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
{aplys, 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 VSkyline, 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 VSkyline 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 exponen-

tially. 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-vektorization thus performs even worse than a non-vectorization version.

To address this problem, we devise an efficient vectorization, called VSkyline, which performs the dominance test with SIMD instructions by determining incomparability in a block of four dimen-
sional values. Our performance evaluation using a performance monitor shows that VSkyline 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 VSkyline, and Section 4 then validates performance evaluation of VSkyline 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 $\text{dom}(D_i)$ of non-negative real number $\mathbb{R}^+$. Let $\mathcal{S}$ be a set of finite $n$ tuples as a subset of $\text{dom}(\mathcal{D})$. A tuple $p$ in $\mathcal{S}$ is represented as $p = (p_1, \ldots, p_d)$, where $\forall i \in [1,d] : p_i \in \text{dom}(D_i)$.

Based on these notations, we formally state 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., $\{p \in \mathcal{S} \mid \exists q \in \mathcal{S} : q \succ p\}$. 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 BNL [2], Index [11], SFS [3], and LESS [4] focused on optimizing tuple ordering to prune more dominated tuples early on; (2) partitioning-based algorithms such as NN [5], BBS[7], ZSearch [6], and OSPS [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 PSkyline, 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 PSkyline, 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 VSkyline, which optimizes the complex condition tests in the dominance test. We apply our vectorization technique to PSkyline. Since data-level parallelism in VSkyline is orthogonal to thread-level parallelism in PSkyline, we can further improve the performance for PSkyline by using our vectorization technique.

### Algorithm 1 PSkyline dominance test

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

### 3.1 PSkyline Overview

PSkyline [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 PSkyline 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 PSkyline 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 \( p_i \) and \( q_i \) values, iterating the loop until the dominance relationship is decided.

Specifically, in an iteration of the loop, \( \text{Dom} \) is set to either Left or Right. If the result of the current iteration is different from that of the previous iteration, \text{Incomparable} is returned. That is, even in the best case, the result is obtained at the second iteration. Returning \text{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

Figure 2: A dominance test in NaiveSIMD

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, $M_t$, $M_r$, 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.

Interpreting the significant bits for the dominance test can be implemented in a naive way (NaiveSIMD) or an optimized way (VSkyline). 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. NaiveSIMD 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, NaiveSIMD 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 ($I$). NaiveSIMD 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, NaiveSIMD executes 1.5x more instructions than PSkyline. Moreover, PSkyline suffers from a
sizable number of mispredicted branches, which accounts for 15~20% of execution cycles on various inputs. NaïveSIMD shares the same characteristics. Thus, NaïveSIMD performs worse than PSkyline.

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 VSkyline, 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 VSkyline performs the dominance test by taking the comparison results of four dimensions as a whole. Similar to NaïveSIMD, 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. NaïveSIMD examines one dimension by one dimension, even after parallelizing compares with SIMD instructions. On the contrary, our VSkyline 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 VSkyline is incomparable, which is the same result shown in Figure 2.

In general, the dominance test in VSkyline 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 VSkyline dominance test

1. \(\text{Dom} \leftarrow \text{Incomparable}; i \leftarrow 1;\)
2. \(\text{while } i \leq d \text{ do}\)
3. \(\text{}\quad M_i, M'_i \leftarrow \text{SIMD.Load}[p[i, i+V-1], q[i, i+V-1]];\)
4. \(\text{}\quad M_{ge}, M'_{ge} \leftarrow \text{SIMD.Compare}(M_i, M'_i);\)
5. \(\text{}\quad \text{GT}, GE \leftarrow \text{SIMD-And}(M_{ge}, M'_{ge});\)
6. \(\text{if GE is set in all significant bits then} // \text{Case 1}\)
7. \(\text{}\quad \text{if Dom = Right then}\)
8. \(\text{}\quad \text{return Incomparable;}\)
9. \(\text{}\quad \text{end if}\)
10. \(\text{}\quad \text{Dom} \leftarrow \text{Left;}\)
11. \(\text{\end if}\)
12. \(\text{\end while}\)
13. \(\text{\end case 1}\)
14. \(\text{\end case 2}\)
15. \(\text{\end case 3}\)
16. \(\text{\end case 4}\)
17. \(\text{\end case 5}\)
18. \(\text{return Dom;}\)

**Case 1** [Left-dominating] if \(\forall i \in [1, V] \mid p_i \geq q_i \wedge \exists j \in [1, V] \mid p_j > q_j\), then \(p \succ q\) holds.

**Case 2** [Right-dominating] if \(\forall i \in [1, V] \mid p_i \leq q_i \wedge \exists j \in [1, V] \mid p_j < q_j\), then \(q \succ p\) holds.

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

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

VSkyline 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 NaïveSIMD can determine dominance result early for all data, it probably performs well enough to compete with VSkyline. 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, VSkyline can outperform NaïveSIMD for most of cases. Even compared to PSkyline, 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 PSkyline and NaiveSIMD 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 VSkyline 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 VSkyline. 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 $M_g$ and $M_e$ with greater-than and greater-than-or-equal comparisons, respectively. Similarly, SIMD.Mask summarizes two comparison results of $M_g$ and $M_e$ 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 VSkyline 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 VSkyline with PSkyline. 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 PSkyline [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 VSkyline, performing the same number of tests at a lesser cost, outperforms PSkyline in all cardinality from 50K to 300K. 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. VSkyline outperforms PSkyline by up to three folds. Observe that, in Figure 4, NaiveSIMD consistently underperforms PSkyline 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, PSkyline, NaiveSIMD, and VSkyline, 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 PSkyline, the numbers of executed instructions (NUM_INST) and branches (NUM_BRANCH) are reduced by 30~35% and 52~63%, respectively. Due to the reduced number of iterations in dominance tests, VSkyline executes less instructions including branches than PSkyline. Meanwhile, NaiveSIMD examines packed conditions one dimension by one dimension, which makes it execute more instructions even than PSkyline. Since extra instructions to manipulate SIMD results such as shift and bit mask instructions are required in NaiveSIMD, it needs to execute more instructions.

One of good characteristics in VSkyline 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 VSkyline. Specifically, compared to PSkyline, the execution cycle (CPU_CLK) of VSkyline 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)

<table>
<thead>
<tr>
<th>threads</th>
<th>algorithms</th>
<th>CPU-CLK (cycle)</th>
<th>NUM-INST (million)</th>
<th>NUM-BR (million)</th>
<th>BR-MISP (%)</th>
<th>CPU-CLK (cycle)</th>
<th>NUM-INST (million)</th>
<th>NUM-BR (million)</th>
<th>BR-MISP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PSkyline</td>
<td>137,101 M</td>
<td>121,580</td>
<td>44,803</td>
<td>6.68</td>
<td>559,313 M</td>
<td>497,009</td>
<td>181,680</td>
<td>6.51</td>
</tr>
<tr>
<td></td>
<td>NaiveSIMD</td>
<td>171,539 M</td>
<td>224,117</td>
<td>47,410</td>
<td>6.52</td>
<td>711,399 M</td>
<td>928,603</td>
<td>195,775</td>
<td>6.40</td>
</tr>
<tr>
<td></td>
<td>VSkyline</td>
<td>49,409 M</td>
<td>79,366</td>
<td>16,919</td>
<td>2.91</td>
<td>199,643 M</td>
<td>330,107</td>
<td>68,105</td>
<td>2.21</td>
</tr>
<tr>
<td>4</td>
<td>PSkyline</td>
<td>54,127 M</td>
<td>43,613</td>
<td>16,139</td>
<td>7.74</td>
<td>151,249 M</td>
<td>130,304</td>
<td>47,843</td>
<td>6.55</td>
</tr>
<tr>
<td></td>
<td>NaiveSIMD</td>
<td>62,686 M</td>
<td>78,650</td>
<td>17,651</td>
<td>7.64</td>
<td>189,548 M</td>
<td>240,708</td>
<td>55,013</td>
<td>6.04</td>
</tr>
<tr>
<td></td>
<td>VSkyline</td>
<td>21,644 M</td>
<td>29,920</td>
<td>6,977</td>
<td>5.30</td>
<td>50,862 M</td>
<td>90,170</td>
<td>20,371</td>
<td>2.38</td>
</tr>
<tr>
<td>8</td>
<td>PSkyline</td>
<td>40,955 M</td>
<td>44,260</td>
<td>7,890</td>
<td>9.84</td>
<td>105,088 M</td>
<td>74,629</td>
<td>22,274</td>
<td>7.00</td>
</tr>
<tr>
<td></td>
<td>NaiveSIMD</td>
<td>49,073 M</td>
<td>46,246</td>
<td>9,029</td>
<td>8.81</td>
<td>128,022 M</td>
<td>132,013</td>
<td>25,625</td>
<td>7.14</td>
</tr>
<tr>
<td></td>
<td>VSkyline</td>
<td>18,360 M</td>
<td>17,710</td>
<td>3,752</td>
<td>6.66</td>
<td>39,456 M</td>
<td>49,159</td>
<td>10,087</td>
<td>2.60</td>
</tr>
</tbody>
</table>

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)

(a) Uniform  (b) Anti-correlated

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 VSkyline.

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 VSkyline 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 PSkyline and VSkyline in uniform and anti-correlated datasets over varying cardinality (100K and 200K). We can observe that, VSkyline consistently outperforms PSkyline, 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 VSkyline, denoted by Vx, when x indicates the number of threads. The hollow markers indicate those of PSkyline, 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 VSkyline consistently outperforms PSkyline, which indicate that our proposed method for dominance tests in VSkyline are improving performances, without hindering the inherent effectiveness of PSkyline.
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, VSkyline deals with packed conditions as a whole, which is able to reduce the number of both branch instructions and branch mispredictions. VSkyline could thus boost the performance by up to three folds. In addition, VSkyline 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 PSkyline source code. This work was supported in part by MKE, Korea under ITRC NIPA-2010- C1100-10214-0008 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


