section 5 is supported by Russian Ministry of Science and Higher Education, agreement No. 075-15-2019-1621.

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. NEC SX-Aurora TSUBASA A300-8. <https://www.nec.com/en/global/solutions/hpc/sx/A300-8.html>, accessed: 2021-04-06
2. NVGRAPH. <https://developer.nvidia.com/nvgraph>, accessed: 2021-06-28
3. Afanasyev, I.V.: Developing an architecture-independent graph framework for modern vector processors and NVIDIA GPUs. *Supercomputing Frontiers and Innovations* 7(4), 49–61 (2021). <https://doi.org/10.14529/jsfi200404>
4. Afanasyev, I.V., Voevodin, V.V., Komatsu, K., Kobayashi, H.: VGL: a high-performance graph processing framework for the NEC SX-Aurora TSUBASA vector architecture. *The Journal of Supercomputing* 77(8), 8694–8715 (2021). <https://doi.org/10.1007/s11227-020-03564-9>
5. Afanasyev, I.V., Voevodin, V.V., Komatsu, K., Kobayashi, H., et al.: Developing efficient implementations of Bellman–Ford and Forward-Backward graph algorithms for NEC SX-ACE. *Supercomputing Frontiers and Innovations* 5(3), 65–69 (2018). <https://doi.org/10.14529/jsfi180311>
6. Afanasyev, I.V., Voevodin, V.V., Komatsu, K., Kobayashi, H., et al.: Analysis of relationship between SIMD-processing features used in NVIDIA GPUs and NEC SX-Aurora TSUBASA vector processors. In: International Conference on Parallel Computing Technologies. Lecture Notes in Computer Science, vol. 11657, pp. 125–139. Springer (2019). https://doi.org/10.1007/978-3-030-25636-4_10
7. Afanasyev, I.V., Voevodin, V.V., Komatsu, K., Kobayashi, H., et al.: Developing efficient implementations of shortest paths and page rank algorithms for NEC SX-Aurora TSUBASA

- architecture. Lobachevskii Journal of Mathematics 40(11), 1753–1762 (2019). <https://doi.org/10.1134/S1995080219110039>
8. Beamer, S., Asanović, K., Patterson, D.: Direction-optimizing breadth-first search. Scientific Programming 21(3-4), 137–148 (2013). <https://doi.org/10.1109/SC.2012.50>
 9. Besta, M., Podstawski, M., Groner, L., *et al.*: To push or to pull: On reducing communication and synchronization in graph computations. In: Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing. pp. 93–104. ACM (2017). <https://doi.org/10.1145/3078597.3078616>
 10. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: A recursive model for graph mining. In: Proceedings of the 2004 SIAM International Conference on Data Mining. pp. 442–446. SIAM (2004). <https://doi.org/10.1137/1.9781611972740.43>
 11. Egawa, R., Komatsu, K., Isobe, Y., *et al.*: Performance and power analysis of SX-ACE using HP-X benchmark programs. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER). pp. 693–700. IEEE Computer Society (2017). <https://doi.org/10.1109/CLUSTER.2017.65>
 12. Egawa, R., Komatsu, K., Kobayashi, H.: Designing an HPC refactoring catalog toward the exa-scale computing era. In: Resch, M.M., Bez, W., Focht, E., Kobayashi, H., Patel, N. (eds.) Sustained Simulation Performance 2014. pp. 91–98. Springer (2015). https://doi.org/10.1007/978-3-319-10626-7_8
 13. Egawa, R., Komatsu, K., Kobayashi, H., *et al.*: Potential of a modern vector supercomputer for practical applications: performance evaluation of SX-ACE. The Journal of Supercomputing 73(9), 3948–3976 (2017). <https://doi.org/10.1007/s11227-017-1993-y>
 14. Egawa, R., Komatsu, K., Takizawa, H.: Designing an open database of system-aware code optimizations. In: 2017 Fifth International Symposium on Computing and Networking (CANDAR). pp. 369–374. IEEE Computer Society (2017). <https://doi.org/10.1109/CANDAR.2017.102>
 15. Gharaibeh, A., Reza, T., Santos-Neto, E., *et al.*: Efficient large-scale graph processing on hybrid CPU and GPU systems. arXiv preprint arXiv:1312.3018 (2013)
 16. Goldberg, A., Radzik, T.: A heuristic improvement of the Bellman-Ford algorithm. Tech. rep., Stanford Univ CA Dept of Computer Science (1993)
 17. Gómez, C., Casas, M., Mantovani, F., Focht, E.: Optimizing sparse matrix-vector multiplication in NEC SX-Aurora Vector Engine. Tech. rep., Technical Report, Barcelona Supercomputing Center (2020)
 18. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: Graphx: Graph processing in a distributed dataflow framework. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). pp. 599–613 (2014)
 19. Gustafson, J.L.: Reevaluating Amdahl’s law. Communications of the ACM 31(5), 532–533 (1988). <https://doi.org/10.1145/42411.42415>

20. Han, M., Daudjee, K.: Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment* 8(9), 950–961 (2015). <https://doi.org/10.14778/2777598.2777604>
21. Karypis, G., Kumar, V.: Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices (1997), <https://hdl.handle.net/11299/215346>
22. Kleinberg, J.M., Kumar, R., Raghavan, P., *et al.*: The web as a graph: Measurements, models, and methods. In: International Computing and Combinatorics Conference. Lecture Notes in Computer Science, vol. 1627, pp. 1–17. Springer (1999). https://doi.org/10.1007/3-540-48686-0_1
23. Komatsu, K., Egawa, R., Hirasawa, S., Takizawa, H., Itakura, K., Kobayashi, H.: Migration of an atmospheric simulation code to an OpenACC platform using the Xevolver framework. In: 2015 Third International Symposium on Computing and Networking (CANDAR). pp. 515–520. IEEE Computer Society (2015). <https://doi.org/10.1109/CANDAR.2015.102>
24. Komatsu, K., Egawa, R., Hirasawa, S., Takizawa, H., Itakura, K., Kobayashi, H.: Translation of large-scale simulation codes for an OpenACC platform using the Xevolver framework. *International Journal of Networking and Computing* 6(2), 167–180 (2016). https://doi.org/10.15803/ijnc.6.2_167
25. Komatsu, K., Egawa, R., Isobe, Y., *et al.*: An approach to the highest efficiency of the HPCG benchmark on the SX-ACE supercomputer. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC15), Poster. pp. 1–2 (2015)
26. Komatsu, K., Watanabe, O., Musa, A., *et al.*: Performance evaluation of a vector supercomputer SX-Aurora TSUBASA. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, Dallas, TX, USA, Nov. 11-16, 2018. pp. 54:1–54:12. SC ’18, IEEE (2018). <https://doi.org/10.1109/SC.2018.00057>
27. Liu, H., Huang, H.H.: Enterprise: breadth-first graph traversal on GPUs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12. ACM (2015). <https://doi.org/10.1145/2807591.2807594>
28. Malewicz, G., Austern, M.H., Bik, A.J., *et al.*: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. pp. 135–146. ACM (2010). <https://doi.org/10.1145/1807167.1807184>
29. Meyer, U., Sanders, P.: δ -stepping: a parallelizable shortest path algorithm. *Journal of Algorithms* 49(1), 114–152 (2003). [https://doi.org/10.1016/S0196-6774\(03\)00076-2](https://doi.org/10.1016/S0196-6774(03)00076-2)
30. Murphy, R.C., Wheeler, K.B., Barrett, B.W., Ang, J.A.: Introducing the graph 500. Cray Users Group (CUG) 19, 45–74 (2010)
31. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Tech. rep., Stanford InfoLab (1999)
32. Pan, Y.: Multi-GPU Graph Processing. Ph.D. thesis, University of California, Davis (2019)

33. Soga, T., Musa, A., Okabe, K., *et al.*: Performance of SOR methods on modern vector and scalar processors. *Computers & Fluids* 45(1), 215–221 (2011). <https://doi.org/10.1016/j.compfluid.2010.12.024>
34. Tiskin, A.: The design and analysis of bulk-synchronous parallel algorithms. Ph.D. thesis, Citeseer (1998)
35. Wang, Y., Davidson, A., Pan, Y., *et al.*: Gunrock: A high-performance graph processing library on the GPU. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 1–12. ACM (2016). <https://doi.org/10.1145/2851141.2851145>
36. Yamada, Y., Momose, S.: Vector engine processor of NEC brand-new supercomputer SX-Aurora TSUBASA. In: International symposium on High Performance Chips (Hot Chips2018) (2018)
37. Zhong, J., He, B.: Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25(6), 1543–1552 (2013). <https://doi.org/10.1109/TPDS.2013.111>

Optimizing Load Balance in a Parallel CFD Code for a Large-scale Turbine Simulation on a Vector Supercomputer

*Osamu Watanabe¹, Kazuhiko Komatsu², Masayuki Sato²,
Hiroaki Kobayashi²*

© The Authors 2021. This paper is published with open access at SuperFri.org

A turbine for power generation is one of the essential infrastructures in our society. A turbine's failure causes severe social and economic impacts on our everyday life. Therefore, it is necessary to foresee such failures in advance. However, it is not easy to expect these failures from a real turbine. Hence, it is required to simulate various events occurring in the turbine by numerical simulations of the turbine. A multiphysics CFD code, "Numerical Turbine," has been developed on vector supercomputer systems for large-scale simulations of unsteady wet steam flows inside a turbine. To solve this problem, the Numerical Turbine code is a block structure code using MPI parallelization, and the calculation space consists of grid blocks of different sizes. Therefore, load imbalance occurs when executing the code in MPI parallelization. This paper creates an estimation model that finds the calculation time from each grid block's calculation amount and calculation performance. It proposes an OpenMP parallelization method for the load balance of MPI applications. This proposed method reduces the load imbalance by considering the vector performance according to the calculation amount based on the model. Moreover, this proposed method recognizes the need to reduce the load imbalance without pre-execution. The performance evaluation shows that the proposed method improves the load balance from 24.4 % to 9.3 %.

Keywords: *turbine simulation code, MPI, OpenMP, hybrid parallelization, vector supercomputer, load balance.*

Introduction

Thanks to advances in large-scale simulations, various phenomena in the real world can be reproduced more realistically by using supercomputer systems. On the other hand, there are still many problems with social infrastructures to be solved in the real world, and the impact of these issues on our society is immeasurable [1]. Therefore, there is no doubt that preventing these problems is beneficial for promoting a safe society. For example, gas and steam turbines are used to generate thermal power. Their failures will have a severe social and economic impact. Therefore, it is necessary to foresee such failures in advance. However, it is difficult to predict these failures from a real turbine [10]. Consequently, it is needed to simulate various phenomena occurring in the turbine by a numerical simulation using a computer to predict the failures. Moreover, the use of supercomputers is indispensable for simulating complex and diverse phenomena that occur in turbines.

Internally, these turbines consist of multiple stages of stator cascades and rotor cascades, and the total number of blades exceeds one thousand. It is difficult from the expense and time to realize the designs of these turbines in the short term in practical ways. To design highly reliable advanced gas turbines and steam turbines, it is necessary to concurrently solve various physical phenomena that occur in parallel with the heat flow phenomena (e.g., adhesion of fine particles to a blade, condensation of water vapor, erosion due to collision of large droplets with a blade, melting of a blade by high-temperature thermal fluid, corrosion of a blade by supercritical water, etc.). Therefore, to design a highly efficient and reliable turbine, it is necessary to develop a multiphysics CFD technology capable of numerically analyzing mathematical models simulating

¹NEC Corporation, Tokyo, Japan

²Tohoku University, Sendai, Japan

this multiphysics in conjunction with governing equations of thermal flows [11, 19]. However, mutual interference of multiphysics in the turbine is due to problematic mutual interference with all heat flow fields in the turbine. For this reason, it is complicated to solve this multiphysics without thoroughly analyzing the total internal heat flow of the turbine simultaneously.

A multiphysics CFD code, “Numerical Turbine,” has been developed for large-scale simulations of unsteady wet steam flow with non-equilibrium condensation inside a turbine [14]. The Numerical Turbine code has been developed on vector supercomputer systems [3, 4, 9], and the code has been optimized so that high computing performance can be obtained with vector supercomputers [7, 20]. The code can be used to analyze the unsteady wet steam flow in the final multi-stage cascade of a real steam turbine. The code also applies numerical solutions for analyzing the complex thermal flow generated inside the final stage of a steam turbine. The code incorporating these mathematical models can numerically elucidate the multiphysics interaction of the thermal flow inside the turbine. It is possible to determine in advance a catastrophic situation leading to turbine instability and blade destruction.

To accurately simulate such various phenomena into the turbine, it is essential to do a whole circumference simulation that analyzes the entire turbine. The number of computational grids to be calculated exceeds 500 million. The amount of calculation and memory used is enormous in doing the full annulus simulation. Moreover, to use analysis results practically, it is necessary to complete calculations within a required time.

The Numerical Turbine code is a block-structured CFD code, and the domain decomposition is chosen for the MPI parallelization of the Numerical Turbine code. Each MPI process is in charge of one or more grid blocks exchanging information with the neighbor ones. The difference in the number of grid points of the grid blocks causes the difference in the calculation amount of the MPI process of each grid block. Consequently, the calculation time of these MPI processes is also different. Thus, executing this code in MPI parallelization causes a load imbalance.

An MPI process with a short calculation time needs to wait for other MPI processes with a long calculation time in such a situation. Additionally, an increase in the number of MPI parallels with load imbalance adversely affects scalability. This situation indicates that the computational resources of the supercomputer are not being used effectively.

Therefore, to execute the Numerical Turbine code on a vector computer, this paper creates an estimation model that finds the calculation time from each grid block’s calculation amount and calculation performance. It proposes a method for reducing the load imbalance by considering the vector performance according to the calculation amount based on the model. For parallelization of the grid blocks, thread parallelization using OpenMP is applied, and load balance is improved by hybrid parallelization. However, in the vector architecture, the length of the vectorized loop has a significant effect on the performance. Because of this, the vector performance according to the calculation amount needs to be considered for improving the load imbalance. Moreover, this proposed method recognizes the need to reduce the load imbalance without pre-execution.

The outline of this paper is as follows. In Section 1, this paper gives an overview of the Numerical Turbine code and characteristics of the code in MPI parallel execution. This paper proposes a method for reducing load imbalance using hybrid parallelization in Section 2. In Section 3, this paper discusses the performance results and concludes the paper and future work in Section 3.3.

1. Numerical Turbine Code

The Numerical Turbine code can simulate unsteady flows with wetness and shocks in actual gas and steam turbines and perform full annulus simulations of the turbines to reproduce unsteady wet-steam moist-air flows in these turbines.

1.1. Numerical Procedure

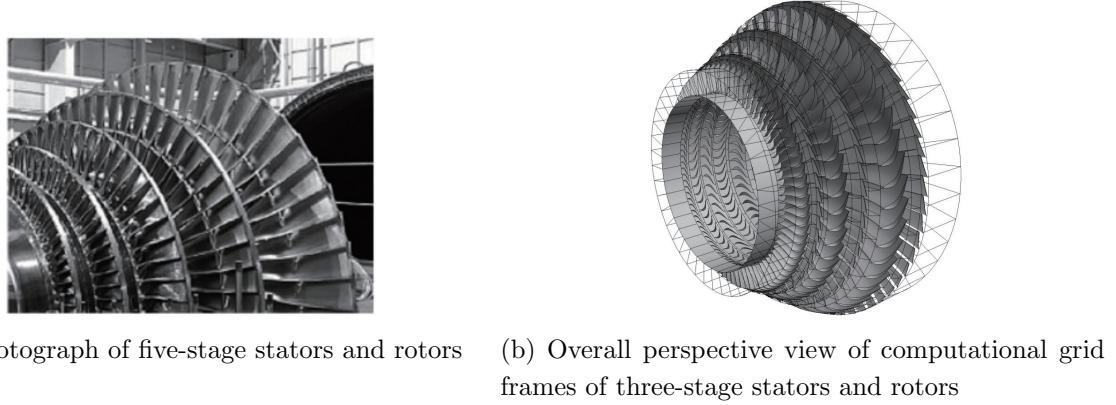
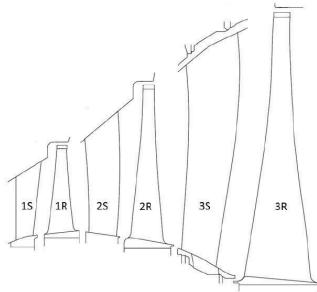
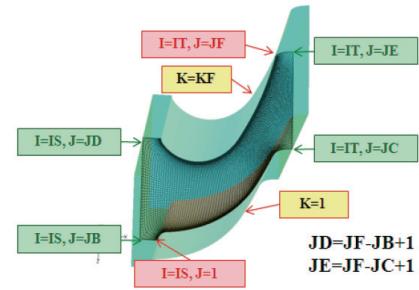
The Numerical Turbine code solves equations consisting of a mass conservation equation of steam considering momentum phase change, a momentum conservation equation, an energy conservation equation, a droplet mass conservation equation, a droplet number density conservation equation, and an equation of turbulence kinetic energy along with its specific dissipation rate as a fundamental equation of compressible flow with condensation. It is assumed that the gas-liquid two-phase flow is a homogeneous flow with a sufficiently small mass fraction of droplets. The equation of the state of wet steam and the equation of the speed of sound are calculated from the equation formulated by Ishizaka et al. [8]. The mass production rate of liquid droplets by condensation is expressed by the sum of condensation nucleation and mass growth by droplet growth based on classical condensation theory. In the Numerical Turbine, the droplet growth is further approximated by the equation with the number density of droplets as a function [8]. The condensation nucleation rate is calculated from the equation of Frenkel [12], and the growth rate of droplets is calculated from the model of Gyarmathy [6]. As the numerical solution, Roe's flux difference separation method [17] and the fourth-order compact MUSCL TVD scheme [22] are used for the spatial difference. The second precision central difference is used for the viscosity term, and the SST model [13] is used for the turbulent flow model. The lower-upper symmetric Gauss-Seidel (LU-SGS) method [23] is used for time integration.

1.2. Computational Space and Grid

As shown in Fig. 1a, the actual turbine is calculated in the grid-structured calculation space shown in Fig. 1b in the Numerical Turbine code. Figure 2 is a schematic diagram of Fig. 1b. In the figure, the stator-blade rows and the rotor-blade rows are marked as 1S, 1R, 2S, 2R, 3S, and 3R, where S is a stator-blade row and R is a rotor-blade row. A pair of a stator-blade row and a rotor-blade row is called a stage. In addition to a stator-blade row and a rotor-blade row, there are an inlet region in front of the first stage, an outlet region in back of the last stage, and intermediate regions in between neighboring blade rows. Each stator- and rotor-blade row consists of grid blocks for each blade passage. Similar to the rows, the other regions consist of grid blocks for each passage. Figure 3 illustrates a schematic diagram of the definition of the computational grid number of a grid block. In the Numerical Turbine, the axial grid number is defined as I, the circumferential grid number as J, and the radial grid number as K. The number and the grid size of these grid blocks vary depending on the rows or the regions.

1.3. Load Imbalance in MPI Parallel Execution

The computational space of the Numerical Turbine code is divided into these grid blocks by MPI parallelization. Therefore, each process in MPI parallelization handles its associated grid block. However, as described in the previous sub-section, since the grid block size is not uniform, load imbalance occurs when the Numerical Turbine code runs in MPI parallelization.


Figure 1. Multi-stage stators and rotors of turbine

Figure 2. Schematic of the stator and rotor blades of the three stages

Figure 3. Calculation grid number definition per grid block

Here, the situation of this load imbalance is shown using actual simulation data. The datum is for a full annulus simulation of the first stage of a compressor, and the total number of the grid blocks is 174. Therefore, the maximum number of MPI processes is 174. Table 1 shows the number of the grid blocks and the number of the grid points in each row of the full annulus simulation data of the first stage of the compressor. As shown in this table, this datum has three types of grid points: $91 \times 91 \times 91$, $45 \times 91 \times 91$, and $16 \times 91 \times 91$. In this datum, the grid blocks of Inlet 2, Rotor, Stator, and Outlet 1 have the largest number of grid points at $91 \times 91 \times 91$. The grid blocks of Inlet 1 and Outlet 2 have the number of grid points at $45 \times 91 \times 91$. The grid blocks of Intermediate 1 and Intermediate 2 have the smallest number of grid points at $16 \times 91 \times 91$.

Table 1. Number of grid blocks and grid number of the full annulus data of the first stage of the compressor

	Inlet 1	Inlet 2	Rotor	Inter- mediate 1	Inter- mediate 2	Stator	Outlet 1	Outlet 2	Total number
# of blocks	16	16	32	16	16	46	16	16	174
Grid number	<i>I</i>	45	91	91	16	16	91	91	—
	<i>J</i>	91	91	91	91	91	91	91	—
	<i>K</i>	91	91	91	91	91	91	91	—

Figure 4 shows the cost distribution of the calculation time, the communication time, and the waiting time when the Numerical Turbine code using this simulation datum runs at 174 MPI processes. In the graph of this figure, the horizontal axis shows the rank number of each MPI process, and the vertical axis shows the execution time. The blue, orange, and gray parts respectively indicate the calculation time, the communication time, and the waiting time. As shown

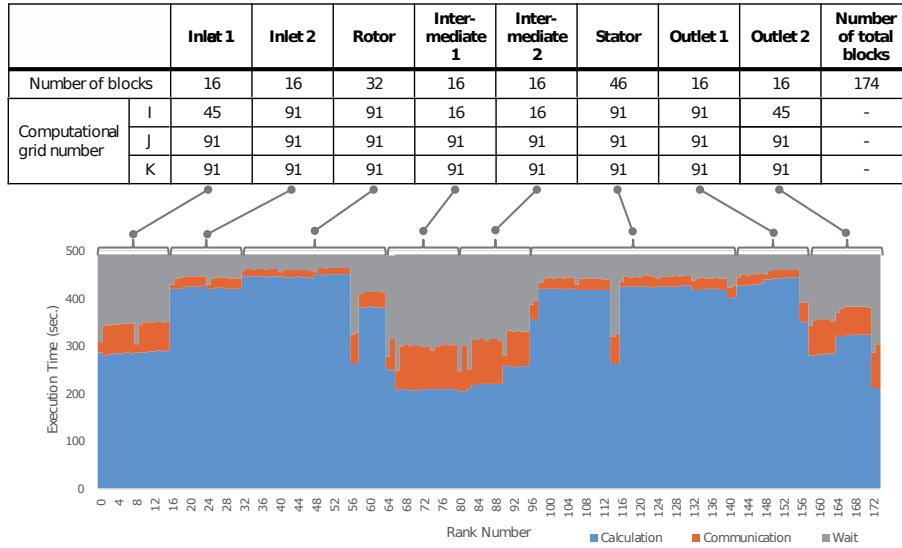


Figure 4. Cost distribution of the compressor in pure MPI and grid number of each grid block

in the figure, the MPI processes of the grid blocks with larger grid points take more time for the calculation than the MPI processes of the grid blocks with fewer grid points. Therefore, the MPI processes of the grid blocks with fewer grid points have a longer waiting time.

In parallel execution with such load imbalance, high-performance simulations with effective use of computational resources are not possible. For realizing parallel execution with effective use of computational resources, the computational load must be leveled. It is necessary to uniform calculation time by parallelizing grid blocks with a larger number of grid points for realizing such execution.

1.4. Related Work

For performing large-scale simulations, parallel execution of simulation codes is indispensable. However, to realize efficient parallel execution, it is necessary to reduce the load imbalance that occurs during parallel execution as much as possible. Research to reduce such load imbalance has been widely conducted.

Musa et al. [15] demonstrated that the workload of each process needed to be as equal as possible for achieving efficient parallelization. Their tsunami simulation program was parallelized by the domain decomposition method using MPI. In the simulation program, calculation amounts in the land areas differed from those in the sea areas. Hence, they adjusted the grid point number of each process to coincide with roughly the same calculation amount. First, the calculation amounts of each MPI process were measured with the same number of grid points per MPI process. Then the grid point number on each MPI process was adjusted to coincide with the almost equal calculation amount by using the previous measurement. As a result, they showed that the load balance in each MPI process handling the nearly even calculation amounts was better than the load balance in each MPI process with the same number of grid points. However, in the vector architecture, the length of the vectorized loop has a significant effect on the performance. Therefore, increasing and decreasing both the calculation amount and the vector length need to be considered for improving the load balance. In addition, the method of Musa et al. [15] needs to be executed in advance to equalize the amount of calculation, even

though the tsunami simulation program causes static load imbalance. The method is not suitable for the Numerical Turbine code that targets various simulation data.

Simmendinger et al. [18] explained how further splitting the block into smaller parts for a block-structured CFD solver was often impractical. They mentioned that smaller MPI domains came with high overhead in terms of communication and decreased convergence rates for implicit solvers. Therefore, they implemented a hybrid parallelization model based on pthreads and MPI. Their implementation showed a significant speed-up when using more cores than domains existing in the mesh.

Giovannini et al. [5] proposed exploiting vectorization capability for serial optimization of a 3-D multi-row, multi-block CFD solver. They also proposed implementing hybrid parallelization (MPI+OpenMP) for the parallelization of the CFD solver. They described that more finely partitioning does not necessarily result in higher scalability. Furthermore, they mentioned that a high domain decomposition could have a detrimental effect on some convergence accelerating techniques (e.g., residual smoothing, multi-grinding, etc.). For these reasons, they did not implement further domain partitioning and pursued code flexibility adopting a hybrid parallelization strategy. The Numerical Turbine code is also a CFD code with a block structure that constitutes a 3-D multi-row and multi-block structure. As shown in Fig. 4, there are processes with a large ratio of communication time. Therefore, further dividing the blocks in MPI parallelization may lead to increasing the communication time and to an increase in execution time. Hence, as Simmendinger et al. and Giovannini et al. mention, applying thread parallelization such as OpenMP is suitable for dividing the blocks.

Rabenseifner et al. [16] discussed that a hybrid parallelization model of MPI+OpenMP had several advantages, and the benefits include improving the load balance. They indicated that one of the significant advantages of OpenMP over MPI is the possible use of loop scheduling. They also showed that no additional programming work or data movement is required for using these loop scheduling. Moreover, they explained that simple static load imbalance at the external level (MPI) and OpenMP loop scheduling could be used as a compromise for the hybrid case. As Rabenseifner et al. [16] described, by applying hybrid parallelization to an MPI program, it is possible to realize to reduce a load imbalance of the program with easy implementation.

Based on these related studies, a way to improve the load balance of the Numerical Turbine code, a block-structured CFD code, is to apply OpenMP parallelization for dividing the grid blocks further and adjusting the workload. Thereby, the computation time among grid blocks can be equalized. However, when running a program on a vector computer, the load imbalance cannot be solved by simply equalizing the calculation amount since the computational performance depends on the loop length of the vectorized loop. Therefore, this paper proposes a method to achieve good load balance by considering the trade-off between calculation amount and the effect of vector length. A significant point to determine the number of parallelization is to perform appropriate domain dividing and mapping of computational resources among different calculation amounts and computational characteristics for each condition, such as the size and number of grid blocks and the number of blade rows in various turbine simulation. Thus, it is not realistic to execute the program in advance for each simulation condition to determine the number of parallelization for improving the load balance. Hence, the proposed method describes a way to reduce the load imbalance without pre-execution.

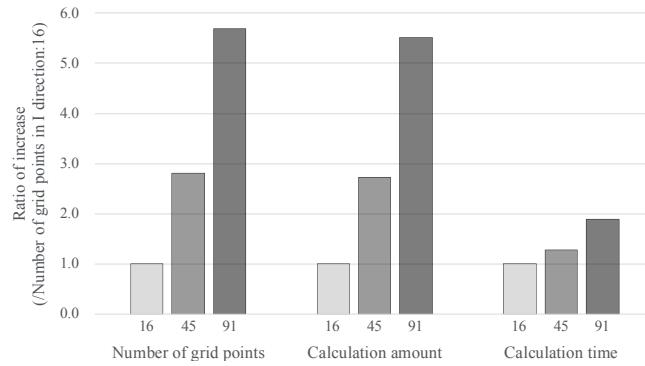


Figure 5. Ratio of calculation amount and calculation time according to the number of grid points

2. Optimizing Load Balance Using Hybrid Parallelization

As mentioned in the previous section, the computational amount of each grid block is different because the Numerical Turbine code is block-structured, and the number of grid points in each grid block is non-uniform. As a result, there is a difference in the calculation time of the MPI process in charge of each grid block. The difference causes load imbalance in MPI parallel execution of the Numerical Turbine code. This paper creates an estimation model that finds the calculation time from each grid block's calculation amount and calculation performance to reduce such load imbalance. It proposes a method of assigning threads to the MPI process in charge of each grid block based on the estimation model.

As shown in Fig. 4, it is clear that the main factor of the load imbalance is the unevenness of the calculation time. In general, the calculation time depends on the calculation amount. If the calculation time is proportional to the calculation amount, the number of threads based on the ratio of the grid points can be assigned to each MPI process. However, in vector processing, the length of the vectorized loop also greatly affects the performance. Figure 5 respectively shows the grid point ratio, calculation amount ratio, and calculation time ratio in the simulation data for a full annulus simulation of the first stage of a compressor shown in the previous section. There are three types of grid points for each grid block in this simulation data. However, these three types of grid points differ only in the number of grid points in the *I*-direction. Therefore, this figure shows the number of grid points in the *I*-direction as the number of grid points. In this figure, each number of grid points in the *I*-direction is 16, 45, and 91. This figure shows values normalized by the case where the number of grid points in the *I*-direction is 16. As shown in this figure, the calculation amount ratio is almost the same as the grid point ratio, whereas the calculation time ratio is smaller than the grid point ratio. This is because the loop length of the vectorized loop becomes longer as the number of the grid points increases so that the calculation performance by vector processing is improved.

For this reason, it is suitable for vector supercomputers to assign the number of threads based on the calculation time ratio. For finding the calculation time ratio, it is necessary to perform pre-execution. However, since the Numerical Turbine code computes simulation data of various problem sizes, it is not suitable in terms of usability to perform pre-execution for each simulation data. Therefore, this paper focuses on the fact that the computation performance is the same if the grid points are the same since the same Numerical Turbine code is used to

calculate different simulation data. It creates the estimation model to obtain the calculation time from each grid block's calculation amount and calculation performance. This paper finds the number of threads assigned to each MPI process without pre-execution based on this model. The following describes the details of the model.

- The following relational equation estimates the calculation time ratio as:

$$\text{calculation time ratio} = \frac{\text{calculation amount ratio}}{\text{calculation performance ratio}}. \quad (1)$$

- The calculation amount ratio can be found in advance because the ratio is equivalent to the grid point ratio.
- The calculation performance can be identified according to the number of grid points because the same Numerical Turbine code is used even for different simulation data. Therefore, the calculation performance ratio can be determined in advance.

Hence, the ratio of the number of threads according to the calculation time ratio can be found in advance by the following equation. According to this ratio, it is possible to determine the number of threads assigned to each MPI process without pre-execution. The ratio of the number of threads is written as:

$$\text{thread number ratio} = \frac{\text{grid point ratio}}{\text{calculation performance ratio}}. \quad (2)$$

The following describes the details of the proposed method based on the above model. To obtain the calculation time ratio without pre-execution, it is necessary to mathematize the relationship in Eq. 1. Hence, the relationship between the number of grid points and the calculation performance in the Numerical Turbine code was clarified by a performance evaluation.

Figure 6 shows the relationship between the number of grid points and the calculation performance in the Numerical Turbine code. To find this relationship, varying the calculation performance was confirmed according to vary the number of grid points using an evaluation code that extracted the calculation parts of the Numerical Turbine code. The calculation parts do not include the communication parts. In this evaluation, only the grid points in the *I*-direction were varied because the number of grid points in the *J*- and *K*-directions of the grid block is the same for all grid blocks.

In this figure, the horizontal axis shows the number of grid points in the *I*-direction, and the vertical axis shows the calculation performance ratio. The calculation performance ratio on the horizontal axis is a normalized value for the calculation performance when the number of grid points in the *I*-direction is 10. The number of grid points in the *I*-direction used is at least two-digit or more in actual simulations. As this figure shows, the calculation performance ratio increases as the number of grid points increases. The relationship is not proportional, and the calculation performance ratio increases significantly until the number of grid points in the *I*-direction changes from two digits to three digits. When the number of grid points in the *I*-direction is about 10, the loop length of the vectorized loops is short, and the vector arithmetic unit cannot fully exhibit its calculation performance. As the number of grid points in the *I*-direction increases, the loop length increases, and the vector arithmetic unit comes to show its high calculation performance.

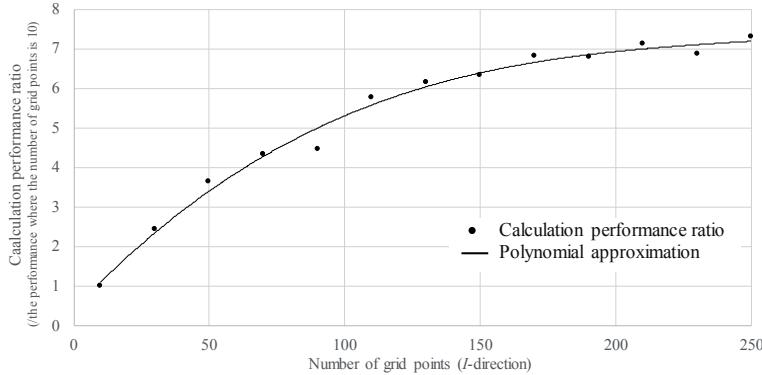


Figure 6. Relationship between the number of grid points and the calculation performance ratio

Table 2. Values of the polynomial coefficients

Coefficient	Value
<i>a</i>	3.62772E-07
<i>b</i>	-0.000272569
<i>c</i>	0.072582348
<i>d</i>	0.412525894

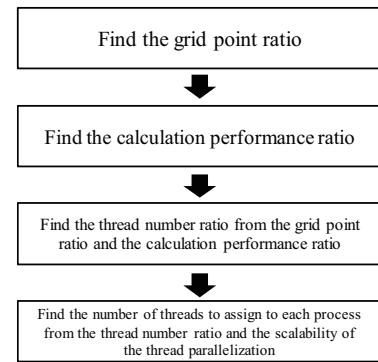


Figure 7. Procedure of finding the number of threads to assign to each process

However, as described in Section 1.4, the calculation amount differs depending on the conditions in the actual simulation. Therefore, it is not appropriate to obtain the calculation performance ratio under various conditions from the graph in terms of accuracy and convenience. Hence, an approximate curve is obtained from the measurement data shown in Fig. 6. In the actual simulation, the calculation performance ratio is obtained from an equation of the approximate curve. Equation 3 is the equation of this approximate curve. The equation of an approximate curve can express the relationship between the number of the grid points in the *I*-direction and the calculation performance ratio shown in Fig. 6.

$$y = a * x^3 + b * x^2 + c * x + d, \quad (3)$$

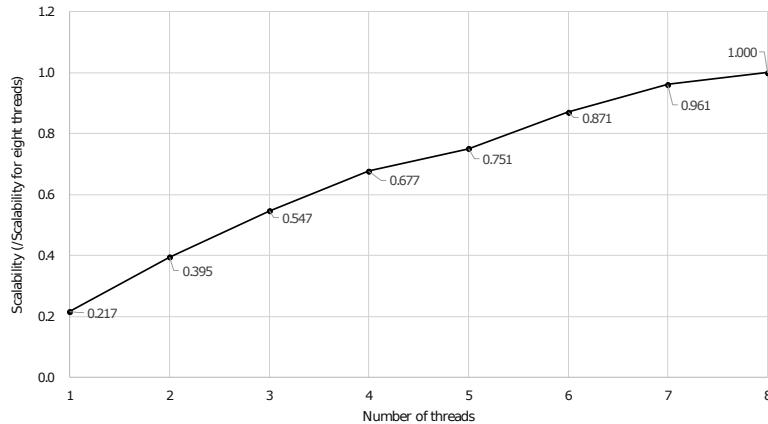
where x is the number of grid points in the *I*-direction, and a , b , c , and d are the polynomial coefficients. Table 2 shows the values of each polynomial coefficient.

Figure 7 shows the procedure of the proposed method, which is roughly composed of four steps. The first step is to calculate each grid point ratio based on the maximum number of grid points (g_N) as shown in Tab. 3, where $g_1 < g_2 < g_3 \dots < g_n < \dots < g_{N-2} < g_{N-1} < g_N$ ($1 \leq n \leq N$). As shown in this table, the grid point ratio at grid point g_n is g_n/g_N .

The second step is to calculate each calculation performance ratio based on the calculation performance of the maximum number of grid points (y_N) as shown in Tab. 3. As shown in this table, the calculation performance ratio at grid point g_n is y_n/y_N . Here, the calculation performance (y_n) is obtained from Fig. 6.

Table 3. Three kinds of ratios

# of grid points	g_1	g_2	g_3	...	g_n	...	g_{N-2}	g_{N-1}	g_N
Grid point ratio	$\frac{g_1}{g_N}$	$\frac{g_2}{g_N}$	$\frac{g_3}{g_N}$...	$\frac{g_n}{g_N}$...	$\frac{g_{N-2}}{g_N}$	$\frac{g_{N-1}}{g_N}$	$\frac{g_N}{g_N}$ $(= 1)$
Calculation performance ratio	$\frac{y_1}{y_N}$	$\frac{y_2}{y_N}$	$\frac{y_3}{y_N}$...	$\frac{y_n}{y_N}$...	$\frac{y_{N-2}}{y_N}$	$\frac{y_{N-1}}{y_N}$	$\frac{y_N}{y_N}$ $(= 1)$
Thread number ratio	$\frac{G_1}{Y_1}$	$\frac{G_2}{Y_2}$	$\frac{G_3}{Y_3}$...	$\frac{G_n}{Y_n}$...	$\frac{G_{N-2}}{Y_{N-2}}$	$\frac{G_{N-1}}{Y_{N-1}}$	$\frac{G_N}{Y_N}$ $(= 1)$

**Figure 8.** Scalability of the thread parallelization of the calculation part

At this point, since the grid point ratio and the calculation performance ratio have been obtained, the calculation time ratio can be obtained by dividing the grid point ratio by the calculation performance ratio at each number of grid points. In the third step, the calculation time ratio is used as the ratio of the number of threads assigned to each grid block because the ratio of the number of threads corresponds to the calculation time ratio. As shown in Tab. 3, $G_n = \frac{g_n}{g_N}$ and $Y_n = \frac{y_n}{y_N}$.

The thread number ratios obtained in Tab. 3 are real numbers less than or equal to 1.0. Since each number of threads is an integer value, it is necessary to determine each number of threads by converting the ratio into an integer value considering performance. Hence, in the final step, each number of threads is determined based on the scalability of thread parallelization in the calculation part of the Numerical Turbine code. Here, the scalability of thread parallelization in the calculation part is obtained as the ratio of the number of threads with the maximum number of the scalability as 1.0. Figure 8 shows the scalability of thread parallelization in the calculation part. This figure identifies the number of threads of the closest scalability to the ratio of the number of threads obtained in Tab. 3. Then, the number of threads is assigned to the grid block of the corresponding number of grid points. Thereby, it is possible to obtain the number of threads according to the actual performance.

Through these four steps, it is possible to determine in advance the number of threads to be assigned to each MPI process based on the ratio corresponding to the calculation time ratio.

Table 4. Specification of Vector Engine: **Table 5.** Software environment of SX-Aurora TSUBASA Type 10AE

Peak performance of core (GFLOPS)	304	NEC Fortran compiler for VE	nfort (NFORT) 3.0.8
Number of cores per VE	8	NEC MPI	NEC MPI 2.10.0
Peak performance of VE (TFLOPS)	2.43		
Memory bandwidth of VE (TB/s)	1.35		
Cache capacity of VE (MB)	16		
Memory capacity of VE (GB)	48		

3. Results and Discussions

This section evaluates the effectiveness of the proposed method to improve the load balance of the Numerical Turbine code by using actual simulation data.

3.1. Input Data Set

This experiment uses the full annulus data of the first stage of the compressor as the input dataset. Table 1 shows the number of grid blocks in each row of full annulus data for the first stage of the compressor and the number of grid points in the grid blocks. As shown in the table, there are a total of 174 blocks and three types of grid points: $91 \times 91 \times 91$, $45 \times 91 \times 91$, and $16 \times 91 \times 91$. The total number of grid points is about 100 million. The number of iterations is 1,000, which is the minimum number required for performance analysis. The total number of grid blocks is 174, so the maximum number of MPI processes is 174.

3.2. Computing Environment Setup

A supercomputer used for this evaluation is the modern vector supercomputer system called SX-Aurora TSUBASA, which was released in 2017 [9, 21], for evaluating the performance of the Numerical Turbine. As described in the previous section, the Numerical Turbine is well optimized for vector architectures. Therefore, to accurately evaluate the overall performance, it is suitable for the performance evaluation on a vector supercomputer. Here, an overview of SX-Aurora TSUBASA is described.

SX-Aurora TSUBASA architecture contains the Vector Engine (VE) and Vector Host (VH), as shown in Fig. 9. The VE executes complete applications, and the VH mainly provides OS functions for connected VEs. The VE consists of one vector processor with eight vector cores, using High Bandwidth Memory modules (HBM2) for uppermost memory bandwidth. The implementation of one CPU LSI with six HBM2 memory modules leads to high memory bandwidth. The VE is connected to the VH, a standard x86/Linux node, through PCIe. This architecture executes an entire application on the VE and the OS on the VH [2].

As shown in Tab. 4, the peak performances of a vector core and a VE are 304 Gflop/s and 2.43 Tflop/s, respectively, and the memory bandwidth of a VE is 1.35 TB/s. Figure 10 shows the configuration of one VH that contains eight VEs, two InfiniBand interfaces, and two Xeon processors. Moreover, SX-Aurora TSUBASA can compose an extensive system by connecting the VHs via the InfiniBand switch. As mentioned in the above explanation, the number of cores per VE of SX-Aurora TSUBASA is eight. Therefore, the maximum number of threads assigned to each MPI process in hybrid parallelization is eight. Regarding the software, Tab. 5 lists the environment of SX-Aurora TSUBASA used in this evaluation.

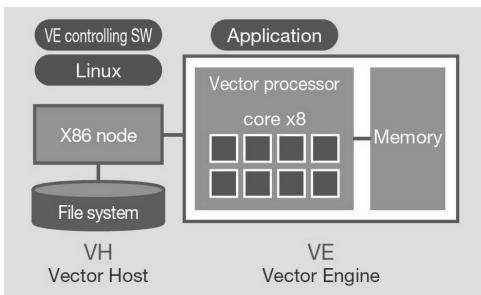


Figure 9. Architecture of SX-Aurora TSUBASA

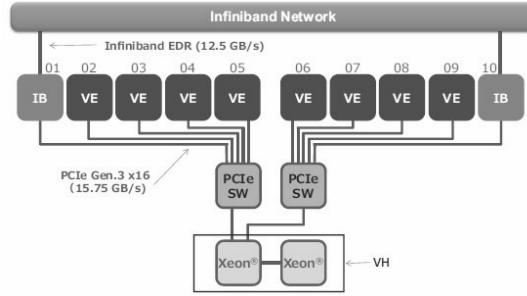


Figure 10. Configuration of SX-Aurora TSUBASA

3.3. Assigning Threads to MPI Processes

This section obtains the number of threads assigning to each MPI process following the method proposed in Section 2 using the full annulus data of the first stage of the compressor and validates the effectiveness of the proposed method.

Each grid point ratio based on the maximum number of grid points is calculated as the first step. As described in Section 3.1, the full annulus data of the first stage of the compressor comprises three types of grid blocks with grid points in the *I*-direction of 16, 45, and 91. Table 6 is obtained by calculating the grid point ratio according to Tab. 3.

Table 6. Three kinds of ratios in the full annulus data of the first stage of the compressor

# of grid points	16	45	91
Grid point ratio	0.176 $(=\frac{16}{91})$	0.495 $(=\frac{45}{91})$	1 $(=\frac{91}{91})$
Calculation performance ratio	0.299 $(=\frac{1.506}{5.033})$	0.628 $(=\frac{3.260}{5.033})$	1 $(=\frac{5.033}{5.033})$
Thread number ratio	0.588 $(=\frac{0.176}{0.299})$	0.788 $(=\frac{0.495}{0.628})$	1 $(=\frac{1}{1})$

As the second step, according to Tab. 3, the calculation performance is obtained based on the calculation performance of the maximum number of grid points. Here, the calculation performance corresponding to each number of grid points is obtained from Fig. 6. Table 6 shows the results of the calculated calculation performance ratios.

The ratio of the number of grid points and the calculation performance ratio in the full annulus data of the first stage of the compressor has been obtained. As the third step, the ratio of the number of threads in each number of grid points is obtained based on Tab. 3. Table 6 shows the ratio of the number of threads obtained.

As the final step, to determine the number of threads in a way that takes performance into account, Fig. 8 is used to find the number of threads with the closest scalability to the ratio of the number of threads obtained in Tab. 6. The ratio of the number of threads in the number of grid points 16 is 0.564. The scalability nearest to this number is 0.547 from Fig. 8, and there are three corresponding threads. Similarly, in the number of grid points 45, the ratio of the number of threads is 0.774, and the scalability of the nearest neighbor to this number is 0.751. Thus, there are five corresponding threads. Since the number of grid points 91 is the maximum number

of grid points, the maximum number of threads is assigned, which is eight. The final number of threads obtained is as shown in Tab. 7.

Table 7. The number of threads obtained by the proposed method

# of grid points	16	45	91
Ratio of the number of threads	0.588	0.788	1
The closest scalability to the above ratios	0.547	0.751	1
# of threads assigning to each MPI process	3	5	8

This section makes a comparison between hybrid parallelization based on the proposed method and the actual calculation time. Table 8 shows the number of threads obtained based on the actual calculation time. Compared to Tab. 7, the number of threads assigned to the grid block with 45 grids is four in the case based on the proposed method, while it is five in the case based on the actual calculation time. On the other hand, the number of threads is the same for the other grids. Therefore, the proposed method can obtain almost the same number of threads as the number of threads obtained based on the actual calculation time. Moreover, the execution time is 144.9 seconds in the case based on the proposed method, and is 144.6 seconds in the case based on the actual calculation time. Hence, the proposed method can obtain the execution time equivalent to hybrid parallelization based on the actual calculation time without pre-execution.

Table 8. The number of threads decided by the actual calculation time

# of grid points	16	45	91
Calculation time (sec.)	216.0	278.1	405.5
Ratio of the number of threads	0.533	0.686	1
The closest scalability to above ratios	0.547	0.677	1
# of threads assigning to each MPI process	3	4	8

Figure 11 shows the execution results by assigning the obtained number of threads to the MPI process in charge of the grid block corresponding to each grid point. To clear the efficiency of the load balance improvement, the vertical axis of this figure is set to the same scale as in Fig. 4. As can be seen from the comparison between Figs. 4 and 11, the hybrid parallelization applying the number of threads obtained by the proposed method equalizes the computation time and achieves good load balance.

For verifying the improvement effect of the load balance by the proposed method, the improvement in the variation between the pure MPI and the proposed hybrid parallelization is confirmed. The coefficient of variation is used to indicate the degree of variation. Table 9 shows the coefficient of variation for the calculation time in the pure MPI and the proposed hybrid parallelization. This table shows that the load balance is improved from 24.4 % to 9.3 % by the hybrid parallelization using the proposed method.

For verifying the efficiency of the computational resource utilization, the proposed hybrid parallelization is compared with the maximum number of threads assigned to all processes. The maximum number of threads assigned to each process is eight because the maximum number of cores per VE is eight, as shown in Tab. 4. Table 10 shows the execution time and the number of cores used for the execution in both cases. As this table shows, the proposed hybrid

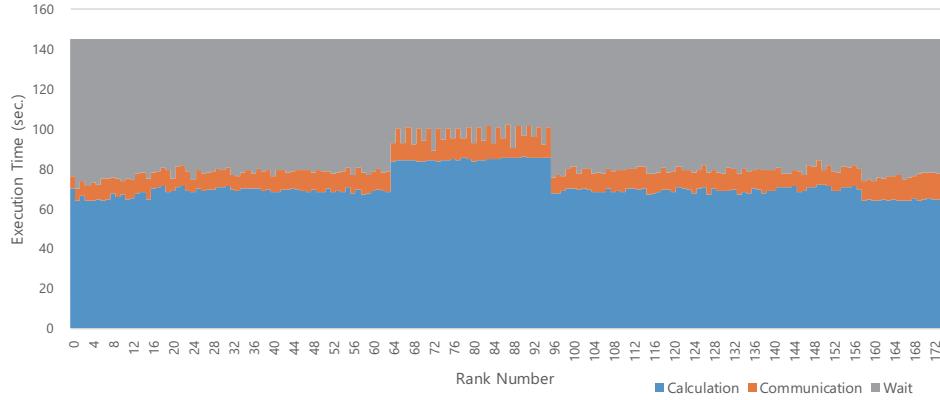


Figure 11. Cost distribution of the compressor in hybrid parallelization reducing load imbalance

Table 9. Coefficient of variation (CV) of the calculation time in each case

	Calculation Time (sec.)			
	minimum	maximum	average	CV
Pure MPI	205.83	441.99	348.42	24.4 %
Proposed hybrid parallelization	63.74	85.92	71.39	9.3 %

parallelization has the same execution time using 82 % of the computational resources of the maximum number of threads assigned to all processes. This section demonstrates the above verification results to reduce the load imbalance by hybrid parallelization, assigning the number of threads in advance based on the proposed method. As a result, the Numerical Turbine code can calculate simulations faster with efficient use of computational resources.

Table 10. Execution time and computational resources used

	Execution time (sec.)	# of cores used
Proposed hybrid parallelization	144.9	1,136
Hybrid parallelization using the maximum number of threads	145.4	1,392

Conclusions

This paper demonstrates that a way to improve the load balance of the Numerical Turbine code, a block-structured CFD code, is to apply OpenMP parallelization for dividing the grid blocks further and adjusting the workload. Thereby, the calculation time among grid blocks can be equalized. For executing the Numerical Turbine code on a vector computer, this paper creates an estimation model that finds the calculation time from each grid block's calculation amount and calculation performance. This proposed method reduces the load imbalance by considering the calculation amount and the effect of vector length based on the model. Moreover, the Numerical Turbine code has a static load imbalance and treats various simulation data. Hence, this proposed method can find suitable numbers of threads to reduce the load imbalance without pre-execution. As a result, the proposed method can improve the load balance from

24.4 % to 9.3 %, and realize 3.32 times speed-up of the Numerical Turbine code with effective usage of the computational resources.

As mentioned above, the Numerical Turbine code treats various simulation data. Some of these data have more grid blocks and more stages than the full annulus data of the first stage of the compressor used for this evaluation. Our future work will verify the effectiveness of the proposed method in these various simulation data treated by the Numerical Turbine code. In addition, the proposed method has been developed for vector supercomputers. This method may be effective for modern scalar supercomputers because the SIMD mechanism has been strengthened, and multi-core has been advanced in such scalar supercomputers. Therefore, our future work will also verify the effectiveness of this method for scalar supercomputers and improve it to a general-purpose method.

Acknowledgements

This research was supported in part by MEXT as “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.” The authors thank Satoru Yamamoto, Takashi Furusawa, and Hironori Miyazawa of Tohoku University for their fruitful discussions and variable comments.

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. Society 5.0. https://www8.cao.go.jp/cstp/english/society5_0/index.html, accessed: 2021-07-02
2. Vector Supercomputer SX Series SX-Aurora TSUBASA. https://www.nec.com/en/global/solutions/hpc/sx/docs/SX-Aurora_e.pdf, accessed: 2021-06-13
3. Egawa, R., Komatsu, K., Isobe, Y., *et al.*: Performance and power analysis of SX-ACE using HP-X benchmark programs. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER). pp. 693–700. IEEE Computer Society (2017). <https://doi.org/10.1109/CLUSTER.2017.65>
4. Egawa, R., Fujimoto, S., Yamashita, T., *et al.*: Exploiting the Potentials of the Second Generation SX-Aurora TSUBASA. In: 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). pp. 39–49. IEEE (2020). <https://doi.org/10.1109/PMBS51919.2020.00010>
5. Giovannini, M., Marconcini, M., Arnone, A., Dominguez, A.: A Hybrid Parallelization Strategy of a CFD Code for Turbomachinery Applications. In: 11th European Conference on Turbomachinery Fluid Dynamics and Thermodynamics, ETC 2015, Madrid, Spain, March 23-27, 2015 (2015)

6. Gyarmathy, G.: Zur Wachstumsgeschwindigkeit kleiner Flüssigkeitstropfen in einer übersättigten Atmosphäre. Zeitschrift für angewandte Mathematik und Physik ZAMP 14(3), 280–293 (1963). <https://doi.org/10.1007/BF01601066>
7. Hougi, Y., Komatsu, K., Watanabe, O., et al.: A hierarchical wavefront method for LU-SGS on modern multi-core vector processors. In: 32nd International Conference on Parallel Computational Fluid Dynamics (2020)
8. Ishizaka, K.: A High-Resolution Numerical Method for Transonic Non-Equilibrium Condensation Flow through a Steam Turbine Cascade. Proc. of the 6th ISCFD, 1995 1, 479–484 (1995)
9. Komatsu, K., Momose, S., Isobe, Y., et al.: Performance Evaluation of a Vector Supercomputer SX-Aurora TSUBASA. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 685–696. IEEE (2018). <https://doi.org/10.1109/SC.2018.00057>
10. Komatsu, K., Miyazawa, H., Yiran, C., Sato, M., Furusawa, T., Yamamoto, S., Kobayashi, H.: Detection of machinery failure signs from big time-series data obtained by flow simulation of intermediate-pressure steam turbines (2021)
11. Lindner, F., Totounferoush, A., Mehl, M., et al.: ExaFSA: Parallel Fluid-Structure-Acoustic Simulation. In: Software for Exascale Computing - SPPEXA 2016-2019. Lecture Notes in Computational Science and Engineering, vol. 136, pp. 271–300. Springer (2020). https://doi.org/10.1007/978-3-030-47956-5_10
12. MacDougall, F.H.: Kinetic Theory of Liquids. By J. Frenkel. The Journal of Physical and Colloid Chemistry 51(4), 1032–1033 (1947). <https://doi.org/10.1021/j150454a025>
13. Menter, F.R.: Two-equation eddy-viscosity turbulence models for engineering applications. AIAA Journal 32(8), 1598–1605 (1994). <https://doi.org/10.2514/3.12149>
14. Miyake, S., Koda, I., Yamamoto, S., et al.: Unsteady Wake and Vortex Interactions in 3-D Steam Turbine Low Pressure Final Three Stages. Turbo Expo: Power for Land, Sea, and Air, vol. Volume 1B: Marine; Microturbines, Turbochargers and Small Turbomachines; Steam Turbines (2014). <https://doi.org/10.1115/GT2014-25491>
15. Musa, A., Watanabe, O., Matsuoka, H., et al.: Real-time tsunami inundation forecast system for tsunami disaster prevention and mitigation. Journal of Supercomputing 74(7), 3093–3113 (2018). <https://doi.org/10.1007/s11227-018-2363-0>
16. Rabenseifner, R., Hager, G., Jost, G.: Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, Weimar, Germany, Feb. 18-20, 2009. pp. 427–436. IEEE (2009). <https://doi.org/10.1109/PDP.2009.43>
17. Roe, P.L.: Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes. J. Comput. Phys. 135(2), 250–258 (1997). <https://doi.org/10.1006/jcph.1997.5705>
18. Simmendinger, C., Kuegeler, E.: Hybrid Parallelization of a Turbomachinery CFD Code: Performance Enhancements on Multicore Architectures pp. 14–17 (2010)

19. Soga, T., Musa, A., Shimomura, Y., *et al.*: Performance evaluation of NEC SX-9 using real science and engineering applications. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. pp. 1–12. ACM (2009). <https://doi.org/10.1145/1654059.1654088>
20. Watanabe, O., Hougi, Y., Komatsu, K., *et al.*: Optimizing memory layout of hyperplane ordering for vector supercomputer SX-Aurora TSUBASA. In: 2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC), Denver, CO, USA, Nov. 18, 2019. pp. 25–32. IEEE (2019). <https://doi.org/10.1109/MCHPC49590.2019.00011>
21. Yamada, Y., Momose, S.: Vector engine processor of NEC’s brand-new supercomputer SX-Aurora TSUBASA. In: International symposium on High Performance Chips (Hot Chips2018) (2018)
22. Yamamoto, S., Daiguji, H.: Higher-order-accurate upwind schemes for solving the compressible Euler and Navier-Stokes equations. *Computers & Fluids* 22(2), 259–270 (1993). [https://doi.org/10.1016/0045-7930\(93\)90058-H](https://doi.org/10.1016/0045-7930(93)90058-H)
23. Yoon, S., Jameson, A.: Lower-upper Symmetric-Gauss-Seidel method for the Euler and Navier-Stokes equations. *AIAA Journal* 26(9), 1025–1026 (1988). <https://doi.org/10.2514/3.10007>