# A Vectorization Technique of Hashing and its Application to Several Sorting Algorithms

Yasusi Kanada*
Central Research Laboratory, Hitachi Ltd.
Kokubunji, Tokyo 185, Japan.

## Abstract

This paper presents a vectorized algorithm for entering data into a hash table. A program that enters multiple data could not be executed on vector processors by conventional vectorization techniques because of data dependences. Our method enables execution of multiple data entry by conventional vector processors and improves the performance by a factor of 12.7 when entering 4099 pieces of data on the Hitachi S-810, compared to the normal sequential method. This method is applied to the address calculation sorting and the distribution counting sort, whose main part was unvectorizable by previous techniques. It improves performance by a factor of 12.8 when $n = 2^{14}$ on the S-810.

## 1. Introduction

Hashing is a fast and widely-used method for entering data and searching for it. A processing of entering *one* piece of data to a hash table or searching it for *one* piece of data is quite fast and there seems to be no need for parallel processing nor possibility of it. However, if a lot of data to be entered or searched for are given, parallel processing seems necessary for performance improvement and it seems possible.

Vector processing by pipelined vector processors, such as the Hitachi S-810 or Cray 2, or SIMD parallel processors, such as CM-2 (Connection Machine 2), is a promising method of parallel processing. In most cases, a program must be transformed to a sequence of vector operations in order to execute it by vector processing. This program transformation is called *vectorization*. It is possible to vectorize a program that *searches* a hash table for multiple pieces of data by previous vectorization techniques [Kuc 81] if the program is written in an appropriate style, because there are no data dependences unsuitable for vectorization [Kuc 81] among the processing of each piece of data. However, in the case of data *entry* into a hash table, there are data dependences intrinsically unsuitable for vectorization, thus it is impossible to be compiled for vector processors by conventional methods. These data dependences are caused by collisions. The detailed reason of unvectorizability will be explained in Section 2.

The address calculation sorting [Flo 60] [Gon 84] is one of the fastest sorting algorithms for sequential processing. When the distribution of data values is known, the data can be sorted in $O(n)$ in average by this method [Gon 84]. If this algorithm is vectorized, it may be the fastest algorithm for vector processing when certain conditions hold, because the order of execution time of the address calculation sorting is lower than most of the previous vectorized and parallelized sorting algorithms [Sto 78] [Bro 81] [Fla 83] [Roe 87] [Ish 88] (In the case of SIMD processors, the execution time of the vectorized address calculation sort will be $O(1)$). However, the main part of this algorithm cannot be vectorized by previous vectorization techniques, because the main part "hashes" the data and enters them into a work table, producing collisions that may cause data dependences unsuitable for vectorization [Ish 88]. Therefore, this algorithm will be vectorized if and probably only if multiple data entry into a hash table is vectorized. There are other sorting algorithms, such as the distribution counting sort, that could not be vectorized for the same reason.

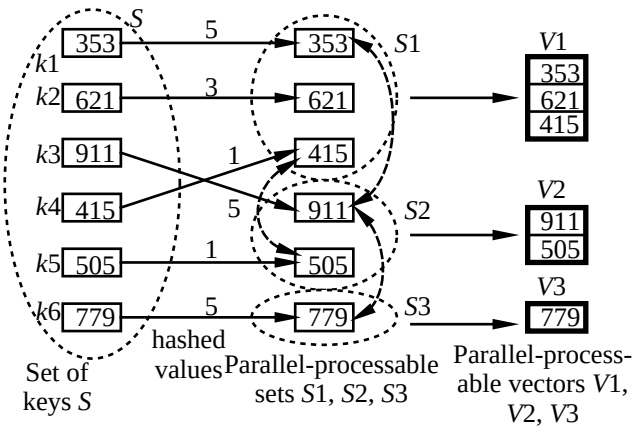
This paper presents a vectorization technique of hashing, the algorithm for data entry into a hash table, and its application to an address calculation sorting and the distributing counting sort. These algorithms require no special hardware for hashing nor sorting. Section 2 describes the vectorization technique and the vectorized algorithm of data entry. Section 3 describes the vectorized algorithm of an address calculation sorting. Section 4 describes the results of performance evaluations of both the vectorized hashing and the above sorting algorithms.

## 2. Vectorized Algorithm of Hashing

We will first explain the reason why a program that enters multiple data into a hash table is not vectorizable by conventional vectorization techniques. When entering data, the hashed values of the data are calculated first. Some pieces of data may have equal hashed values, namely *collision* may occur. There are data dependences among the processings of colliding data because each processing reads and rewrites the same entry of the hash table. Therefore, these processings must be performed sequentially.

However, there is a possibility of vector processing. Uncolliding data can be processed in parallel. So, if colliding data are detected and their entries are suppressed or are made ineffective, the rest of the data can be entered by vector processing. The colliding data can be processed later in the same manner.

**Figure 1** explains the above process conceptually. There are six keys, $k1$ through $k6$, to be entered in the hash table in this example. The hashed values of keys $k1$, $k3$ and $k6$ are the same, and those of keys $k4$ and $k5$ are the same, but others are different. Because the hashed values of keys $k1$, $k2$ and $k4$ are different, they can be processed in parallel, so they are classified to set $S1$. The hashed value of key $k3$ is the same as that of $k1$, and that of key $k5$ is the same as that of $k4$, so $k3$ and $k5$ cannot be processed in parallel with the keys in $S1$. However, they can be processed in parallel each other. So they are classified to set $S2$. The hashed value of key $k6$ is the same as that of $k1$ and $k3$, so it cannot be processed in parallel with the keys in neither $S1$ nor $S2$, so it is classified to set $S3$. If the elements in $S1$, $S2$ and $S3$ are organized into vectors as shown in Figure 1, each of these vectors can be processed using vector operations.




Broken arrows show the data dependences unsuitable for parallel processing.

Figure 1: Possibility of parallel processing in multiple data entry into a hash table

* The author's current address is as follows: Center for Machine Translation, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA. The e-mail address is yk@a.nl.cs.cmu.edu.

**Figure 2** shows the vectorized version of an open-hashing algorithm [Knu 73] that enters colliding keys into different entries of the hash table. A chain-hashing algorithm [Knu 73] that enters colliding keys into an entry as a chain of keys can be vectorized using the same technique, but it is not shown in this paper. In this algorithm, vectors 2 and 3 in Figure 1 will not be generated explicitly, but it conceptually implements the above method. Normally, a key is entered with a piece of data into a hash table, but this is omitted in the algorithm in Figure 2, and the values of all the keys are asserted to be different because of simplicity. We will argue about the elimination of this assertion in the next section. Each unused entry of the hash table is initialized to a special value *unentered*, which is not used as a key value. Value *unentered* is used for the purpose of displaying whether the entry is used or not.

The algorithm is written in a language with the parallel array assignment statement, and the **where** statement such as that of Fortran 8X. Each assignment in each parallel array assignment statement may be performed in parallel. However, no two statements can be executed in parallel, if the parallel execution may cause a wrong result. For example: If $A$ = (1, 2, 3), $B$ = (10, 11, 12), and $M$ is a *mask vector*, which is a boolean vector used for controlling vector operations, and the value is (*true*, *false*, *true*), the following statement updates the value of $A$ as (10, 2, 12);

**where** M **do** A := B; **end where**;

This language also has the *countTrue* function and the **where** operator. If $M$ is a mask vector, expression *countTrue*($M$) returns the number of trues in $M$. For example, if $M$ is array (*true*, *false*, *true*), then *countTrue* returns 2. Expression $A$ **where** $M$ means a vector of elements of $A$ which corresponds to *true* elements of $M$. For example, if $A$ = (1, 2, 3) and $M$ = (*true*, *false*, *true*), $A$ **where** $M$ returns (1, 3). Expression $A[x : y]$ in Figure 2 means a slice (subarray) of $A$, ($A[x]$, $A[x+1]$, …, $A$[y]).

In Figure 2, keys are stored into the hash table at Statements 2.1 and 2.4. However, if there are collisions between these keys, some stored keys are overwritten by other keys. So the entries to which these keys are written are checked at Statement 2.2. If table entry *table*[*hashedValue*[$i$]] is overwritten in Statement 2.1 after *key*[$i$] was assigned, condition *key*[$i$] = *table*[*hashedValue*[$i$]] does not hold. That means *key*[$i$] was not really entered. The above condition holds only when the value of *key*[$i$] is stored and kept in *table*[*hashedValue*[$i$]], because of the assumption that each value of the key is unique. The keys are used as *identifiers*; they can be used as such because of the above assumption that each is unique. The above check can be done by "vector-compare" operation of vector processors. It generates a mask vector. Then, the unentered elements are collected at Statement 2.3. They are stored into other entries of the hash table at Statement 2.4. The collection is done by a vector-compress (compressing store) operation in vector processors such as the S-810 [Nag 84]. The above process is repeated until all the keys are entered.

The above check and reassignment guarantee that this algorithm enters all the keys correctly regardless of order of assignments, even if there are collisions. That means each element in statements 2.1 and 2.4 can be assigned completely in parallel. No serialization is necessary. We will call the technique used for vectorizing the hashing algorithm the *overwrite-and-check* technique. A very closer technique is implicitly used in Appel and Bendiksen's vectorized garbage collection algorithm [App 89].

**Figure 3** shows an execution example of the program in Figure 2. For simplicity's sake, the hash table in the example is small. The hashing function *hash*($x$) used here is $x$ **mod** 6. The keys to be entered are 353, 621, 415 and 911, so the input of the algorithm is an array contains these values. There is an entered key, 103, in the hash table at the beginning.

The execution process will be explained below. The hashed value of each key is computed first. Then, the *first type of collisions*, collisions between the keys to be entered and the keys already entered in the hash table, are checked at the condition expressions of the **where** statements in Figure 2. In this example, only the third element, 415, is colliding, so the third element of the mask vector becomes false and other elements become true. Other keys, 353, 621 and 911 are entered into the hash table. Keys 353 and 911 are written into the same table entry. Key 911 overwrites key 353 here because the hashed values are the same.

The *second type of collisions*, that is, collisions among elements to enter, is detected in the next step. Key 353 is detected as a colliding key. As the result, keys 353 and 415 are detected as unentered keys, and they are re-packed (compressed) into the array of keys. Hashed values are also re-packed and used for calculating the alternative hashed values. The two types of collisions are checked for the new hashed values. No collision occur in this time, so all the keys are successfully entered, and the iteration of the for loop stops.

Most of the above operations can be performed using vector operations efficiently in vector processors such as the S-810.

## 3. Application to Sorting

There is a variation of address calculation sorting that is called the *linear probing sort* [Gon 84]. This algorithm uses a work array, $C$. Data are "hashed" and stored into $C$. The "hashing function" has the following property.

$$data[i] \le data[j] \Rightarrow hash(data[i]) \le hash(data[j]) \; ; \quad (1 \le i \le n,\ 1 \le j \le n).$$

Because of this property, it is not really a hashing function, but *overwrite-and-check* technique can be applied in the same way. The order of data stored in array $C$ is sorted because of this function, if it is not disordered by the processing of colliding data. The data in $C$ are not contiguously stored, so they are packed into another array. The original array, *data*, can be used for this purpose. The algorithm is shown in Appendix.

---

**input** *table* : the hash table.
*key*[1 : *n*] : A set of keys to be entered {only keys are entered in this algorithm}.
**output** *table* : the hash table which *key*[1 : *n*] are entered in.

---

```
local   hashedValue[1 : n], entered[1 : n].          /* local variables */
/* Computing hashed values and entering data into the table */
hashedValue[1 : n] := hash(key[1 : n]);
        /* calculate hashed values {for example, hash(x) = x mod size(table)}.
where table[hashedValue[1 : n]] = unentered do
        /* detection of confliction among the data to enter and */
        /* already entered data. */
    table[hashedValue[1 : n]] := key[1 : n];                          (2.1)
        /* enter the keys only where unentered. More than one */
        /* data may be written in a hash table entry. */
end where;

for i in 1 .. size(table) loop
/* Checking unentered elements and collecting them */
    entered[1 : n] := (key[1 : n] = table[hashedValue[1 : n]]);
    nrest := countTrue(entered[1 : n]);                               (2.2)
        /* count number of trues in boolean array 'entered'. */
    hashedValue[1 : nrest] := hashedValue[1 : n] where not entered[1 : n];
        /* pack unentered elements in hashedValue[1 : n]. */
    key[1 : nrest] := key[1 : n] where not entered[1 : n];            (2.3)
        /* pack unentered elements in key[1 : n]. */

/* Testing whether data entry is finished */
    if nrest = 0 then exit loop;    /* exit the for-loop */
    n := nrest;

/* Computing the subscripts for the next step and entering data */
    hashedValue[1 : n] := (hashedValue[1 : n] + 1) mod size(table);
    where table[hashedValue[1 : n]] = unentered do
        table[hashedValue[1 : n]] := key[1 : n];                      (2.4)
            /* enter the keys only where unentered. More than one */
            /* data may be written in a hash table entry. */
    end where;
end loop;
```

Figure 2: The vectorized algorithm for entering data into a hash table

(omitted)

Figure 3: An example of the vectorized data entry into a hash table

This algorithm can be vectorized using the *overwrite-and-check* technique. **Figure 4** shows the vectorized algorithm. The data to be sorted are asserted to be non-negative here. This program consists of six parts, A through F. They correspond to the same name parts in the scalar algorithm except E, which is specific to vector processing which is based on the *overwrite-and-check* technique. The new data are inserted into the sorted array *C* in part C. However, if old data are already stored in the places, they are saved to array *work* in part C and restored to the next available places of *C* in part D.

There are two major differences between the hashing used in this address calculation sorting and the simple hashing shown in Section 2. One difference is that same values (keys) are allowed for input data in this algorithm. Thus data values cannot be used as identifiers when checking the collisions, so the negated indices of data (—*ι* in Figure 4) are used instead. The data are stored after *overwrite-and-check* in the same place where their indices were stored when checking. The assertion that the data are non-negative is necessary because of the sharing of array between identifiers and data to be sorted, but this assertion can be eliminated if a different array is used for each purpose.

The other difference is the processing of colliding data. Colliding data must be inserted to an appropriate place in the sequence of sorted data. All the colliding data are attempted to insert in parallel using vector operations. Each unused entry of *C* is initialized to a special value *unentered* which is greater than any data value. This makes the above insertion possible.

**Figure 5** shows an example of the address calculation sorting process comparing to the original sequential sorting process. Though this algorithm uses a lot of local arrays, the size of these arrays, except *C*, can be remarkably reduced by an optimizing transformation. However, the size of *C* must be at least twice as large as *n*.

The distribution counting sort [Knu 73] can also be vectorized using the *overwrite-and-check* technique. The vectorized distribution counting sort algorithm is not shown here because of page limitations.

```
input   A[1 : n]:   array to sort {the element values should be in [0, Vmax)}.
output  A[1 : n]:   sorted array.

local   C[0 : 3*n — 1], uninsertable[1 : n], work[1 : n], entered[1 : n],
        toShift[1 : n], index[1 : n], next[1 : n], nonempty[1 : n].

C[0 : size(C) — 1] := unentered; /* initialize C  (unentered = Vmax) */

/* A. Computing "hashed" values */
hashedValue[1 : n] := int(float(2 * size(C) * A[i]) / Vmax);  nrest := n;

repeat
 /* B. Finding the table entries to insert the data */
    repeat
        uninsertable[1 : nrest] := (C[hashedValue[1 : nrest]] ≤ A[1 : nrest]);
            /* check the first type of collision with stored data. */
            /* If hashedValue[i] ≠ unentered, the right-hand side */
            /* condition holds, i.e., there is a first type of collision. */
        Nuninsertable := countTrue(uninsertable[1 : nrest]);
            /* count the number of uninsertable (colliding) data. */
        where uninsertable[1 : nrest] do
            hashedValue[1 : nrest] := hashedValue[1 : nrest] + 1;
        end where;
    until Nuninsertable = 0;
            /* repeat until there is no first type of collision. */
/* C. Inserting the data */
    work[1 : nrest] := C[hashedValue[1 : nrest]];
        /* save the original values of C to work. */
    C[hashedValue[1 : nrest]] := —ι;
        /* store the identifiers to check {—ι is array (—1, —2, …, —nrest)}. */
        /* An entry of C may be written twice or more (overwritten). */
    entered[1 : nrest] := C[hashedValue[1 : nrest]] = —ι;
        /* check the second type of collision between newly entered data. */
    where entered[1 : nrest] do
        C[hashedValue[1 : nrest]] := A[1 : nrest];
    end where; /* enter */
/* D. Shifting the work array elements {only for successfully inserted data} */
    toShift[1 : nrest] := entered[1 : nrest] and (work[1 : nrest] ≠ unentered);
    NtoShift := countTrue(toShift[1 : nrest]);
    work[1 : NtoShift] := work[1 : nrest] where toShift[1 : nrest];
    index[1 : NtoShift] := (hashedValue[1 : nrest] + 1)
            where toShift[1 : nrest];
    while NtoShift > 0 do
        next[1 : NtoShift] := C[index[1 : NtoShift]];
        C[index[1 : NtoShift]] := work[1 : NtoShift];
        nonempty[1 : NtoShift] := (next[1 : NtoShift] ≤ unentered;
        count  := countTrue(nonempty[1 : NtoShift]);
        work[1 : count] := next[1 : NtoShift]
                where nonempty[1 : NtoShift];    /* pack work. */
        index[1 : count] := index[1 : NtoShift] + 1
                where nonempty[1 : NtoShift];    /* pack index. */
        NtoShift := count;
    end while;
/* E. Collecting the uninserted data for the next iteration */
    irest := countTrue(not entered);
    hashedValue[1 : irest] := hashedValue[1 : nrest]
            where not entered[1 : nrest];
    A[1 : irest] := A[1 : nrest] where not entered[1 : nrest];
    nrest  := irest;
until nrest = 0;        /* until all the data are inserted. */
/* F. Packing the sorted data into A */
A[1 : n] := C[0 : size(C) — 1] where (C[0 : size(C) — 1] ≠ unentered);
```

Figure 4: The vectorized algorithm of the address calculation sorting

| | *A* (array to sort) | | | | *C* (work array) | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Step | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 Index |
| 0. | ( | 38 | 11 | 42 | 39) ( | * | * | * | * | * | * | * | * | * | * | *) The range of the keys is [0, 100). *hash*(*x*) = [(8/100)*x*]. (* |

= *Vmax*).

| | | |
|---|---|---|
| 1. | (\|38 11 42 39) ( * * * 38 * * * * * * * *) | *hash*(38) = 3. |
| 2. | ( 38\|11 42 39) (11 * * 38 * * * * * * * *) | *hash*(11) = 0. |
| 3. | ( 38 11\|42 39) (11 * * 38 42 * * * * * * *) | *hash*(42) = 3. Entered to *C*[4] because *C*[3] < 39 < *C*[4]. |
| 4. | ( 38 11 42\|39) (11 * * 38 39 42 * * * * * *) | *hash*(39) = 3. 42 is shifted because *C*[3] < 42. |
| | ⇒ shifted | |
| 5. | ( 11 38 39 42) | The sorted result is packed into *A*. |

(a) The sequential version

| Step | Phase† | *A* (array to sort) 1 2 3 4 | *C* (work array) 0 1 2 3 4 5 6 7 8 9 10 11 Index | |
|---|---|---|---|---|
| 0. | A | (38 11 42 39) | ( * * * * * * * * * * * *) | The range of the keys is [0, 100). *hash*(*x*) = [(8/100)*x*]. (* = *Vmax*). |
| 1. | B&C1 | (38 11 42 39) | (−2 * * −4 * * * * * * * *) | Store the negated indices of *A* to *C*. —1, —3 and —4 are written to *C*[3]. |
| 2. | C2 | (38 11 42 39) | (11 * * 39 * * * * * * * *) | Store the data whose indices are succeeded to store into *C*. |
| 3. | D&E | (38 42) | (11 * * 39 * * * * * * * *) | Collect the data that are failed to store into *C*. |
| 4. | B&C1 | (38 42) | (11 * * -1 -2 * * * * * * *) | Store the negated indices of *A* to *C*. 39 is saved to *work*[1]. |
| 5. | C2 | (38 42) | (11 * * 38 42 * * * * * * *) | Store the data whose negated index is successfully stored into *C*. |
| 6. | D1 | (38 42) | (11 * * 38 39 * * * * * * *) | 42 is saved to *next*[1] and 39 is stored to *C*[4]. |
| 7. | D2&E | (38 42) | (11 * * 38 39 42 * * * * * *) | 42 is stored to *C*[5]. |
| 8. | F | (11 38 39 42) | | The sorted result is packed into *A*. |

† Indicated in Figure 4.

(b) The vectorized version

Figure 5: An example of the sequential and vectorized address calculation sorting

## 4. Performance Evaluation

This section presents a performance model of the vectorized hashing, the result of performance evaluation of hashing and sorting using the *overwrite-and-check* technique.

### 4.1 A Performance Model

We will show an execution model of vectorized hashing and the performance model. If the average number of hash table accesses for a key is $c$, and the number of keys to enter is $n$, the scalar execution time, $t_s$, is modeled as follows:

$$t_s = acn,$$

where $a$ is a constant whose value is dependent on implementation. If the number of hash table access is $c'$ for the worst case in the $n$ keys, the vector execution time, $t_v$, is modeled as follows:

$$t_v = a'cn + b'c',$$

where $a'$ and $b'$ are constants whose values are implementation dependent. The value of constant $a'$ should be smaller than that of $a$ for the sake of performance improvement. The number of iterations performed by the for loop in Figure 2 is equal to $c'$. The first term of the vector execution time is the part that is proportional to the vector length. The second term is the overhead of vector execution. The overhead is independent of the vector length, and it is nearly proportional to the number of vector instructions. The acceleration ratio, $\alpha$, is the quotient of the scalar time by the vector time:

$$\alpha = 1 / \left( \frac{a'}{a} + \frac{b'c'}{acn} \right).$$

### 4.2 Evaluation of Hashing

**Figures 6** and **7** show the result of evaluation of multiple data entry into a hash table on the S-810 model 20. The hash table is initially empty, and various number of uniformly random keys are entered. Two hash table sizes, 521 and 4099, are tested. The results of the former are shown in Figure 6, and that of the latter in Figure 7. The horizontal axis in Figures 6(a) and 7(a) shows the load factor *after* all the keys are entered. The load factor is the ratio of filled entries in the hash table. Figures 6(a) and 7(a) show the time to enter all the keys in an array, and Figures 6(b) and 7(b) show the acceleration ratios. An optimized version of the algorithm in Figure 2 is used for this performance evaluation.

The peak acceleration ratios, 5.6 and 12.7, are obtained when the load factor is between 0.1 and 0.5. The reason why the acceleration ratio is lower when the load factor is less than 0.1 is that the vector length is not enough for using the vector pipeline of the S-810 efficiently. The reason why the acceleration ratio is lower when the load factor is more than 0.5 can be explained as follows: when the load factor is increased, $c'/c$ is also increased because $c'$ increases faster than $c$. Then the second term of $t_v$ is increased and the acceleration ratio is decreased.

(omitted)

(a) The execution time (b) The acceleration ratio

Figure 6: The vector execution time and the acceleration ratio of multiple data entry ($n$ = 521)

(omitted)

(a) The execution time (b) The acceleration ratio

Figure 7: The vector execution time and the acceleration ratio of multiple data entry ($n$ = 4099)

(omitted)

(a) $n$ = 521 (b) $n$ = 4099

Figure 8: The acceleration ratio of multiple data search

Performance of multiple data *search* is also evaluated. The result is shown in **Figure 8**. The acceleration ratio is 5 to 10 in most cases when the vector length is enough and the collision rate is not high. The acceleration ratio is lower than that of the data entry, because there are more arrays to re-pack in the case of search.

### 4.3 Evaluation of Sorting

Both the address calculation sorting and the distribution counting sort are evaluated. **Table 1** shows the result. The acceleration ratio of the address calculation sorting is 7.65 when $n = 2^{10}$ and it is 12.84 when $n = 2^{14}$.

Table 1: The CPU time and the acceleration ratio of vectorized sorting algorithms

| Algorithm | $n$ | S-810/20 CPU time (μs) | | Acceleration ratio |
|---|---|---|---|---|
| | | Sequential | Vectorized | |
| Address Calculation Sorting* | $2^6$ | 289 | 110 | 2.62 |
| | $2^{10}$ | 4,286 | 560 | 7.65 |
| | $2^{14}$ | 66,955 | 5,215 | 12.84 |
| Distributio n Counting Sort** | $2^6$ | 12,206 | 1,522 | 8.02 |
| | $2^{10}$ | 13,072 | 1,738 | 7.52 |
| | $2^{14}$ | 30,089 | 5,667 | 5.31 |

*The size of the work array $C$ is $3n$.
**The size of the work array is $2^{16}$, which is the range of the data.

## 5. Conclusion

The vectorization technique of hashing, presented in this paper and called the *overwrite-and-check* technique, enables execution of multiple data entry by vector processing and improves performance by a factor of 12.7 when entering 4099 pieces of data on the Hitachi S-810, compared to the conventional sequential method. Applied to the address calculation sorting, this technique improves performance by a factor of 12.8 when $n = 2^{14}$ on the S-810.

The application of the vectorized hashing is limited because performance is improved only when many pieces of data are entered or searched at once. However, in applications such as the address calculation sorting, the performance is remarkably improved. The *overwrite-and-check* technique may become a general technique for pipelined vector processors and SIMD parallel processors, and they may be helpful for reducing locked time or serialized accesses of shared resources among processing units in parallel processing systems.


## Acknowledgement

The author wishes to thank Dr. Sakae Takahashi and Seiichi Yoshizumi of Hitachi Ltd. for their continuing support of his research, and to thank Dr. Michiaki Yasumura and Masahiro Sugaya of Hitachi Ltd. for their useful comments.


## References


[App 89] Appel, A. W., and Bendiksen, A.: Vectorized Garbage Collection, *J. Supercomputing*, Vol. 3, pp. 151-160, 1989.

[Bro 81] Brock, H. K., Brooks, B. J., and Sullivan, F.: DIAMOND: A Sorting Method for Vector Machines, *BIT*, Vol. 21, pp. 142-152, 1981.

[Fla 83] Flanders, P. M., and Reddaway, S. F.: Sorting on DAP, *Parallel Computing 83*, pp. 247-252, Elsevier Science Publishers B. B., North-Holland, 1984.

[Flo 60] Flores, I.: Computer Times for Address Calculation Sorting, *J. ACM*, Vol. 7, No. 4, pp. 389-409, 1960.

[Gon 84] Gonnet, G. H.: Handbook of Algorithms and Data Structures, Addison-Wesley, 1984.

[Ish 88] Ishiura, N., Takagi, N., and Yajima, S.: Sorting on Vector Processors, *Transaction of Association of Information Processing*, Vol. 29, No. 4, pp. 378-385, 1988 (in Japanese).

[Knu 73] Knuth, D. E.: Sorting and Searching, *The Art of Computer Programming*, Vol. 3, Addison-Wesley, 1973.

[Kuc 81] Kuck, D. J., Kuhn, R. H., Pauda, D. A., Lease, B. R., and Wolfe, M. J.: Dependence Graphs and Compiler Optimizations, *Eighth ACM Symposium on Principles of Programming Languages*, pp. 207-218, 1981.

[Nag 84] Nagashima, S., et al.: Design Consideration for High-Speed Vector Processor: S-810, *Proc. IEEE International Conference on Computer Design*, pp. 238-242, 1984.

[Roe 87] Roensch, W., and Strauss, H.: Timing Results of Some Internal Sorting Algorithms on Vector Computers, *Parallel Computing*, Vol. 4, pp. 49-61, 1987.

[Sto 78] Stone, H. S.: Sorting on STAR, *IEEE Transaction on Software Engineering*, Vol. 4, No. 2, pp. 138-146, 1978.


## Appendix: The Scalar Address Calculation Sorting Algorithm

---

**input** $A[1 : n]$: array to sort {the element values should be in [0, *Vmax*)}.
**output** $A[1 : n]$: sorted array.

---

```
local   C[0 : 3*n — 1];
for i in 0 .. size(C) — 1 loop C[i] := unentered; end loop; /* Initialize C. */
/* Scatter the data into C: */
for i in 1 .. n loop
/* A. Computing a "hashed" value of A[i]. */
    hashedValue := int(float(2 * size(C) * A[i]) / Vmax);
/* B. Finding the table entry to insert the new data A[i]: */
    while C[hashedValue] ≤ A[i] loop
        hashedValue := hashedValue + 1;
    end while;
/* C&D. Inserting the new data and shifting the data in C: */
    w := C[hashedValue]; C[hashedValue] := A[i];
    while w ≠ unentered loop
        hashedValue := hashedValue + 1;
        x := C[hashedValue]; C[hashedValue] := w; w := x;
    end while;
end for;
/* F. Packing the sorted data into A. */
count := 0;
for i in 0 .. size(C) — 1 loop
    if C[i] ≠ unentered then
        count := count + 1; A[count] := C[i];
    end if;
end for;
```