Automatically Avoiding Memory Access Conflicts on SX-Aurora TSUBASA

Naoki Ebata*, Ryusuke Egawa†*, Yoko Isobe†‡, Ryoji Takaki§, Hiroyuki Takizawa†*
* *Graduate School of Information Sciences*
Tohoku University
Sendai, Japan
Email: naoki.ebata.r7@dc.tohoku.ac.jp
† *Cyberscience Center*
Tohoku University
Sendai, Japan
Email: {egawa, isobe, takizawa}@tohoku.ac.jp
‡ *AI Platform Division*
NEC Corporation
Tokyo, Japan
§ *Institue of Space and Astronautical Science*
Japan Aerospace Exploration Agency
Sagamihara, Japan
Email: ryo@isas.jaxa.jp

***Abstract*—This paper presents a method for automatically avoiding performance degradation due to memory access conflicts on Vector Engines (VEs) of NEC SX-Aurora TSUBASA. Although VEs have the world's top-class theoretical memory bandwidth, the sustained memory bandwidth is significantly decreased if frequent memory access conflicts occur. In order to fully utilize the potential memory bandwidth of VEs, programmers need to optimize codes manually to avoid the conflicts. However, optimizing all applications executed on VEs is a very time-consuming and cumbersome task for programmers. Therefore, we propose a method to automatically avoid memory access conflicts. First, we introduce a metric that can predict memory memory access conflicts based on memory addresses. Then, based on the metric, we implement an array-like C++ class that automatically avoids memory access conflicts. We evaluate the performance gain by our approach using a simple benchmark and kernel codes of a computational fluid dynamics application. The evaluation results demonstrate that the proposed method can successfully reduce the negative effects of memory access conflicts on performance.**

***Keywords*-NEC Vector Engine, Memory Access Conflict, Data Arrangement**

I. Introduction

In recent years, the performance of many scientific applications is limited by the sustained memory bandwidth because the annual growth rate of the memory bandwidth has been smaller than that of the computational performance. This is well known as the *Memory Wall* problem [1]. Therefore, there is a strong demand for a computing system that can achieve a high sustained memory bandwidth for practical scientific workloads.

NEC SX-Aurora TSUBASA (SX-Aurora), which is a modern vector computing system, is a good candidate to meet the demand. SX-Aurora is equipped with an x86 processor (Vector Host, VH) and vector processors (Vector Engines, VEs). VEs have the world's top-class memory bandwidth as of September 2019 [2]. Its theoretical memory bandwidth is 1.2 TB/s (Type 10B). Besides, since the ratio of its double-precision floating-point operation performance to memory bandwidth, the so-called system B/F ratio, is high of 0.5, VE is one of the most well-balanced processors at present. Therefore, VEs are highly suited to execute memory-intensive applications. Some studies have reported that VEs show outstanding performances for such applications [3][4].

However, in some cases, the sustained memory bandwidth can be much lower than the theoretical memory bandwidth. Such performance degradations are caused by the underlying memory architecture. Therefore, in order to fully utilize the potential memory bandwidth of the processors, programmers must write their codes while considering the memory architecture.

A typical cause degrading the sustained memory bandwidth is a memory access conflict. If multiple accesses to a specific memory bank or port occur at the same time, they conflict and result in a severe degradation of sustained memory bandwidth [5]. Although VEs have a high theoretical memory bandwidth compared to previous SX-series processors, the penalty of a memory access conflict is almost the same as those in the previous ones. Therefore, the negative effects of memory access conflicts become relatively larger on VEs.

Figure 1 shows the performance of a VE on a kernel code, named *cflux*, in a computational fluid dynamics (CFD) application, while changing its array size N^3. The performance is measured in terms of Mega Lattice UPdate per second

978-1-7281-7445-7/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPSW50202.2020.00139

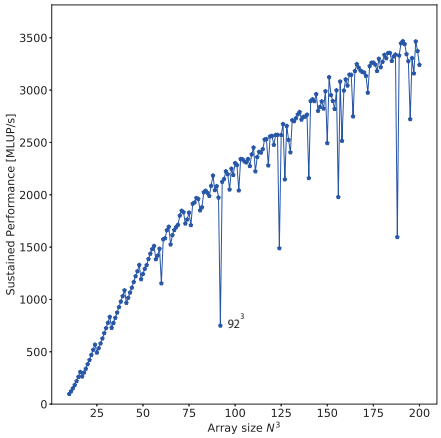


Figure 1. An example of performance degradation due to memory access conflicts.

(MLUP/s). More details of *cflux* are described in Section IV. In Figure 1, sharp performance drops are observed at some array sizes. For example, at the array size 92^3, the sustained performance is decreased by 62% compared to the previous array size 91^3. This performance drop is caused by memory access conflicts, and could be a serious problem for memory-intensive applications.

Padding is a code tuning technique to avoid memory access conflicts by inserting extra elements into an array [6]. For example, if an array is declared as `double a[M][N];` and memory access conflicts occur frequently at this array size, programmers can avoid the conflicts by changing the code to `double a[M][N+p];`. However, the code optimization and tuning of the padding size `p` needs to be manually done by programmers. Therefore, to fully exploit the potential memory bandwidth of VEs, programmers need to rewrite the code and tune the padding size for all applications that run on VEs. It is a very time-consuming task for programmers.

Shuang et al.[7] have proposed a method for automatically optimizing data access by calculating the degree of bank conflict with static array access information about CUDA shared memory. They have demonstrated that their method can improve the performance on several benchmarks. However, the memory architecture assumed in CUDA is totally different from that of vector processors. For example, unlike the assumption in [7], the total number of banks per VE is not a power of 2. Thus, their method cannot be applied to vector processors directly. In addition, although it is widely known that data padding is effective to eliminate bank conflicts on classic vector processors, the architecture design of modern vector processors, such as VEs, is different from that of classic vector processors. A modern vector processor consists of multi cores, has hierarchical caches, and supports out-of-order execution. As a result, such a processor has several runtime factors, on which the memory access pattern could depend. Therefore, conventional approaches for classic vector processors are not as effective as ever, and thus this paper discusses a novel approach for modern vector processors.

In this work, we propose a method to automatically avoid memory access conflicts on VEs. We analyze the relationship between memory addresses and memory access conflicts on VEs and introduce a simple metric that is useful to predict memory access conflicts. We first demonstrate that the metric works for a simple benchmark. Also, based on the metric, we implement an array-like C++ class that allocates the memory so as to prevent memory access conflicts. We also demonstrate that this implementation can prevent memory access conflicts for a simple and kernel codes of a CFD application.

The rest of this work is as follows. Section II explains the correspondence between memory addresses and memory banks in VEs. Section III describes proposed metrics and the C++ classes in detail. In Section IV, the performance evaluation results using a simple benchmark and four kernel codes are shown to discuss the usefulness of the proposed approach. Section V summarizes this work and describes our future prospects.

II. Memory Configuration of Vector Engines

VEs, the latest vector processors, can perform the same 256 operations with a single instruction. In order to take advantage of such a high throughput, VEs are designed to be able to efficiently fetch a large amount of data from main memory. To achieve the high theretical peak memory bandwidth, VE uses High Bandwidth Memory 2 (HBM2) as the main memory module. One HBM2 module is able to transfer 0.2 TB of data per second and a VE has six HBM2 modules. Therefore, the entire theoretical memory bandwidth is derived as $6 \times 0.2 = 1.2$ TB/s.

For efficient data transfers, VEs have many memory channels and memory banks. The HBM2 module is connected to Last Level Cache (LLC) with eight channels, and each channel has 32 memory banks in the case of VE Type 10B. Thus, a VE has $6 \times 8 \times 32 = 1,536$ memory banks. When all memory access requests go to different memory banks, memory access conflicts do not occur and each channel can transfer data efficiently. Therefore, it is important to simultaneously access as many memory banks as possible to achieve high sustained memory bandwidth.

In VEs, a sequence of memory addresses is assigned to physical memory banks by round-robin for memory allocation. VEs handle a 128-byte continuous data region as one unit called a *memory cell*, which is assigned to the same memory bank. In this paper, a mapping of memory cells to banks is called a *data arrangement*. Also, a memory cell is the unit of data transfer between main memory and LLC. For discussions, we express the b-th bank of

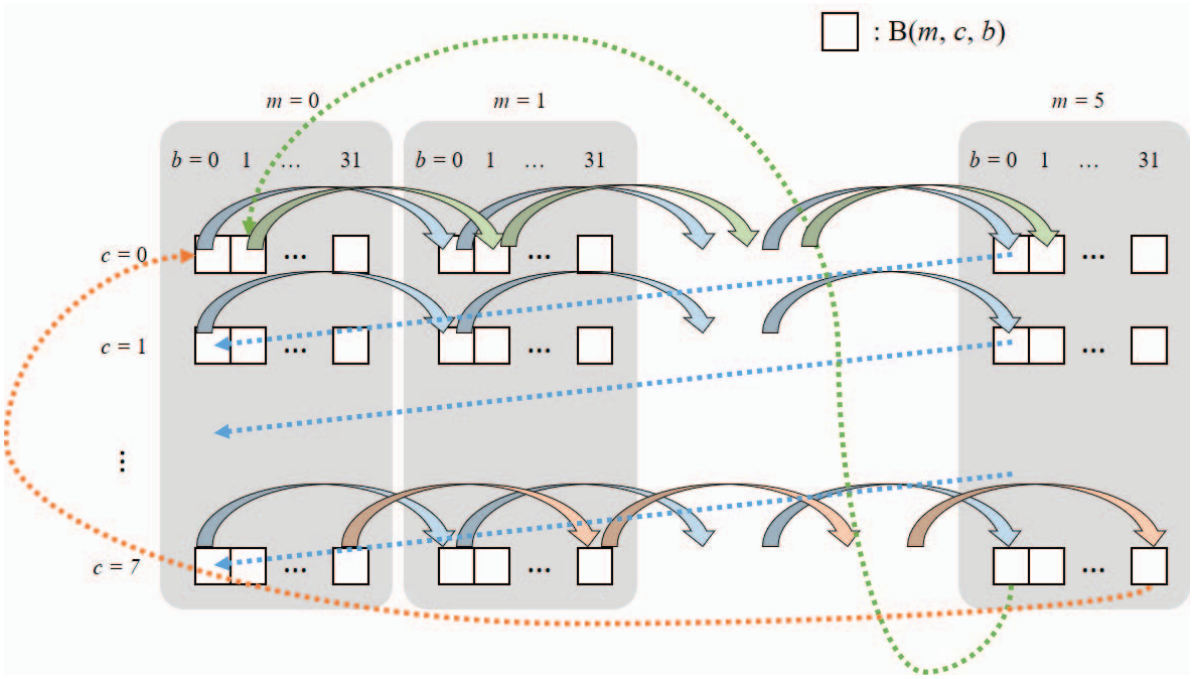


Figure 2. The assingment of memory banks.

the c-th channel in the m-th memory module as B(m, c, b), and Figure 2 illustrates the following explanation in a schematic way. First, consecutive memory cells are assigned to different memory modules preferentially such as B(0, 0, 0)→B(1, 0, 0)→B(2, 0, 0)→...→B(5, 0, 0). After going around all modules, memory cells are assigned in the channel priority order. For example, B(5, 0, 0)→B(0, 1, 0)→B(1, 1, 0)→...→B(5, 1 ,0)→B(0, 2, 0)→...→B(5, 7, 0). Finally, after going around all channels, memory cells are assigned in order of memory banks such as B(5, 7, 0)→B(0, 0, 1)→B(0, 0, 1)→...→B(5, 0, 1)→B(0, 1, 2)→...→B(5, 7, 31). In the case of VE Type 10, after a memory cell is assigned to B(5, 7, 31), the next cell is assigned to B(0, 0, 0).

One of the reasons why using the round-robin method is that consecutive data are likely to be accessed in a short period of time. If data accessed in a short period use the same memory component (a channel or a bank), the requests are concentrated on the specific component. Therefore, by using the round-robin method, consecutive data are physically distributed and arranged so that as many channels and banks as possible can be used in parallel.

However, despite the use of such an arrangement method, memory access conflicts frequently occur in a specific data arrangement as shown in Figure 1. This is because the elements of multiple arrays are assigned to the same memory bank and simultaneously accessed. Accordingly, in order to eliminate the effects of memory access conflicts, an additional mechanism for data arrangement is required.

III. Avoiding Memory Access Conflicts

This section describes the proposed method. In Section III-A, we introduce a simple metric d that can predict memory access conflicts if accessed memory addresses are known. Section III-B describes a C++ class, `abc` that implements a memory region so as to avoid memory access conflicts based on the proposed metrics.

A. A metric to predict memory access conflicts

As described in Section II, a way of arranging data onto memory modules determines how frequently memory access conflicts occur. In other words, if the data arrangement in main memory is known, it is possible to estimate the probability of conflicts. We introduce a simple metric d that can quantify the risk of memory access conflicts based on memory architecture and memory addresses.

For the two data, `a` and `b`, that are accessed in a short time period, metric d is defined as Equation (1) where $ADRS_{\mathrm{a}}$ and $ADRS_{\mathrm{b}}$ are the memory addresses of `a` and `b`, respectively, N_{bank} is the number of memory banks, and S_{cell} is the size of a memory cell in byte.

$$d = (\lfloor \frac{ADRS_{\mathrm{a}}}{S_{\mathrm{cell}}} \rfloor - \lfloor \frac{ADRS_{\mathrm{b}}}{S_{\mathrm{cell}}} \rfloor) \bmod N_{\mathrm{bank}}. \quad (1)$$

This metric simply means the distance between two banks, to which `a` and `b` are assigned. Under the assumption where the memory bank assigning rule is the same as described in Section II, it is possible to identify the memory bank having a memory cell if its memory address is known. Also,

```
1 for(int i = 0; i < N; i++){
2       b[i] += a[i];
3 }
```

Figure 3. Source code of *vector add*.

Table I
SYSTEM SPECIFICATIONS

NEC SX-Aurora TSUBASA A300-8	
VE Type	Type 10B
Compiler	nc++ (NCC) 2.5.1
VEOS	veos-2.2.2-1.el7.x86_64
Theoretical memory bandwidth	2.15 (TFLOPS)
Theoretical computational performance	1.20 (TB/s)

the distance between assigned locations of two memory addresses can be calculated as follows. First, the memory addresses are divided by the memory cell size and the resultant fraction is truncated because a 128-byte-aligned continuous memory region is in the same memory cell. That is, the initial address of the memory cell including a given memory address is calculated. After that, the address difference between two memory cells is calculated. Finally, the difference is divided by the number of banks, N_{bank}, to calculate the reminder, d, which is the distance between locations of the two banks. Therefore, since the way of arranging data to banks is already known, the distance in banks between data a and b can be calculated as d.

This metric easily can be extended to a nested loop. As an example, suppose a simple benchmark, *vector add*, shown in Figure 3, which adds two vectors, `a` and `b`. Then, the array elements accessed at the first loop iteration are `a[0]` and `b[0]`. Let $d(\mathrm{a[i]}, \mathrm{b[i]})$ be the difference d between `a[i]` and `b[i]`. Then, $d(\mathrm{a[i]}, \mathrm{b[i]})$ can be approximated by $d(\mathrm{a[0]}, \mathrm{b[0]})$ in the cases of loops with regular memory access patterns. Although this approximation is not available for irregular memory access patterns, many scientific applications have kernels with regular memory access patterns, such as stencil and stream kernels. Therefore, the proposed method can reduce the negative effects of memory access conflicts by using the metric to estimate the risk of conflicts.

Using this metric, we empirically investigate the relationship between d and performance on the *vector add* kernel. The system specification used in this preliminary evaluation is shown in Table I.

One iteration of *vector add* performs three memory accesses (`load a[i]` and `b[i]`, and `store b[i]`), and one addition. Thus, the ratio of memory access to double-precision floating-point operation in the code, the so-called code B/F ratio, is $sizeof(double) \times 3/1 = 24$. Therefore, the performance of *vector add* should be limited simply by the sustained memory bandwidth if the loop length is large enough.

Figure 4(a) shows that the preliminary evaluation results. As shown in the figure, the sustained memory bandwidth significantly degrades at particular array sizes because memory access conflicts frequently occur at the sizes. Therefore, even in such a simple benchmark, it is difficult to make full use of the theoretical memory bandwidth.

In Figure 4(b), the sustained memory bandwidths are sorted by d. It clearly shows the relationship between d and the sustained memory bandwidth. There is an obvious tendency that the sustained memory bandwidth degrades when d is approximately a multiple of 512. By using this magic number of 512, we can estimate that the risk of memory access conflicts is high or low. In this work, therefore, we define that the risk is high if Equation (2) holds:

$$512i - \alpha \leq d \leq 512i + \alpha, \tag{2}$$

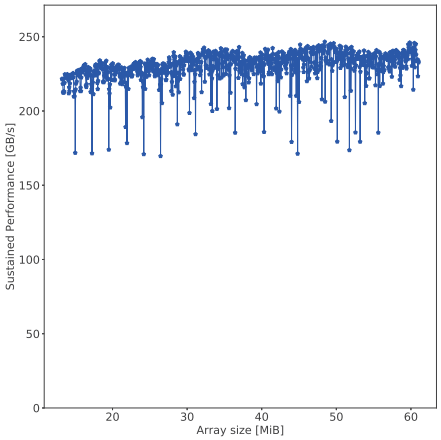
where $i = 0, 1, \cdots, N_{bank}/512$, and α is a hardware-specific constant parameter. In the case of VE Type 10B used in this work, α is set to 32. The estimation accuracy will be discussed in Section IV.

B. A C++ class library to avoid memory access conflicts

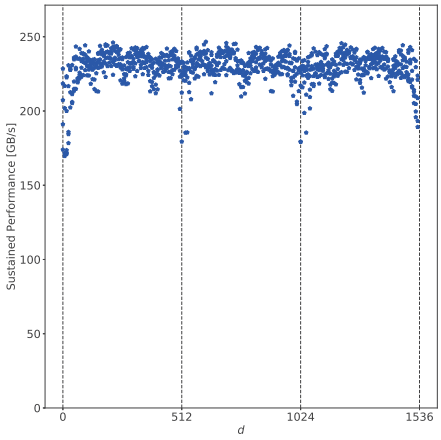
We propose a method for automatically avoiding memory access conflicts using the metric defined by Equation (1). In this work, we design a C++ class library, named `abc`, to automate the data padding to avoid memory access conflicts.

Figure 5 illustrates how to use the `abc` class, taking *vector add* as an example. The library, `abc`, offers two classes, `abc::group` and `abc::array`. An array can be represented by an instance of `abc::array`. Multiple instances of `abc::array` are grouped by using `abc::group`. Every instance of `abc::array` is declared with using an instance of `abc::group` as the second argument. In Figure 5, `sample` declared in Line 5 is an instance of `abc::group`, and used to declare two instances of `abc::array`, `a` and `b`, in Lines 6 and 7. The constructor of `abc::array` allocates a memory region so that the risk of memory access conflicts given by Equation (2) becomes low for other instances of `abc::array` within the same group. Therefore, the `abc` library can automate the memory allocation with lowering the risk of memory access conflicts.

As shown in Figure 5, every element of `abc::array`s can be accessed by using the parenthesis operator such as `a(i,j,k)` corresponds to `a[i][j][k]`. However, the overhead of operator () could be visible in performance. Therefore, the `abc::array` class also offers member function, `abc::array::get_ptr()`, to retrieve the initial address of the memory region allocated for the instance. For example, if `a` is instantiated by `abc::array<double> a (Ni,Nj,Nk,sample)`, its member function, `a.get_ptr()` returns the address of `a(0,0,0)` in the double pointer type. In Section IV, we use this function for avoiding the overhead.



(a) The behavior of sustained memory bandwith when running *vector add* changing its array size.



(b) The relationship between d and sustained memory bandwidth.

Figure 4. The performance analysis of *vector add*.

```
1  #include "abc.cpp"
2
3  int main(){
4        const int N = 10000;
5        abc::group sample;
6        abc::array<double> a(N, sample);
7        abc::array<double> b(N, sample);
8
9        // initialize
10       for(int i = 0; i < N; i++){
11             a(i) = 1.0;
12             b(i) = 2.0;
13       }
14
15       // vector add
16       for(int i = 0; i < N; i++){
17             b(i) += a(i);
18       }
19
20       return 0;
21 }
```

Figure 5. Source code of *vector add* implimented with `abc`.

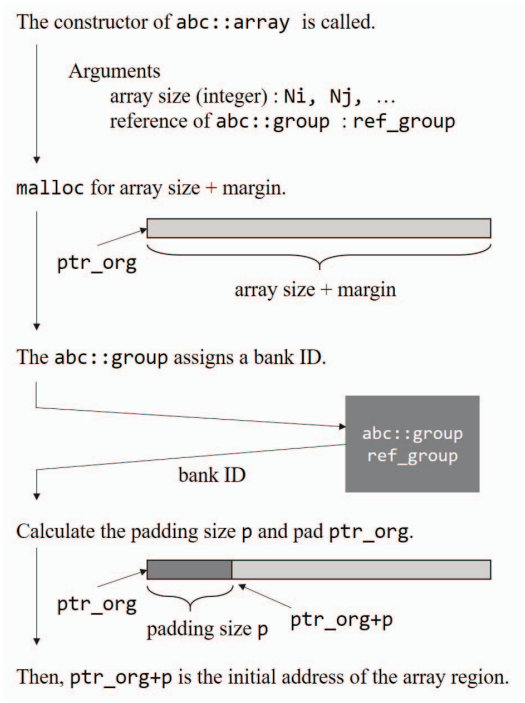


Figure 6. How the `abc::array` constructor works.

Figure 6 shows how the `abc::array` constructor works. The constructor takes the array size in each dimension and a reference to an `abc::group` instance. First, the standard `malloc` function is called to allocate a memory region. At the memory allocation, the region size is set to a specified size plus a certain margin. In this work, margin is set to $N_{bank} \times S_{cell}$. Suppose that the `malloc` function call returns with `ptr_org`. Then, `abc::group` converts `ptr_org` to a bank ID, and calculates d for another instance of `abc::array` within the same group. Finally, the padding size, p, is calculated so that the risk of memory access conflicts is low as mentioned later. The `abc::get_ptr()` returns `ptr_org+p` as the initial address of the allocated memory region.

The most important point in this class library is how the `abc::group` assigns a bank ID to an `abc::array`. The `abc::group` needs to suppress the risk of conflicts for any combination of `abc::array` instances in the group. To this end, we employ the following equation to decide the bank ID of each abc::array instance.

$$ID(i) = \begin{cases} 0 \quad (i = 1) \\ \frac{N_{bank} \times \{2 \times \{(i-1) \bmod 2^{\lfloor \log_2(i-1) \rfloor}\} + 1\}}{2^{\lfloor \log_2(i-1) \rfloor + 1}} \quad (i \neq 1), \end{cases} \tag{3}$$

where $ID(i)$ is the bank ID for the i-th `abc::array` in an

Table II
THE QUANTITATIVE EVALUATION OF *vector add*

	Original	`abc`
Maximum	246.6 GB/s	237.4 GB/s
Minimum	169.6 GB/s	209.8 GB/s
Average	229.8 GB/s	224.3 GB/s
Standard deviation	10.7	6.8

`abc::group`. If the number of `abc::array` instances in a group is less than a certain threshold, Equation (3) guarantees that the distance between any pair of the `abc::array` instances in the same group holds Equation (2). The threshold value depends on the hardware as discussed in Section IV.

IV. PERFORMANCE EVALUATION

In this section, we evaluate our method for *vector add* shown in Fig 3 and four kernels of UPACS-Parts (Unified Platform for Aerospace Computational Simulation-Parts) [8], which is a group of CFD application kernels. They can be automatically vectorized by the NEC C++ compiler (nc++) by selecting the appropriate compiler directives and compile options. In addition, UPACS-Parts are parallelized with OpenMP [9]. The system used for the evaluation is shown in Table I.

A. Vector Add

In this section, we evaluate our proposal on *vector add* shown in Figure 3. We execute *vector add* on a single core of a VE using `abc::array::get_ptr()`.

Figure 7 shows the sustained performance of the original code and the code implemented with `abc`. As shown in the figure, the `abc` implementation code successfully avoids the performance degradation caused by memory access conflicts. While there are some sharp performance drops in the original implementation, the `abc` implementation eliminates them.

For quantitative evaluation, we show the minimum, maximum, average and standard deviation of the sustained memory bandwidths in Table II. The maximum and average bandwidths of the `abc` implementation are slightly smaller than those of the original one due to the side effects of manually changing the data arrangements. On the other hand, the minimum bandwidth of the `abc` implementation is higher than that of the original one. Also, the standard deviation of the sustained memory bandwidths with the `abc` implementation is smaller than that with the original one. Therefore, these results demonstrate that `abc` can successfully avoid the significant performance degradations that occur in the original code.

B. UPACS-Parts

In this section, we discuss the performance gain of our proposal using UPACS-Parts, which is a group of kernel codes used in UPACS [8]. UPACS is a CFD code for aerospace applications that has been developed by JAXA (Japan Aerospace eXploration Agency). We evaluate the performance of our method for four kernels of UPACS-Parts: *cflux*, *vflux*, *cfacev*, *muscl*. The specification of the kernels is shown in Table III. Since all of them have a relatively large number of memory access operations, compared to the number of floating point operations, their performance is limited by the sustained memory bandwidth. Therefore, to efficiently execute CFD applications such as UPACS, it is important to fully exploit the system's potential memory bandwidth and thereby to achieve a high sustained memory bandwidth.

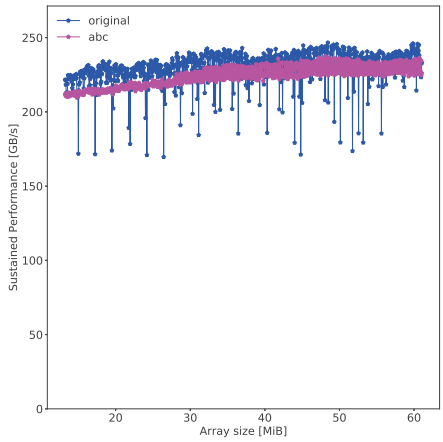
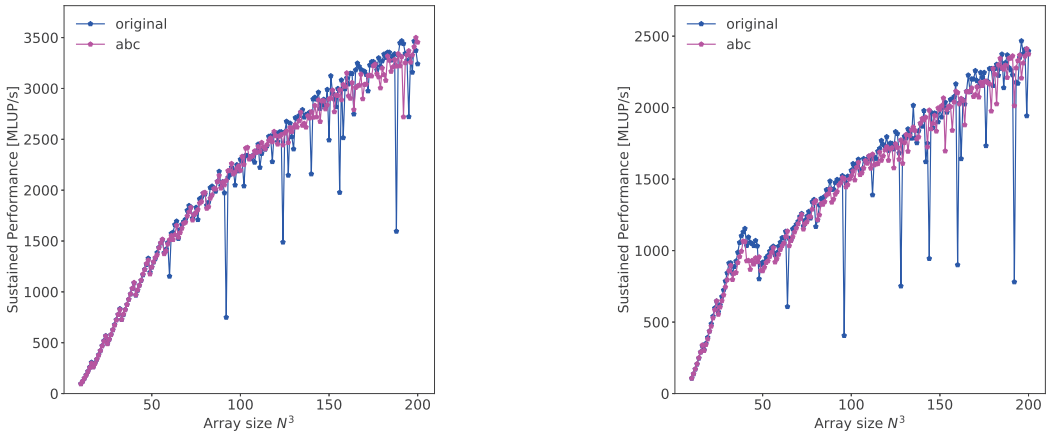


Figure 7. The evaluation results of *vector add*.

Since the original UPACS-Parts kernels are written in Fortran, we rewrite four kernels in C++. Although there are some trivial differences between them such as Fortran specific semantics, it is almost no effect to the performance characteristics. Note that there is no technical challenge to use the proposed method itself for Fortran codes. Although the programming interface of `abc` might need to be redesigned for Fortran, it hardly changes the performance characteristics. Since many scientific applications are written in Fortran, the Fortran interface of `abc` would also be discussed in the future.

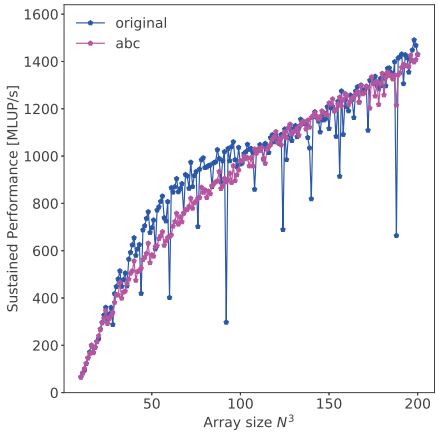
In addition, we implemented these kernel codes with `abc`. Since all the kernels consist of a nested loop, `abc::arrays` belong to a same `abc::group`. We use `abc::array::get_ptr()` to avoid overheads.

Figure 8 shows the sustained memory bandwidth of the original C++ implementation and the `abc` implementation while changing its array size N^3 from 10^3 to 200^3. In the following evaluation, since UPACS-Parts kernels are parallelized with OpenMP, we execute them with eight threads to use all the cores of a VE. In both implementations, the sustained memory bandwidths grow in almost proportional to their array sizes. The reason is that sustained memory

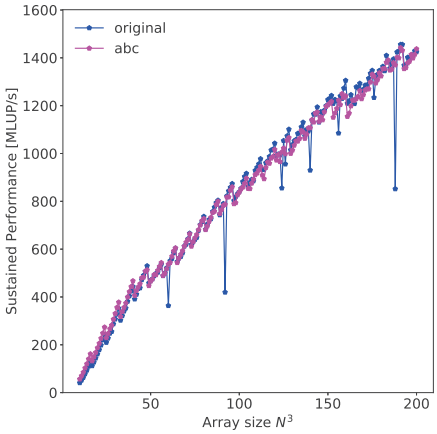


(a) *cflux*

(b) *vflux*



(c) *cfacev*



(d) *muscl*

Figure 8. The evaluation results of UPACS-Parts.

Table III
THE SPECIFICATION OF UPACS-PARTS

Kernel	Computaion	Eq. term	Prog. Type
cflux	Flux	Convention	Stream
vflux	Flux	Viscous	Stream
muscl	Physical quantity complementation of cell surface	Convention	Stencil
cfacev	Physical quantity complementation of cell surface	Vicous	Stencil

bandwidth of a VE would basically grow linearly with the average vector length until the vector length exceeds 256[1].

Figure 8 clearly shows that, in the original C++ implementation, the performance degradations can be seen at some array sizes for all the four kernels. On the other hand, in the `abc` implementation, the negative effects of memory access conflicts are obviously alleviated. Although even the `abc` implementation cannot perfectly eliminate performance drops, it can significantly reduce not only the number of array sizes causing severe performance drops, but also the performance drop ratio in comparison with the original implementation. These results show that the proposed method and implementation are effective to improve the sustained memory bandwidths of real CFD applications such as UPACS.

The reason of the remaining small performance degradations in the `abc` implementation can be explained as follows. UPACS-Parts loops are parallelized with OpenMP, and hence executed with eight threads in the above evaluation. Thus,

[1]Since the average vector length is determined by the length of the innermost loop, loop optimization techniques such as loop collapse [5] will further improve the performance, even though the loop optimization is outside the scope of this work.

the outermost loop is executed in a work-sharing fashion with eight threads. As a result, an array access of one thread could conflict with another array access of a different thread. In addition, *cfacev* and *muscl* are stencil kernels, and thus multiple elements of the same array are accessed within one loop iteration, resulting in a higher risk of conflicts. To completely prevent the conflicts, we would need careful data layout optimizations such as *intra-padding* [7][10]. However, automatic data layout optimization in a generalized way is not established yet. Consequently, as shown in the evaluation results, this paper has presented a practical approach to achieving high sustained memory bandwidth by considering only the initial address of each array to avoid most of memory access conflicts.

V. Conclusion and future work

This paper has proposed a method to automatically avoid memory access conflicts. First, a simple metric has been proposed to quantify the risk of memory access conflicts. Then, this paper presents an array-like C++ class that allocates a memory region so as to avoid memory access conflicts with considering other arrays in the same group. We have evaluated the performance gain by our proposal using a simple benchmark and kernel codes of a CFD application. As a result, the proposed method can successfully reduce the negative effects of memory access conflicts on performance.

In this work, we have implemented the proposed method as a C++ class. However, this class is specialized for VEs and not portable to other platforms. In the HPC community, there are some widely-used libraries that can arrange data layouts such as `Kokkos` [11]. Therefore, in our future work, we will implement our approach in the form of such a library for VEs.

Acknowledgment

This work is partially supported by MEXT Next Generation High-Performance Computing Infrastructures and Applications R&D Program “R&D of A Quantum-Annealing-Assisted Next Generation HPC Infrastructure and its Applications” and Grant-in-Aid for Scientific Research(B) #16H02822 and #17H01706.

References

[1] W. A. Wulf and S. A. McKee, “Hitting the memory wall: implications of the obvious,” *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[2] “Nec sx-aurora tsubasa - vector engine,” https://www.nec.com/en/global/solutions/hpc/sx/vector_engine.html, (Accessed on 02/14/2020).

[3] K. Komatsu, S. Momose, Y. Isobe, O. Watanabe, A. Musa, M. Yokokawa, T. Aoyama, M. Sato, and H. Kobayashi, “Performance evaluation of a vector supercomputer sx-aurora tsubasa,” in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 685–696.

[4] O. Watanabe, Y. Hougi, K. Komatsu, M. Sato, A. Musa, and H. Kobayashi, “Optimizing memory layout of hyperplane ordering for vector supercomputer sx-aurora tsubasa,” in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 2019, pp. 25–32.

[5] “Programming on vector machines,” https://www.hpc.nec/api/v1/forum/file/download?id=XPlhd4, (Accessed on 02/17/2020).

[6] D. H. Bailey, “Unfavorable strides in cache memory systems (rnr technical report rnr-92-015),” *Scientific Programming*, vol. 4, no. 2, pp. 53–58, 1995.

[7] G. D. P. Shuang Gao, “Optimizing cuda shared memory usage,” in *SC15 Poster*, 2015.

[8] R. Takaki, K. Yamamoto, T. Yamane, S. Enomoto, and J. Mukai, “The development of the upacs cfd environment,” in *International Symposium on High Performance Computing*. Springer, 2003, pp. 307–319.

[9] “The openmp api specification for parallel programming,” https://www.openmp.org/, (Accessed on 02/14/2020).

[10] H. Takizawa, T. Yamada, S. Hirasawa, and R. Suda, “A use case of a code transformation rule generator for data layout optimization,” in *Sustained Simulation Performance 2016, pp.21-30, Springer*, 2016.

[11] H. C. Edwards, C. R. Trott, and D. Sunderland, “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.