VSkyline: Vectorization for Efficient Skyline Computation

Sung-Ryoung Cho[†] Jongwuk Lee[‡] Seung-Won Hwang[‡]
Hwansoo Han[†] Sang-Won Lee[†]

[†]School of Information & Communications Engineering
Sungkyunkwan University
Suwon, 440-746, Korea
{applys,hhan,swlee}@skku.edu

[‡]Department of Computer Science and Engineering
Pohang University of Science and Technology
Pohang, 790-784, Korea
{julee,swhwang}@postech.edu

ABSTRACT

A dominance test, which decides the *dominance* relationship between tuples, is a core operation in skyline computation. Optimizing dominance tests can thus improve the performance of all existing skyline algorithms. Towards this goal, this paper proposes a *vectorization* of dominance tests in *SIMD architectures*. Specifically, our vectorization can perform the dominance test of multiple consecutive dimensions in parallel, thereby achieving a speedup of *SIMD parallelism degree* in theory. However, achieving such performance gain is non-trivial due to complex control dependencies within the dominance test. To address this problem, we devise an efficient vectorization, called VS l ne, which performs the dominance test with SIMD instructions by determining *incomparability* in a block of four dimensional values. Experimental results using a performance monitor show that VS l ne considerably reduces the numbers of both executed instructions and branch mispredictions.

1. INTRODUCTION

For the past decade, skyline queries [2, 3, 4, 5, 6, 7, 11] have gained considerable attention for helping multi-criteria decision-making processes in a large-scale data set of tuples. Because skyline computation depends heavily on tuple-wise comparisons to check the *dominance* relationship between tuples, called *dominance tests*, existing skyline algorithms tend to focus on avoiding unnecessary dominance tests either by pruning non-skyline candidates early or by partitioning an entire data set into multiple subsets.

Despite such optimizations, a huge number of dominance tests is still inevitable in skyline computation. In particular, as dimensionality increases, the number of dominance tests increases exponentially. Since the dominance test is a core operation in skyline computation, we aim to optimize the dominance test itself. Specifically, when the values of tuples are scanned sequentially for the dominance test, there exists an opportunity to take advantage of data-level parallelism in *SIMD* (*single instruction, multiple data*) architectures. Although SIMD architectures are available in almost all modern CPUs, to the best of our knowledge, no skyline algorithm makes use of SIMD architectures to achieve efficient dominance tests.

In this paper, we propose a *vectorization* of dominance tests, which can determine dominance relationship efficiently, by performing comparison for multiple consecutive dimensions in parallel. Given a SIMD parallelism degree, *e.g.*, currently four, we can boost the performance by up to four folds in theory. However, achieving such performance gain is non-trivial for skyline computations. In general, a dominance test on multiple dimensions should be performed in a sequential order for each dimension, as the result of dominance test at a specific dimension is dependent on the results of preceding dimensions and the result for multiple dimensions is thus decided at the last dimension. For these dependencies, SIMD comparison of multiple dimensions requires a series of conditional branches to inspect the comparison result at each dimension with consideration of the preceding dominance test results. As the performance of SIMD architectures can be greatly hindered by complex conditional branches [9], a naive-vectorization thus performs even worse than a non-vectorization version.

To address this problem, we devise an efficient vectorization, called VS l ne, which performs the dominance test with SIMD instructions by determining *incomparability* in a block of four dimensional values. Our performance evaluation using a performance monitor shows that VS l ne considerably reduces the numbers of both instructions including branches and branch mispredictions. We also stress that this speedup is orthogonal to existing parallelization efforts, which suggests that the existing algorithms can benefit from our finding as well.

The rest of this paper is organized as follows. Section 2 briefly reviews skyline computation and SIMD technology. Section 3 proposes an efficient vectorized skyline algorithm VS l ne, and Section 4 then validates performance evaluation of VS l ne in extensive synthetic data sets. Finally, Section 5 concludes our paper.

2. BACKGROUND

2.1 Skyline Queries

We first introduce basic notations to address skyline query problem. Let $\mathcal{D}$ be a finite d-dimensional space, *i.e.*, $\{D_1, \ldots, D_d\}$, where each dimension has a domain $dom(D_i)$ of non-negative real number $\mathbb{R}^+$. Let $\mathcal{S}$ be a set of finite n tuples as a subset of $dom(\mathcal{D})$. A tuple p in $\mathcal{S}$ is represented as $p = (p_1, \ldots, p_d)$, where $\forall i \in [1, d] : p_i \in dom(D_i)$.

Based on these notations, we formally states some notions commonly used in the skyline literature [2, 3, 4, 5, 6, 7, 11]. Throughout this paper, we consistently use *max* operator for skyline queries. Specifically, given two tuples p and q, p *dominates* q on $\mathcal{D}$ if $\forall\ i \in [1, d] : p_i \geq q_i$ and $\exists\ j \in [1, d] : p_j > q_j$, denoted as $p \succ q$. Also, it is said p and q are *incomparable* if they do not dominate each other, denoted as $p \sim q$.

Given $\mathcal{S}$, a skyline query on $\mathcal{D}$ thus returns a subset of tuples, or *skyline*, that are no worse than, or not dominated by, any other tuples in $\mathcal{S}$, *i.e.*, $\{p \in \mathcal{S} | \nexists q \in \mathcal{S} : q \succ p\}$. A tuple in skyline is called a *skyline tuple*.

The skyline computation depends heavily on dominance tests between tuples. In the worst case, a naive skyline algorithm exhaustively performs dominance tests for all possible $n(n-1)/2$ tuple pairs, incurring quadratic cost.

To address this problem, existing skyline algorithms aimed to reduce unnecessary dominance tests. Specifically, they could be classified into two categories: (1) *sorting-based* algorithms such as N [2], Inde [11], S S [3], and SS [4] focused on optimizing tuple ordering to prune more dominated tuples early on; (2) *partitioning-based* algorithms such as NN [5], S[7], Se rc [6], and OS S [13] focused on dividing an entire dataset into multiple subsets to exploit "region-level" optimization.

On the other hand, due to the CPU-intensive property of dominance tests [8, 12], skyline computation can take advantage of *parallelism* as two forms – (1) *thread-level parallelism* and (2) *data-level parallelism*.

All existing parallel skyline algorithms fall into the category of *thread-level parallelism*, distributing partitioned subsets into independent threads, and then merging local skylines into global skyline. Specifically, Vlachou et al. [12] developed angle-based data partitioning to balance the computation overhead between threads in data distribution. Recently, Park et al. [8] proposed S l ne, which optimizes merge processing to enhance thread utilization, by computing multiple dominance tests in parallel.

To the best of our knowledge, no existing skyline algorithm has attempted data-level parallelism [9], performing the same operation on multiple data simultaneously. We propose a *vectorization* of dominance tests to achieve a theoretical speedup of SIMD parallelism degree. We stress that this speedup is orthogonal to thread-level parallelism, and skyline algorithms using thread-level parallelism can thus exploit data-level parallelism as well.

2.2 SIMD Technology

Almost every modern CPU supports SIMD instructions. For instance, the x86 microprocessor provides hundreds of MMX (multi-media extensions) and SSE (streaming SIMD extensions) instructions. A SIMD instruction operates on vectors of data. Therefore, one obvious performance benefit of using SIMD instructions is to process multiple elements at a time, and we can expect a speedup of SIMD parallelism degree, denoted as V, in theory.

SIMD operations can work best when handling a great deal of identically structured data, *e.g.*, arrays in *for* loop, for data-level parallelism. For example, many database algorithms such as sequential scans, scalar aggregations, index traversals, and joins perform repetitive operations on an array of tuples (and each tuple in turn is an array of columns). Zhou and Ross [14] showed how the inner loop of those operations can benefit from using SIMD instructions. We call this transformation *vectorization* (or *SIMDization*).

Along the same line, skyline computation is also a good candidate for vectorization. It requires numerous tuple-wise dominance tests in a large set of tuples, where a dominance test sequentially accesses two tuples in array structures.

However, it is a non-trivial task to successfully vectorize dominance tests. It is well known that SIMD is at its weakest in *control* statements [9], and unfortunately complex conditional branches are innate in dominance tests. Furthermore, conditional branches are problematic in modern pipelined architectures because of serious branch misprediction penalty from instruction pipeline flush and other bookkeeping overheads ensuring operational consistency [1].

For this reason, if we can reduce or eliminate conditional branch instructions during vectorization, it might provide potentially larger performance gain. For example, Zhou and Ross [14] showed that they can obtain considerable performance enhancement by eliminating conditional branch instructions while vectorizing database operations, thus avoiding the branch misprediction penalty. However, the vectorization techniques are relatively simple because the result on each element is independent from other elements.

In clear contrast, in the dominance test, the result of *incomparability* at a specific dimension is also dependent on the results at previous dimensions. This property of *conditional dependency* can make a naive vectorization of dominance tests even poorer than a non-vectorized version. The obvious benefit of vectorization with SIMD instructions is the decrease of loop iterations, resulting in less number of executed instructions. A naive vectorization, however, can require more extra instructions to manipulate the values in SIMD registers, which negates the benefit of SIMD vectorization. To address this problem, we can limit the number extra instructions and make complex conditional control flow more predictable. As a result, the total number of executed instructions and mispredicted branches are reduced, which ultimately improves the performance. The next section discusses an efficient vectorization of dominance tests.

3. SKYLINE COMPUTATION

This section first overviews a non-vectorized dominance test in S l ne, and shows that a naive vectorization is inefficient to handle complex conditions resulting from SIMD comparisons. We then propose an efficient vectorization for dominance tests, called VS l ne, which optimizes the complex condition tests in the dominance test. We apply our vectorization technique to S l ne. Since data-level parallelism in VS l ne is orthogonal to thread-level parallelism in S l ne, we can further improve the performance for S l ne by using our vectorization technique.

Algorithm 1 S l ne dominance test

```
1: $Dom \leftarrow Incomparable$; $i \leftarrow 1$;
2: while $i \leq d$ do
3:     if $p_i < q_i$ then
4:         if $Dom = Left$ then
5:             return $Incomparable$;
6:         end if
7:         $Dom \leftarrow Right$;
8:     else if $p_i > q_i$ then
9:         if $Dom = Right$ then
10:            return $Incomparable$;
11:        end if
12:        $Dom \leftarrow Left$;
13:    end if
14:    $i \leftarrow i + 1$;
15: end while
16: return $Dom$;
```

3.1 PSkyline Overview

S l ne [8] extends skyline computation leveraging thread-level parallelism, by dividing an entire dataset into multiple subsets and merging local skylines from each thread. Specifically, the *map* process is to distribute the dataset to threads and compute each local skyline. Then, *reduce* process is to merge local skylines into a global skyline in parallel. A key contribution of S l ne is to optimize thread utilization for both *pmap* (*map* process) and *pmerge* (*reduce* process). Conceptually, *pmap*, among the tuples assigned to each thread, prunes out some of non-skyline tuples that cannot be contained in a final skyline by using *partial sorting.* Also, *pmerge* aggregates the local results, and returns the final skyline by maximizing thread-level parallelism. These two modules, however, rely on the classic pair-wise dominance test.

To illustrate, Algorithm 1 describes the dominance test in S l ne as well as most of the existing skyline algorithms. Given two tuples p and q, p_i and q_i represent the i-th dimensional values. Typically, the dominance test starts with comparing p_1 and q_1 values, iterating the loop until the dominance relationship is decided.

Specifically, in an iteration of the loop, Dom is set to either $Left$ or $Right$. If the result of the current iteration is different from that of the previous iteration, $Incomparable$ is returned. That is, even in the best case, the result is obtained at the second iteration. Returning $Incomparable$ means two tuples are incomparable. In such case, any further comparison on the rest of the dimensions is unnecessary.

We could see that a single dominance test includes quite a few conditional branches, which are particularly unpredictable. Since comparison results of dimensional values can be any, the condi-

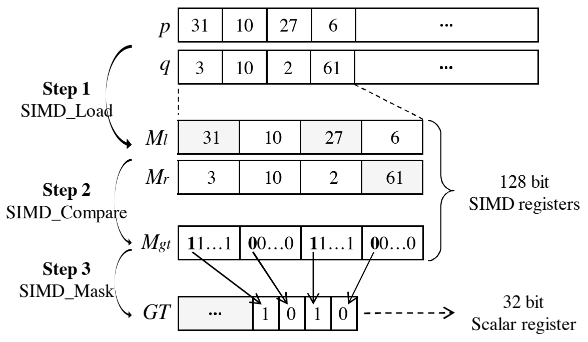


Figure 1: SIMD operations

tional statements (lines 3-13) in Algorithm 1 can flow any direction. This actually makes conditional branches in the dominance test unpredictable. This unique characteristic of the dominance test suggests that modern CPUs with deep pipelines will suffer from branch mispredictions, which significantly degrade the performances. Indeed, it is commonly believed that even 5% branch misprediction can degrade performance by as much as 20∼30% in modern CPUs [14]. We also observe similar symptoms from S l ne. The mispredicted branches account for about 15∼20% of the CPU cycles for S l ne.

3.2 Vectorization and Packed Conditions

Typical dominance tests proceed sequentially one dimension by one dimension as in Algorithm 1. To vectorize dominance tests, we need to compare four dimensions simultaneously with SIMD instructions, where SIMD parallelism degree is four. Thus, the comparison results across four dimensions are stored in a SIMD register. Since the result of the dominance test has a dependence on the comparison result of previous dimensions, interpreting the SIMD comparison result requires extra instructions to manipulate SIMD registers.

Figure 1 illustrates the steps of the vectorized dominance test over four dimensions. SIMD_Load, SIMD_Compare, and SIMD_Mask operations are performed in order and four bits are computed as a final summary of SIMD comparison. SIMD registers, M_l, M_r, and M_{gt} are 128 bit long each. A general register, GT is 32 bit long. Each SIMD register can keep four elements, each of which is 32 bit long type (e.g. `int` or `float`). SIMD register M_{gt} contains the result of *greater-than* comparison. If element-wise comparison is true, all 32 bits for that element are set to 1s. Otherwise, all 32 bits are set to 0s. SIMD_Mask summarizes the SIMD comparison result to four bits by taking the most significant bit of each element. Those four bits are stored in the lower four bits of GT register. Throughout this paper, we call those four bits *significant bits*.

	1	2	3	4
$GT(p)$	1	0	1	0
$GT(q)$	0	0	0	1
Dom	L	L	L	I

Figure 2: A dominance test in NaiveSIMD

Interpreting the significant bits for the dominance test can be implemented in a naive way (N eSIMD) or an optimized way (VS l ne). To evaluate incomparability between two tuples, we need to examine the result of each dimension comparison, and determine the dominance result with consideration of the previous dimensions. Because comparison results of four dimensions are packed into significant bits, the conditions to decide the dominance result are also packed into those significant bits. We call this situation *packed conditions*. In the next section, we discuss why a naive interpretation of packed conditions causes performance loss, and propose an optimized interpretation for packed conditions.

3.3 Vectorized Dominance Test

3.3.1 NaiveSIMD

Figure 2 illustrates a naive vectorization of the dominance test for two tuples shown in Figure 1. Bits in $GT(p)$ are set when p is greater than q and bits in $GT(q)$ are set when q is greater than p. *Dom* shows the intermediate result of the dominance test up to the corresponding dimension. L, R, and I represent *left-dominate*, *right-dominate*, and *incomparable*, respectively. N eSIMD uses two comparison results and computes dominance results with the same control logic as Algorithm 1. Examining the comparison results one dimension by one dimension, N eSIMD determines the dominance result by using the same control logic in Algorithm 1 except that it performs a SIMD comparison over four dimensions.

Up to third dimension, the intermediate result in *Dom* is *left-dominating (L)*. At the forth dimension, its comparison result is inconsistent with the previous dimension. Thus, the final dominance test for p and q becomes *incomparable (I)*. N eSIMD often requires many extra instructions, such as shift, bitwise logical operations, to manipulate the results stored in significant bits. If we count the number of executed instructions for the case depicted in Figure 2, N eSIMD executes 1.5x more instructions than S l ne. Moreover, S l ne suffers from a

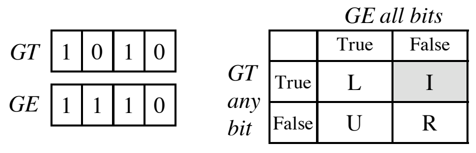


Figure 3: A dominance test in VSkyline

sizable number of mispredicted branches, which accounts for 15∼20% of execution cycles on various inputs. N   eSIMD shares the same characteristics. Thus, N   eSIMD performs worse than   S   l ne.

To maximize the benefit from SIMD vectorization, we need to devise a new dominance test logic, which can examine comparison results as a whole, not by individual dimensions. In the next section, we introduce VS   l ne, which can effectively handle packed conditions with limited number of extra instructions and far less number of branch mispredictions.

3.3.2 VSkyline

Figure 3 illustrates how VS   l ne performs the dominance test by taking the comparison results of four dimensions as a whole. Similar to N   eSIMD, we need two comparison results, but slightly different comparisons, *GT* (*greater-than*) and *GE* (*greater-than-or-equal*). Then, we decide the dominance result over four dimensions at once by using the truth table shown in the right side of Figure 3. N   eSIMD examines one dimension by one dimension, even after parallelizing compares with SIMD instructions. On the contrary, our VS   l ne examines all four dimensions as a whole.

As shown in the truth table, we examine two conditions. If any bit in *GT* is set to 1, the *GT-any-bit* becomes true. Otherwise, false. If all bits in *GE* are set to 1s, the *GE-all-bits* becomes true. Otherwise, false. Depending on two conditions, we can decide the dominance result up to four dimensions at once. L, R, I, and U represent *left-dominating*, *right-dominating*, *incomparable* and *undecided*, respectively. *Undecided* means two tuples have the same values for all dimensions so far and further test on the rest of the dimensions are required to decide the dominance result. Using the same example in Figure 1, its dominance result in VS   l ne is *incomparable*, which is the same result shown in Figure 2.

In general, the dominance test in VS   l ne categorizes packed conditions into the following four cases. Each case corresponds to a dominance result of four dimensions, which is shown in the truth table of Figure 3.

Algorithm 2 VS   l ne dominance test

```
1: $Dom \leftarrow Incomparable$; $i \leftarrow 1$;
2: while $i \le d$ do
3:     $M_l$, $M_r \leftarrow SIMD\_Load(p[i, i+V-1], q[i, i+V-1])$;
4:     $M_{gt}$, $M_{ge} \leftarrow SIMD\_Compare(M_l, M_r)$;
5:     $GT$, $GE \leftarrow SIMD\_Mask(M_{gt}, M_{ge})$;
6:     if $GE$ is set in all significant bits then
7:         if $GT$ is set in any significant bit then //Case 1
8:             if $Dom = Right$ then
9:                 return $Incomparable$;
10:            end if
11:            $Dom \leftarrow Left$;
12:        end if
13:        //Undecided: Case 4
14:    else if $GT$ is set in any significant bit then //Case3
15:        return $Incomparable$;
16:    else //Case 2
17:        if $Dom = Left$ then
18:            return $Incomparable$;
19:        end if
20:        $Dom \leftarrow Right$;
21:    end if
22:    $i \leftarrow i + V$;
23: end while
24: return $Dom$;
```

Case 1 [**Left_dominating**] if $\forall i \in [1, V]$: $p_i \ge q_i \wedge \exists j \in [1, V]$: $p_j > q_j$, then $p \succ q$ holds.

Case 2 [**Right_dominating**] if $\forall i \in [1, V]$: $p_i \le q_i \wedge \exists j \in [1, V]$: $p_j < q_j$, then $q \succ p$ holds.

Case 3 [**Incomparable**] if $\exists i \in [1, V]$: $p_i > q_i \wedge \exists j \in [1, V]$: $p_j < q_j$, then $p \sim q$ holds.

Case 4 [**Undecided**] if $\forall i \in [1, V]$: $p_i = q_i$, then the dominance relationship is not decided yet.

VS   l ne executes far less instructions. As we need not to iterate each dimension to interpret the packed conditions, lots of extra instructions to handle loops and SIMD register values are actually eliminated. If N   eSIMD can determine dominance result early for all data, it probably performs well enough to compete with VS   l ne. However, skyline algorithms frequently handle large datasets and the average number of dimensions to examine is more than just one or two. In such cases, VS   l ne can outperform N   eSIMD for most of cases. Even compared to   S   l ne, we execute less number of instructions, as we handle four dimensions in one iteration. According to our experiments, executed instructions are reduced by 28∼45% for various inputs, which is translated to the CPU cycle reductions by 30∼60%.

In addition, handling packed conditions from SIMD comparisons produces more predictable control structures. Since we executes less number of branches and dominance results are decided at once for multiple dimensions, hardware branch predictors can work more favorably in our algorithm. Meanwhile, both S l ne and N eSIMD process each dimension one by one, which results in more diverse outcomes for the same branch instructions, thus making hardware branch predictors difficult to predict their branch directions. Performance gap between the two and VS l ne is largely due to far less mispredicted branches. According to our experiments, the number of mispredicted branches are reduced by 53∼97% for various inputs, which is translated to the CPU cycle reductions by 10∼20%.

Algorithm 2 shows the vectorized version of the dominance test used in VS l ne. Three main operations such as SIMD_Load, SIMD_Compare, and SIMD_Mask is illustrated in Figure 1 with slight modifications. Specifically, SIMD_Load read four dimensions from each tuple to compare, which is the same. Modifications are applied to SIMD_Compare and SIMD_Mask. SIMD_Compare generates two results in M_{gt} and M_{ge} with *greater-than* and *greater-than-or-equal* comparisons, respectively. Similarly, SIMD_Mask summarizes two comparison results of M_{gt} and M_{ge} into GT and GE, respectively. Conditional control flows which test two conditions, *GE-all-bits* and *GT-any-bit*, lead to four cases – *left-dominating*, *right-dominating*, *incomparable* and *undecided* as specified in the algorithm.

In summary, two performance boosters of VS l ne are as follows: 1) reduction in number of executed instructions due to SIMD vectorization of dimension value load and compare, and 2) reduction in number of mispredicted branches due to simplified control. In the next section, we show experimental results for those performance boosters.

4. EXPERIMENTAL EVALUATION

This section reports our evaluations results by comparing our proposed algorithm VS l ne with S l ne. All algorithms were implemented in C language, and all experiments were conducted on a Linux server with a 2.6.18 kernel. The server was also equipped with an Intel i7-860 quad core processor running at 2.80GHz and 6GB main memory. To analyze the performance, we used Intel VTune performance analyzer [10].

We used synthetic datasets with the same experimental settings in S l ne [8]. Specifically, we generated two different data sets according to uniform and anti-correlated distributions. To measure scalability, we varied the dimensionality from 4 to 24 with an increment of two and the cardinality from $50K$ to $400K$ with a double increment in numbers. Similarly, we varied the number of threads from 1, 4, then 8. As SIMD parallelism degree is four, we executed the maximum multiple of four dimensions with SIMD instructions and the remaining dimensions with non-SIMD instructions. Again, we clarify that our key contribution is not to reduce the number of tuple-to-tuple tests, but the cost of each test by using SIMD vectorization. As a result, we observed that VS l ne, performing the same number of tests at a lesser cost, outperforms S l ne in all cardinality from $50K$ to $400K$. To present multiple aspects of the performance, we used the results for two cardinalities of $100K$ and $200K$.

4.1 Effects of Vectorization

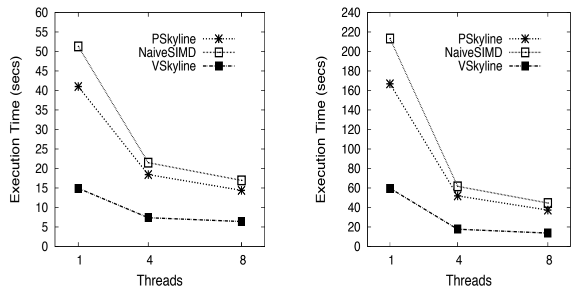
Figure 4 depicts the effects of vectorization in synthetic datasets. VS l ne outperforms S l ne by up to three folds. Observe that, in Figure 4, N eSIMD consistently underperforms S l ne over varying number of threads. The same observation holds for different cardinality and dimensionality. A possible explanation to such performance degradation is the naive approach in handling packed conditions, which cancels out the advantages of SIMD vectorized loads and compares.

Specifically, Table 1 compares three skyline algorithms, S l ne, N eSIMD, and VS l ne, with two datasets on different number of threads. Uniform and anti-correlated datasets, each of which has 12 dimensions and $200K$ data points, are used for our detailed performance analysis. The numbers for 4, 8 threads display the measured numbers on one thread, as numbers on other threads are very similar. Compared to S l ne, the numbers of executed instructions (NUM_INST) and branches (NUM_BR) are reduced by 30∼35% and 52∼63%, respectively. Due to the reduced number of iterations in dominance tests, VS l ne executes less instructions including branches than S l ne. Meanwhile, N eSIMD examines packed conditions one dimension by one dimension, which makes it execute more instructions even than S l ne. Since extra instructions to manipulate SIMD results such as shift and bit mask instructions are required in N eSIMD, it needs to execute more instructions.

One of good characteristics in VS l ne is the reduced number of mispredicted branches. As it processes packed conditions as a whole with two SIMD comparisons, control flow becomes more predictable than others. Thus, its mispredicted branch ratios (BR_MISP) become far smaller than the other algorithms, implying the performance gain of VS l ne. Specifically, compared to S l ne, the execution cycle (CPU_CLK) of VS l ne is reduced by 55∼65%. We also measured detailed performance counts with various dimensionality and cardinality and observed

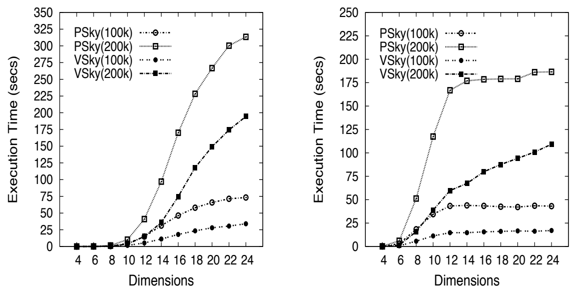
Table 1: Performance comparison of skyline algorithms (12d, 200K)

		Uniform				Anti-correlated			
threads	algorithms	CPU_CLK (cycle)	NUM_INST (million)	NUM_BR (million)	BR_MISP (%)	CPU_CLK (cycle)	NUM_INST (million)	NUM_BR (million)	BR_MISP (%)
1	PSkyline	137,104 M	121,850	44,803	6.66	559,313 M	497,099	181,660	6.51
	NaiveSIMD	171,539 M	224,117	47,410	6.52	711,399 M	928,903	195,775	6.40
	VSkyline	49,469 M	79,296	16,919	2.91	199,643 M	330,107	68,105	2.21
4	PSkyline	54,127 M	43,613	16,139	7.74	151,849 M	130,304	47,843	6.55
	NaiveSIMD	62,898 M	78,650	17,951	7.04	180,548 M	240,708	55,013	6.04
	VSkyline	21,614 M	29,520	6,977	5.30	50,862 M	90,170	20,371	2.38
8	PSkyline	40,955 M	26,170	7,890	9.71	108,988 M	74,628	22,247	7.90
	NaiveSIMD	49,073 M	46,246	9,029	8.81	128,822 M	132,013	25,625	7.14
	VSkyline	18,350 M	17,710	3,752	6.86	39,456 M	49,159	10,087	2.60



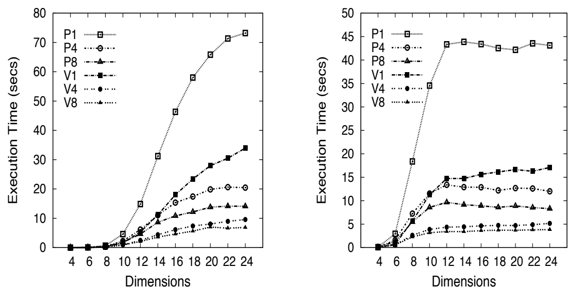
(a) Uniform (b) Anti-correlated

Figure 4: Performance of algorithms (12d, 200K)



(a) Uniform (b) Anti-correlated

Figure 5: Scalability on single-core (100K, 200K)



(a) Uniform (b) Anti-correlated

Figure 6: Scalability on multi-core (100K)

similar trends to the case in Table 1. The number of mispredicted branches, not only the number of instructions are reduced, improving the performance of VS l ne.

4.2 Effects of Dimensionality

Theoretical maximal speedup that can be obtained from vectorization is V-fold, where V indicates the degree of data-level parallelism. In our experiment, V is four as our SIMD instructions handle four data values. SIMD vectorization, however, requires extra instructions to manipulate the result stored in SIMD registers. Thus, actual speedup cannot reach the theoretical optimum, as VS l ne incurs the same overheads. In addition, it has additional computational overheads of deciding how to branch, after comparisons, which further hinders from achieving the optimal speedup. Figure 5 reports evaluation results of S l ne and VS l ne in uniform and anti-correlated datasets over varying cardinality (100K and 200K). We can observe that, VS l ne consistently outperforms S l ne, though the speedup varies over different parameter settings.

Figure 6 extends these evaluations, varying the number of threads. The solid markers indicate the results of VS l ne, denoted by Vx, when x indicates the number of threads. The hollow markers indicate those of S l ne, denoted by Px for using x threads. Note that, comparing when number of threads is four, both algorithms achieve up to 3-times speedup from those when number of threads is one. Though such speedup decreases, when comparing numbers of threads are four and eight, the overall performances monotonically improves as the number of threads increases and VS l ne consistently outperforms S l ne, which indicate that our proposed method for dominance tests in VS l ne are improving performances, without hindering the inherent effectiveness of S l ne.

5. CONCLUSIONS

In this paper, we proposed an efficient vectorization for skyline computation, exploiting data-level parallelism of dominance tests by using SIMD architectures. We first showed that a naive vectorization of dominance tests performs even worse than a non-vectorized version. The naive vectorization sequentially handled packed conditions for one dimension as in the non-vectorized version, incurring the increase of extra instructions to manipulate dominance tests. To address this problem, VS l ne deals with packed conditions as a whole, which is able to reduce the number of both branch instructions and branch mispredictions. VS l ne could thus boost the performance by up to three folds. In addition, VS l ne can be more efficient as the parallelism degree in future CPUs increases, and the idea of handling packed conditions can shed light on other similar applications.

ACKNOWLEDGMENTS

We are grateful to Dr. Sungwoo Park for providing S l ne source code. This work was supported in part by MKE, Korea under ITRC NIPA-2010-(C1090-1021-0008)and MEST, Korea under NRF Grant (NRF-2009-0084870). This research was supported by National IT Industry Promotion Agency (NIPA) under the program of Software Engineering Technologies Development.

6. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

[2] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[3] J. Chomicki, P. Godfery, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, pages 717–719, 2003.

[4] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB*, pages 229–240, 2005.

[5] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286, 2002.

[6] K. C. Lee, B. Zheng, H. Li, and W.-C. Lee. Approaching the skyline in Z order. In *VLDB*, pages 279–290, 2007.

[7] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progessive algorithm for skyline queries. In *SIGMOD*, pages 467–478, 2003.

[8] S. Park, T. Kim, J. Park, J. Kim, and H. Im. Parallel skyline computation on multicore architectures. In *ICDE*, pages 760–771, 2009.

[9] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface(4th edition)*. Morgan Kaufmann Publishers Inc., 2008.

[10] J. Reinders. *VTune Performance Analyzer Essentials*. Intel Press, 2005.

[11] K. Tan, P. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, pages 301–310, 2001.

[12] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *SIGMOD*, pages 227–238, 2008.

[13] S. Zhang, N. Mamoulis, and D. W. Cheung. Scalable skyline computation using object-based space partitioning. In *SIGMOD*, pages 483–494, 2009.

[14] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156, 2002.