Towards Automation Of Sorting Implementations For SIMD Architectures

Minhaj A. Khan
Bahauddin Zakariya University, Pakistan.

Abstract: Most of the sorting approaches are based on standard algorithms which are optimized in terms of number of steps (comparisons or interchanges etc.). The lack of parallelism in the sorting algorithms results in sub-optimal implementations requiring the code to be manually optimized. This paper presents an automated approach for sorting data on SIMD architectures. The elements are sorted in such a way that they fully exploit the parallelism of these architectures. In the first phase, the elements are arranged in sorted vectors with minimum number of comparisons through partially evaluated cases. These vectors are then merged while using element broadcast to produce a large virtual vector that is sorted, and may be incorporated in Quick sort for sorting large data. The experiments performed on Intel Core-2 duo and Power-4 processors using icc and gcc compilers show that our approach improves the performance of the sorting code on these architectures.

INTRODUCTION

SIMD (Single Instruction Multiple Data) instructions facilitate the execution of multiple operations in parallel. Modern processors support these instructions using different standards such as SSE (Intel Corporation, 2007) and Altivec (IBM Corporation, 2005) instruction sets. The code incorporating the SIMD instructions is assumed to speed up the performance of an application by the factor with which the operations are performed in parallel. However, the real performance achievement is diminished by two facts:

- The architectural level overheads may downgrade the performance of the application. For example, the misalignment of data (IBM. CELL, 2007), cache misses (Inoue et al., 2007) and the branches (Gedik et al., 2007) etc. all impact the performance of the application.
- The amount of parallelism inherent to the algorithm may not be large.

Unfortunately, the sorting implementations suffer from both of the above mentioned situations. In addition, the code generated by the compilers is not fully adapted to the architecture specific features. This limitation necessitates the code to be manually optimized. Although the manual efforts may improve the performance of the parallel code, it is still possible to produce high performance sorting code in an automated way.

The sorting algorithms are widely studied and implemented in well known libraries like Intel MKL (Intel Corporation), LAPACK (Anderson et al., 1999), VecLib (Apple Corporation, 2008) and STL (Nvidia Corporation, 2008). These sorting libraries are tuned for different architectures. Quick sort is employed by these implementations for sorting data of large size, and the Insertion sort is employed for sorting data of small size. The main reason for such implementations is that the Quick sort is regarded as the most effective sorting algorithm that achieves significant performance for data with different characteristics while benefiting from different architecture specific optimizations such as data locality. Similarly, for small data sizes, even the iterative Quick sort (instead of using recursive Quick sort) becomes costly and necessitates a simple Insertion sort to be implemented. However, the large complexity of insertion sort for average case, O(n^2) does not make it a good candidate for being applied to large input sizes.

In the worst case, the complexity of Quick sort is O(n^2) which is larger than that of Merge sort, which is O(nlogn). Despite the reduced complexity, the Merge sort is not considered to be effective since it is unable to properly exploit the data locality which degrades performance for large input data size.

In this paper, we suggest an automated approach aimed at improving the performance of sorting code. This approach works by exploiting the SIMD instructions to make the operations more parallel in an automated way. The code for sorting data is optimized by incorporating various primitives that also support portability for multiple architectures.

Sorting Approach:

We target the automated multi-level sort that benefits from SIMD instructions, and uses vector merge operation instead of insertion sort in order to keep the data locality to its maximum. For data existing within processor’s cache memory, we tune the code through SIMD instructions and vector merge operations. The code to sort vectors using SIMD instructions is termed as kernel at level 1, K_{l1}, whereas the code for performing vector merge operation is termed as the level 2 kernel, K_{l2}.

The final code then benefits from the following two characteristics: improved parallelization and data locality. For parallelism for the kernel K_{l1}, the SIMD instructions which provide vectorized shuffling,
comparison and broadcast operations may be used to increase instruction level parallelism (ILP). For data locality, the cache size is be used to limit the kernel KL2 generation, since the buffers within cache size benefit from data locality. To accomplish this, a variant of merge sort can be adapted to use SIMD instructions effectively. The final code with both the kernels, KL1 and KL2, is then embedded into the simple Quick sort code to work for large data sizes.

<table>
<thead>
<tr>
<th>Input: Vectors V1 and V2, each having 4 elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Sorted vectors V8 and V9</td>
</tr>
</tbody>
</table>

**Step a:**

// Sorting 2 elements through shuffled replication

Shuffle (V1[a2,a0], V3[a1,a0])
Shuffle (V2[a2,a0], V3[a3,a2])
Shuffle (V1[a3,a1], V4[a1,a0])
Shuffle (V2[a3,a1], V4[a3,a2])

Cmp MASK == V4 < V3
If (MASK == xxx0)
    Shuffle (V1[a1,a0], V4[a1,a0]);
If (MASK == xx0x)
    Shuffle (V1[a3,a2], V4[a3,a2]);
If (MASK == x0xx)
    Shuffle (V2[a1,a0], V5[a1,a0]);
If (MASK == 0xxx)
    Shuffle (V2[a3,a2], V5[a3,a2]);

**Step b:**

// Arranging 4 elements to generate sorted vectors

// Sorting vector V4
Shuffle (V4[a1,a0,a1,a0], V6[a3,a2,a1,a0])
Shuffle (V4[a2,a3,a3,a2], V7[a3,a2,a1,a0])
Cmp MASK = V6 < V7
Call Sort4 (MASK, V4, V8)

// Sorting next vector V5
Shuffle (V5[a1,a0,a1,a0], V6[a3,a2,a1,a0])
Shuffle (V5[a2,a3,a3,a2], V7[a3,a2,a1,a0])
Cmp MASK = V6 < V7
Call Sort4 (MASK, V5, V9)

---

**Algorithm 1:** Sorting vectors using kernel KL1.

The SIMD-based sorting algorithm makes use of vectorization primitives in order to sort data using vectors.

**Automating Optimization Through Implementation of Primitives:**

We define macros for implementation of primitives in order to process the vector elements that make our approach to be easily portable and automated. The macros have the following formats:

**Shuffle (Source, Destination):**

Copies the given elements from source vector to destination vector at the corresponding indices.

**Cmp MASK = source1 < source2:**

Compares the source vectors for the corresponding elements in the vectors, and returns the mask as the destination vector. The 128-bit value in the mask can be used to interpret the result of the operation. For each (32-bit) element of the source vectors, 32-bit 1’s are returned if the operation for the corresponding elements is true, otherwise 0’s are returned. While comparing the mask values, we represent simple digit value ‘1’ to correspond to 32-bit 1’s, ‘0’ to correspond to 32-bit 0’s and ‘x’ to correspond to either 32-bit 1’s or 32-bit 0’s.
**CopyElements (source index, destination index, number of elements):**
Copies a specified number of elements starting from the index of source vector to elements starting at the index of destination vector.

**Generating Large Virtual Vectors:**
Initially, input data elements are sorted using SIMD instructions (given in Algorithm 1 and Algorithm 2) to generate two vectors that are merged subsequently (using Algorithm 3) to generate a large virtual vector.

```
Procedure Sort4 (MASK, V_x, V_y)
switch(MASK)
  case 0x11:
    Shuffle (V_x[a3,a1,a2,a0], V_y[a3,a2,a1,a0]); break;
  case xx01:
    Shuffle (V_x[a1,a3,a0,a2], V_y[a3,a2,a1,a0]); break;
  case xx10:
    Shuffle (V_x[a3,a1,a0,a2], V_y[a3,a2,a1,a0]); break;
  case x0xx:
    Shuffle (V_x[a1,a0,a3,a2], V_y[a3,a2,a1,a0]); break;
  case xx11:
    Shuffle (V_x[a1,a3,a2,a0], V_y[a3,a2,a1,a0]); break;
  case 1xxx:
    Shuffle (V_x[a3,a2,a1,a0], V_y[a3,a2,a1,a0]); break;
```

Algorithm 2: Pseudo-code of procedure Sort4, used for sorting 4-elements of V_x.

**Sorting Individual Vectors:**
As given in Algorithm 1, the data elements are first shuffled within vectors so that the minimum number of comparisons is required using vector compare (Cmp) operations. After comparison and shuffling using step-a, the vectors contain the sorted pairs by simple shuffling (placing the smallest element first). The vectors V_4 and V_5 are generated after step-a which contain the pairs sorted.

<table>
<thead>
<tr>
<th>a3</th>
<th>a1</th>
<th>a2</th>
<th>a0</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>a3</td>
<td>a0</td>
<td>a2</td>
</tr>
<tr>
<td>a3</td>
<td>a1</td>
<td>a0</td>
<td>a2</td>
</tr>
<tr>
<td>a1</td>
<td>a3</td>
<td>a2</td>
<td>a0</td>
</tr>
<tr>
<td>a1</td>
<td>a3</td>
<td>a2</td>
<td>a0</td>
</tr>
<tr>
<td>a3</td>
<td>a2</td>
<td>a1</td>
<td>a0</td>
</tr>
<tr>
<td>a3</td>
<td>a2</td>
<td>a1</td>
<td>a0</td>
</tr>
<tr>
<td>a3</td>
<td>a2</td>
<td>a1</td>
<td>a0</td>
</tr>
</tbody>
</table>

Once the pairs of elements in a vector are sorted, the sorting of the entire vector (4-elements) takes place, using step-b. The vector is then shuffled according to the pattern that enables to obtain the sorted vector. Consider the vector V_4 = [a3,a2,a1,a0] with 4 elements, the new vectors V_6 and V_7 after shuffling would contain,

V_6 = [a1,a0,a1,a0],
V_7 = [a2,a3,a3,a2].

This shuffling keeps the number of compare operations to its minimum. In general, the 4 unsorted elements of a vector require 4! possible sorted sequences of that vector. However, using the result of the step-a, we have, a_0 < a_1 and a_2 < a_3. Therefore, we are left with six possible sequences to be checked at step-b, as given in Table 1.

The Algorithm 1 then invokes the procedure Sort4, that takes as parameters the mask, source vector and destination vector, to generate the sorted vector, V_y.

An example of sorting individual vectors, V_1 and V_2, is also shown in Figure 1. At step-a, the elements are shuffled and then compared to produce two sorted vectors that can be merged using the step-b.

**Merging Vectors:**
The vectors sorted (using Algorithm 1 and Algorithm 2) in the previous phase are merged to form a large virtual vector that resides within the set of vector registers. As shown in Algorithm 3, a single element of the vector V_9 is broadcast in a vector V_7 to be compared with the vector V_8. After comparison, either the smaller elements of the first vector are placed into the resultant vector (case 2,3 & 4) , or the broadcast element is moved to the resultant vector (case 1). The best case of merging vector occurs when the last element of the first vector
(V₈) is less than the broadcast element (V₉[a₀]). As the worst case, While loop may execute 4 times. The remaining elements in the vectors are then moved to the resultant vector which is a large virtual vector, using the CopyElements primitive.

**Fig. 1:** Sorting pair of vectors using shuffling and compare instructions as given in the algorithm (steps a and b). The resultant vectors V₈ and V₉ can now be merged.

```plaintext
// Merging the sorted vectors (each having 4 elements) through element broadcast to generate a large virtual vector
Let S = V₈ U V₉, R = V₁ U V₂ be virtual vectors.
k=0, j=0, i=0

WHILE { i <= 3 & j <= 3}
// Broadcast V₉[a₀] to V₇
Shuffle (V₉[a₀,a₀,a₀,a₀], V₇[a₀,a₁,a₂,a₃])
Cmp MASK = V₇ < V₈
switch(MASK){
case xxx1:
    Shuffle (V₉[a₀], R[aₖ]); j++; V₉ = V₉ >> 32; break;
case xx10:
    Shuffle (V₈[a₀], R[aₖ]); i++; V₈ = V₈ >> 32; break;
case x100:
    Shuffle (V₈[a₁,a₀], R[aₖ+1,aₖ]); i+=2; V₈ = V₈ >> 64; break;
case 1000:
    Shuffle (V₈[a₂,a₁,a₀], R[aₖ+2,aₖ+1,aₖ]); i+=3; V₈ = V₈ >> 96; break;
case 0000:
    Shuffle (V₈[a₃,a₂,a₁,a₀], R[aₖ+3,aₖ+2,aₖ+1,aₖ]); i+=4; V₈ = V₈ >> 128; break;
}
CopyElements (&S[k], &R[k], n*2-k-1)

Algorithm 3. Merging the vectors, kernel K₄₂.

**Experimental Setup and Results:**
For a comparison of the code generated by our automated optimization using SIMD instructions and the scalar implementation, we used Power-4 processor with L₁ cache of 32KB, L₂ cache of 1.5MB, having 1.6GHz processor frequency, and the Intel Core-2 duo processor with data L₁ cache of 16KB, L₂ cache of 256KB and L₃ cache of 3MB. The compiler gcc v 3.2 has been used for experimentation with Power-4 processor, whereas the compiler icc v 9.0 has been used for experimentation with Intel Core-2 duo processor.
For experimentation, the SSE-3.0 (Intel Corporation, 2007) SIMD vector instructions have been used for Intel Core-2 duo processor, and Altivec (IBM Corporation, 2005) instruction set has been used for Power-4 processor. Both these implementations support 128-bit vectors that have been used for generating sorting kernels with 32-bit floating-point data elements.

The performance results in terms of speedup achieved by vectorized implementation for sorting 32-bit floating-point random data on both architectures are shown in Figure 3. Our optimized implementation performs significantly better than the scalar implementation, for small sizes of data. For large sizes, however, the speedup decreases due to the architectural constraints discussed in Section 1. The Intel Core-2 duo implementation achieves a maximum speedup of 1.98, whereas, for Power-4 processor, it achieves a maximum speedup of 1.88. On average, the Intel Core-2 duo implementation achieves a speedup of 1.70, and for Power-4 processor, it achieves a speedup of 1.60.

Related Work:
In the work related to tuning sorting libraries (implemented in SPIRAL), Li et al., 2004. Use the architecture specific features to improve the performance of sorting algorithms. Different sorting algorithms (Quick sort, CC-Radix sort and Multi-way merge) are then tuned on the basis of entropy and data sizes. In contrast, our approach uses single structure of sorting code that is adapted to benefit from better optimizations and better data locality. In addition, our approach is data independent and does not make assumptions about the nature of data.

Another approach of searching optimal sorting algorithm using genetic algorithms is provided in (Industrial and Applied Mathematics). A large search space of different parameters such as number of pivots, radix size & partition size etc. is deployed for which a better sorting code is identified through the genetic algorithm. On average, the tuned sorting code performs 62% faster than IBM ESSL on Power-4 processor with different standard deviations. This differs from our approach in which we adapt the sorting kernel to improve the merging of elements through SIMD instructions.

An improved implementation of qsort function for C libraries has been given in (Bentley and mcIlroy, 1993). Their code employs an effective partitioning algorithm (a variant of partitioning by Sedgewick (Li et al., 2004) together with a cost model and optimized swapping implementation. The tuning part specializes swapping code for different data types.

AA-sort (IBM. CELL, 2007), CELLSort (SGI Corporation, 1994) and GPUTerasort (Gedik, 2007) are sorting algorithms which employ the vectorized sorting. AA-sort uses a variant of Combsort for in-core processing of the elements, and an odd-even merge algorithm for out-of-core merging phase. Similarly, CELLSort uses bitonic sort while using SIMD instructions on Synergistic Processing Elements(SPEs) of the CELL processor (Govindaraju et al., 2006). It requires manual implementation of memory accesses (using DMA) and communication between the SPEs and PPE. The GPUTerasort, on the other hand, incorporates a hybrid sorting approach adapted for GPUs (Nvidia Corporation, 2008). The parallelism exploited through multiple computing elements improves the performance, however, none of these approaches provides an automated approach to tuning a sorting kernel.

ATLAS code generator (Sedgewick, 1978) tunes the code for mathematical kernels (like BLAS operations) through transformations like blocking or loop unrolling. In contrast, we leave the transformation decisions to the compiler, and tune the code through SIMD instructions.

Conclusion And Future Work:
In this paper, we suggest an approach that makes use of SIMD instructions for generating sorting code. A better parallelism and data locality is obtained by tuning the code for the particular architecture using two-level
kernels. The kernel $K_{L1}$ employs the SIMD sorting for some data elements. The second kernel $K_{L2}$ merges the kernel while working for data existing within processor cache memory.

The approach makes use of shuffling of data elements to first sort pairs of elements in the input vectors. Subsequently, the pairs are sorted to generate 4-elements sorted vectors. The sorted elements are placed in different vectors thereby generating a large virtual vector.

Our optimization approach is automated using the primitives and may work for various architectures. The experimentation results show that the optimized implementations perform significantly better than the scalar implementations, on the Intel Core-2 duo and the Power-4 processors.

As future work, we intend to enhance the primitives to support multiprocessor architectures including those with heterogeneous systems such as CELL processors and GPUs.

REFERENCES


