The Computer Journal Advance Access published August 19, 2015

doi:10.1093/comjnl/bxv063

Fast Quicksort Implementation Using AVX Instructions

Shay Gueron[1,2]* and Vlad Krasnov[3]

[1]*Department of Mathematics, University of Haifa, Haifa, Israel*
[2]*Intel Corporation, Israel Development Center, Haifa, Israel*
[3]*CloudFlare Inc., London, UK*
**Corresponding author: shay@math.haifa.ac.il*

This article describes a technique for implementing the quicksort sorting algorithm. Our method 'vectorizes' the computations and leverages the capabilities of the advanced vector extensions (AVX) instructions, available on Intel Core processors, and of the AVX2 instructions that were introduced with Intel's recent architecture codename Haswell. Our solution offers several advantages when compared with other high-performance sorting implementations, such as the radix sort, as implemented in Intel IPP library, or the introsort, as implemented in the C++ STL. In addition to sorting numeric arrays, our method can also be used to sort complex structures with numeric keys and even pointers to such structures.

Keywords: sort; quicksort; Sandy Bridge; Haswell; SIMD; AVX; AVX2

Received 2 November 2014; revised 28 May 2015
Handling editor: Rajeev Raman

1. INTRODUCTION

Sorting is one of the classical problems in computer science, and it is used in a variety of applications and on a variety of platforms. A few examples are SQL servers, databases, searches and image processing [1, 2]. A lot of research has been invested in optimizing various sorting algorithms. In this article, we focus on 'quicksort' [3], which is one of the fastest sorting algorithms that are used in practice, and as such, it is a target for optimizations [4–8].

There are numerous sorting algorithms and techniques, and no single implementation is optimal for all situations. Some implementations optimize for the data types, the amount of sorted data and its (expected) distribution or for the available memory.

Some sorting algorithms can be optimized straightforwardly by means of parallelization, and in such cases, it is obvious that single instruction multiple data (SIMD) architecture [9] is a useful tool. One example is the 'merge sort' algorithm [10, 11]. Radix sort is another example [12]. Other special cases, such as very short arrays, can also be handled efficiently, with SIMD instructions [13]. However, this is not the case for quicksort because the steps of this algorithm split a given list of elements into two 'sublists' that may have a different number of elements. This operation cannot be parallelized straightforwardly (two adjacent elements in the list can be assigned to different groups, and this introduces a branch in the implementation). For this reason, there are no published techniques (to the best of our knowledge) that leverage SIMD instructions to improve the performance of quicksort, compared with using the arithmetic logic unit (ALU) instructions.

In this article, we develop a method for implementing quicksort, using the AVX instructions (a set of SIMD instructions) that were introduced in the Intel's Sandy Bridge architecture. The key twist in our method is using lookup tables to generate a special shuffle mask that accelerates the sort operation. The method is further improved by the AVX2 instructions introduced in Intel's Haswell architecture [14].

2. PRELIMINARIES: QUICKSORT

Quicksort is a very fast comparison-based sorting algorithm. Although its worst-case time complexity is $O(n^2)$, it has average time complexity of $O(n \log n)$ [3], comparable with merge sort that has both a worst-case and average time complexity of $O(n \log n)$. While the complexities of merge sort and quicksort are comparable, quicksort tends to be faster in practice. A sorting algorithm with an even better complexity is radix sort [12], having linear complexity $O(m \times n)$, where m is the number of 'digits' in the key. As shown in [12, 15], it is the

fastest performing sort for short keys, as long as the size of the sorted data does not exceed half the available memory. Radix sort becomes ineffective as soon as this threshold is crossed due to a large amount of required page swapping. By comparison, quicksort and merge sort also require additional space allocation, on top of the space for the sorted data. However, in quicksort, only the first level of recursion could actually use the entire space (and on average only half of it), and subsequent recursion levels use less space. Quicksort also requires a deep recursion stack. Both merge sort and quicksort benefit from sequential access to data, which is much faster than random access on modern SSDs (Solid States Drives). Another disadvantageous property of radix sort is that its performance decreases linearly with the size of the key.

Quicksort is a simple 'divide and conquer' approach for sorting a given unsorted list (array) of elements [2]. We describe this algorithm briefly. In the first step, a single element of the unsorted list is chosen as a 'pivot' element, according to some heuristics. Then, all the elements are divided into two (sub) lists: elements smaller than or equal to the pivot are assigned to one list, and the remaining elements are assigned to the second list. The algorithm is then applied recursively to both lists. The recursion stops when one of the lists includes only one element. Finally, the concatenation of the two lists is the required sorted array. Figure 1 illustrates this flow on a small array.

Quicksort can be implemented in two variants. Variant 1 uses extra space: when an element is compared with the pivot, it is copied to one of two lists. One of these lists can override the original list. Variant 2 uses two pointers: one pointer iterates from the bottom of the list to its top, and the other pointer iterates from the top of the list to its bottom. When the first pointer finds an element that is larger than the pivot element, and the second pointer finds an element that is smaller than the pivot element, these two elements are swapped. The algorithm stops when the two pointers meet, and the pivot element is placed at that point. This method is implemented in the C library [16]. Such an implementation is not 'stable': the order between two 'equal' elements is not preserved. It is important to understand that stable sorting is a significant property of a sorting algorithm, especially for sorting lists by multiple keys, e.g. first by one key and then by another key. In such cases, keeping the order that results from the initial sort is desirable.

We also note here that software implementations of these variants, on a modern out-of-order processor, take a performance hit due to branch mispredictions [6].

Our approach here can be applied to Variant 1 of quicksort, which requires extra space. When the selected pivot element is the last element in the list, it performs a stable sort.

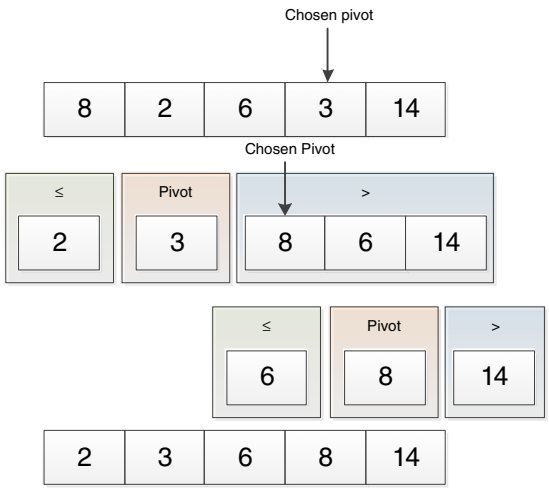

FIGURE 1. An illustration of the quicksort recursion.

3. THE UNDERLYING IDEA

SIMD instructions perform, simultaneously, the same operation on several 'elements' that are stored in the instructions' operands (registers or memory locations) [9]. For some class of computations, SIMD-based implementations can provide higher throughput than the throughput achieved by using ALU operations. However, note that SIMD instructions do not support different branches for different elements in the same register.

Recall that, basically, quicksort requires one comparison and one move in each step, depending on the result of the comparison—thus a branch is required. The comparison step of quicksort is naturally 'SIMD friendly' because each element can be compared with the pivot element, independent of the other elements. However, splitting the elements into two lists is not SIMD friendly because it seems to require a different branch for each element. We overcome this problem by generating a special mask from the comparison result and by using this mask as an index to a (precomputed) lookup table that stores shuffle masks. These shuffle masks help us separate the elements into two lists.

4. IMPLEMENTING QUICKSORT WITH AVX

In this section, we demonstrate our method for sorting 32-bit integers. However, an analogous method can be used for other data types, such as 64-bit integers, and single/double precision real numbers (using the appropriate SIMD instructions, of course).

With AVX, we use 128-bit registers. Each register can hold four elements, each one is a 32-bit integer (or a single precision floating point number). We first describe the algorithm in Fig. 2.

In the following paragraphs, we explain how we can implement Algorithm 1 efficiently. Our notations use the AT&T assembler language syntax (i.e. the destination is the rightmost operand).

```
Algorithm 1: The parallelized quicksort loop
--------------------------------------------
Input:
    A, an array with N elements.
    N, the number of elements in A.
Output:
    A, an array with N elements, where the first K-1 elements are smaller-than-equal to the pivot,
    the Kth element is the pivot, and the remaining (N-K) elements are larger than the pivot.
    K, the index of the chosen pivot in the output array.
Setup:
    Choose an element (denoted P) of the array as the pivot.
    Larr  a pointer to an auxiliary array that stores elements X from the list, such that X <= P.
    Garr  a pointer to an auxiliary array that stores elements X from the list, such that X > P
    Lptr = Larr
    Gptr = Garr
While N >= 4:
1. Load 4 elements of A into a 128-bit SIMD register, D.
2. Compare the elements to the pivot, P.
3. Arrange, sequentially, all the elements X such that X <= P, in a SIMD register, L.
4. Arrange, sequentially, all the elements X such that X > P, in a SIMD register, G.
5. lc = the number of elements in L.
6. lg = the number of elements in G.
7. Store L at Lptr
8. Store G at Gptr
9. Lptr = Lptr + 4*lc
10. Gptr = Gptr + 4*lg
11. N = N-4
End while
If N>0 finish serially
Last step:
A = Concatenate (Larr, Garr)
K = (Lptr-Larr)/4 - 1
Output: A, K
```

FIGURE 2. Pseudo code for the parallelized quicksort loop.

Copies of the pivot element are placed in a SIMD register using the VPBROADCASTD instruction (this instruction duplicates a 32-bit value into a SIMD register):

VBROADCASTSS (pivot), P

Step 1 of the algorithm is performed by using the VMOVDQU instruction (this instruction performs an unaligned load into a SIMD register):

VMOVDQU (in), D

Step 2 is performed by using the VPCMPGTD instruction (this instruction compares two vectors of integers. The result is a vector that holds the element 0xffffffff if the comparison result was 'true' for that element, and 0 otherwise):

VPCMPGTD D, P, C

For floating point numbers, we use the instruction VCMPPS.

Step 3 is where the new algorithmic twist starts. First, we use the VMOVMSKPS instruction to generate a mask that consists of the most significant bit of each 32-bit element (the sign bit, if it is viewed as a floating point number) in a SIMD register and places it in a GPR (General Purpose Register):

VMOVMSKPS C, r

When this instruction completes, Register r holds a 4-bit mask that identifies the elements (of the comparison), which were greater than the pivot element. The 4-bit bitwise complement of the value in r identifies the elements that are less or equal to the pivot element.

We can now use the mask to shuffle the register, in a way that all the required elements are arranged sequentially in the register. To this end, we use two precomputed tables that hold an appropriate shuffle mask for each possible value r. One table holds 16 masks for r, and the other holds 16 masks for its bitwise complement. The values in the tables are provided in the Appendix.

Steps 3 and 4 are, therefore, performed this way:

SHL $4, r
VPERMILPS permTableLesser(, r), D, L
VPERMILPS permTableGreater(, r), D, G

We point out that the instruction VPERMILPS was introduced with the AVX instructions set and is therefore unavailable on SIMD architectures prior to the Sandy Bridge architecture. It shuffles the 32-bit elements in the second source operand,

according to the indices in the first source operand, and stores the result in the destination register.

Step 6 is performed by using POPCNT (an instruction that counts the number of set bits in a GPR). Step 5 is actually the result of subtracting the result of Step 6 from the number 4:

```
POPCNT r, r0
MOV  $4, r1
SUB r0, r1
```

Now, we store G and L at the right array:

```
VMOVDQU L, (Lptr)
VMOVDQU G, (Gptr)
```

Finally, we increment the pointers:

```
LEA (Lptr, r0, 4), Lptr
LEA (Gptr, r1, 4), Gptr
```

When the array contains less than four elements, it is handled using a regular GPR-based code.

This implementation has two advantages: (i) it is capable of handling four elements in parallel and (ii) it is branch free. Consequently, this leads to an efficient software implementation.

Our implementation takes further advantage of the high throughput of the SIMD instructions (in the Core i7-4770 processor), by handling 40 elements of the list every iteration, which are stored in a total of 5 SIMD registers. A code snippet that illustrates our implementation is given in the Appendix. Figure 3 illustrates the process.

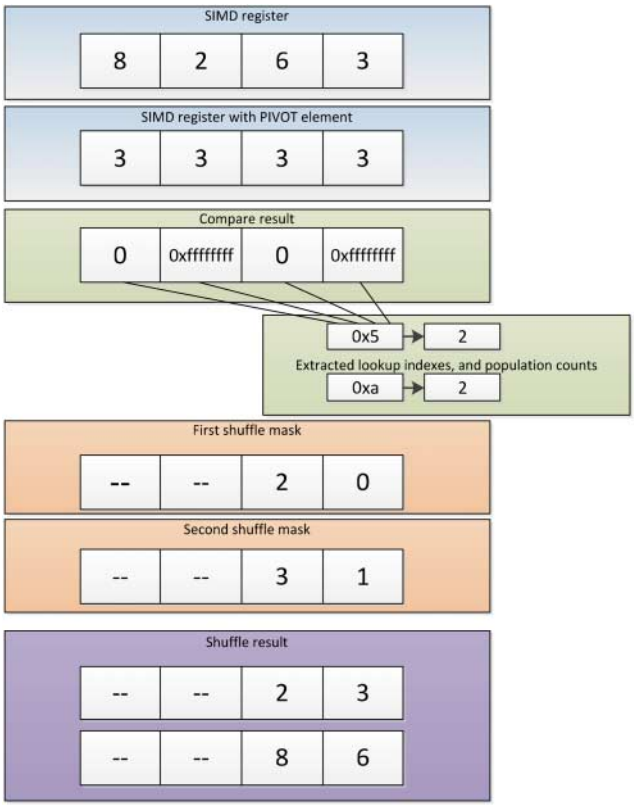

FIGURE 3. Illustrating vectorized quicksort.

5. QUICKSORT WITH AVX2

The AVX2 instructions (on integers) set was introduced with the Intel Architecture Codename Haswell. This set provides additional shuffle instructions and promotes the AVX integer instructions to operate on 256-bit registers. Another addition of AVX2, which we use here, are the new 'Gather' instructions. These instructions are able to load, simultaneously, several elements from separate memory locations. We demonstrate how these additional instructions can improve our implementation.

5.1. Wider registers with additional shuffle operations

With wider registers, it is possible to double the number of elements that are handled in a single iteration: we can load eight 32-bit elements or four 64-bit elements. We note that operating on eight elements is advantageous in terms of operations per element, but the down side is the significantly larger required tables: the mask can now have 8 bits, so we need two tables with 256 entries. In total, this implementation (when run on a Haswell processor) improves the performance by roughly 5%. On the other hand, 64-bit elements can be handled similarly to the way 32-bit elements are handled using AVX, and this leads to significant speedup, as we show in Section 6.

5.2. Gather instructions

With the new Gather instructions, it is possible to sort pointers to complex data structures. To this end, the implementation should be slightly modified. For example, suppose that sort pointers to the following C++ class:

```
class person
{
public:
   long id;
   long age;
   person();
};
```

First, the pointers need to be loaded (this is done as shown above). Then, the Gather instructions are used for loading the keys that are used for the sorting, as follows:

```
VMOVDQU      (in), D
VPCMPEQQ     T, T, T
VPGATHERQQ     T, (,D), K
```

In this case, we loaded the *id* field as the comparison key. Similarly, we can use *age* as comparison by: VPGATHERQQ T, 8(,D), K.

The K register is then compared with the pivot element in order to generate the shuffle mask, and the shuffle mask is applied to the pointers that reside in D.

By comparison, radix sort can only perform indirect sort of compound objects. It would require to copy all the keys to an auxiliary array, apply a special sort that produces an array of indexes and then sort the pointers according to the indexes.

6. RESULTS

6.1. Technical details

We implemented our technique for the most common data types: 32-/64-/128-bit integers, 32-/64-bit floating point numbers.

We use two libraries for reference. The first one is the IPP library [15], which is a library of performance primitives that is specifically optimized for Intel processors. We used the radix sort implemented of IPP, which is the fastest in-memory sort we are aware of. It requires an additional space of the size of the sorted data. We used IPP version 8.1.0 for the performance comparison. The IPP library provides functions to sort 32-/64-bit floating point numbers, and 32-bit signed integers, but lacks functions to sort 64− and 128-bit integers. For these data types, we interpolate the numbers based on the properties of radix sort and the available data.

The second library is STL [17], a standard C++ library. In particular, we compare the performance of our method with that of the STL sort implementation, which combines introsort and insertion sort (introsort itself is a combination of quicksort and heapsort). The STL implementation uses introsort until a recursion threshold is hit and then switches to the insertion sort algorithm. STL's quicksort implementation uses the median of three heuristic to select the pivot element. We point out that it is not a stable sort.

The STL sort is implemented using a C++ template and therefore available for any data type. When using the g++ compiler (version 4.8.1) with the gcc special type __uint128_t, it allows for a very efficient sort of 128-bit integers.

When sorting pointers without actually copying the data, the memory locality principle is not preserved. This leads to a large number of page faults and, therefore, to significantly slower performance for all the sort implementations. Consequently, the performance varies when sorting pointers (our method is consistently faster than the alternatives). Note that our algorithm provides a stable sort: this property can be important when sorting pointers (although it is irrelevant when sorting simple data types).

We also mention that when we have <32 elements in the sorted array, our implementation switches to insertion sort, which is more efficient for small arrays.

For the comparison, we generated random arrays of each data type, having various lengths (using the std::rand() C function and std::time(0) as the seed). The experiments were carried out on a 3.4 GHz Core i7-4770 (Haswell architecture) processor. Intel® Turbo Boost Technology was disabled in these experiments. Each experiment was repeated 10 000 times (randomizing the input arrays each iteration), and the performance was determined as the average time for sorting the arrays. The time was measured in CPU cycles by reading the time-stamp counters on the CPU before and after the sort.

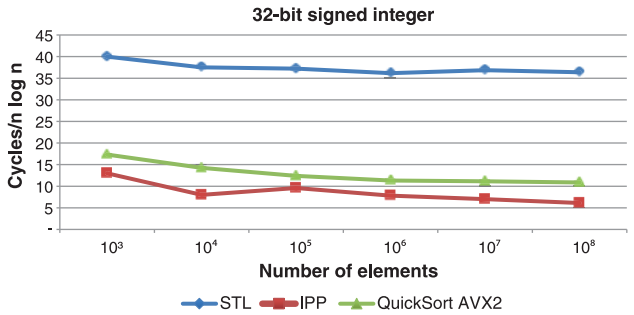

FIGURE 4. The performance of STL introsort, IPP radix sort and our AVX2 quicksort algorithms on Intel® Core i7-4770, 32-*bit signed integers*.

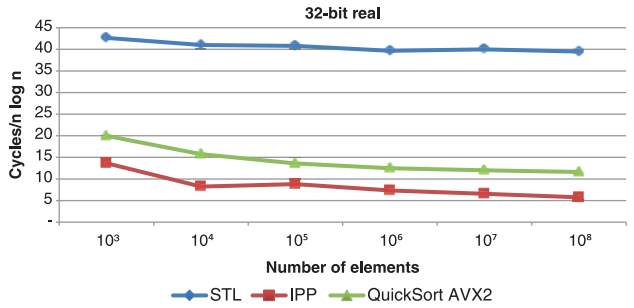

FIGURE 5. The performance of STL introsort, IPP radix sort and our AVX2 quicksort algorithms on Intel® Core i7-4770, 32-*bit real numbers*.

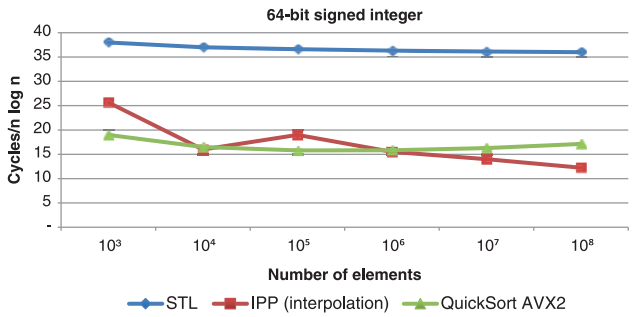

FIGURE 6. The performance of STL introsort, radix sort and our AVX2 quicksort algorithms on Intel® Core i7-4770, 64-*bit signed integers*. The IPP performance of radix sort is (optimistically) approximated as twice its 32-bit performance due to the lack of dedicated function to sort 64-bit signed integers.

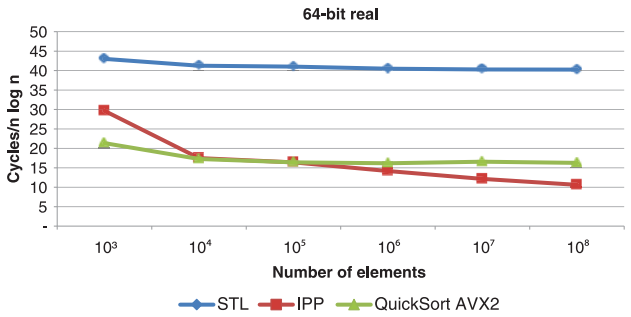

FIGURE 7. The performance of STL introsort, IPP radix sort and our AVX2 quicksort algorithms on Intel® Core i7-4770, 64-*bit real numbers*.

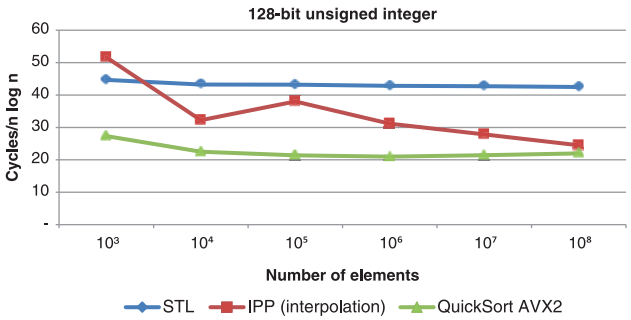

FIGURE 8. The performance of STL introsort, radix sort and our AVX2 quicksort algorithms on Intel® Core i7-4770, 128-*bit unsigned integers*. The IPP performance of radix sort is (optimistically) approximated as four times its 32-bit performance, due to the lack of dedicated function to sort 128-bit unsigned integers.

We verified that the intermediate averages of 5000 and 7500 repetitions also give similar results.

6.2. In memory sort

In this section, we report performance numbers for the case when all the sorted data can fit in memory (including the extra space required by IPP radix sort). Our test-bench machine had 16 GB of memory, but some of it is used by the OS, and a significant portion by the call stack (due to deep recursion).

Figures 4–8 show the results. The AVX2 quicksort is faster for small arrays for 64-bit keys. For 128-bit keys, it is faster on any amount of data.

6.3. Out of memory sort

In this section, we demonstrate the performance of the various algorithms when the data exceeds half of the available physical memory (Fig. 9). In this case, the data with the auxiliary space cannot fit entirely in the memory, and some of it swapped into the SSD drive. For that experiment, we used 2 147 483 647 elements for the 32-bit keys, which amounts to almost 8 GB of data (IPP passes the parameter as a signed integer, so we could not use exactly 8 GB). For 64-bit keys, we used 1 073 741 824 elements (exactly 8 GB of data). In the experiment, we allocated memory only for the main array and the auxiliary array, and the rest of the available memory was used by the OS and the call stack. The results are summarized in Table 1.

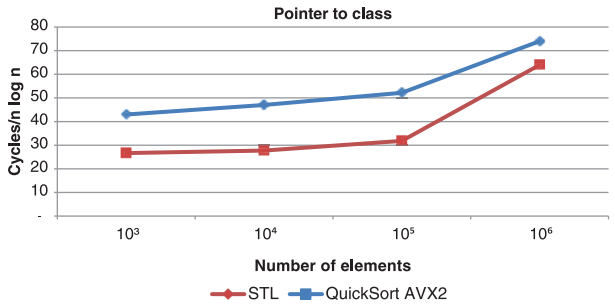

FIGURE 9. The performance of STL introsort, and our AVX2 quicksort algorithms on Intel® Core i7-4770, *pointers to a simple class*.

TABLE 1. The performance (in CPU cycles/byte) of STL introsort, radix sort and our AVX2 quicksort algorithms on Intel® Core i7-4770.

Performance in cycles/byte	32-Bit signed	32-Bit real	64-Bit real
STL	338	325	183
IPP	530	341	556
AVX2 quicksort	114	137	89

The size of sorted data is half of the size of the physical memory.

7. CONCLUSION

We introduced the first Quicksort implementation that leverages SIMD instructions for significant performance gains, for a variety of data types. The performance benefits are demonstrated on the Haswell architecture, and show consistent higher performance when compared with STL implementation, making it the fastest implementation of quicksort to date. Although it loses to the radix sort implementation of IPP, for some data types, our implementation outperforms the IPP implementation when the data and the auxiliary array do not fit in memory. Moreover, our method can be tailored for any data type, with a small effort, whereas IPP only supports a small set of basic types and the performance of radix sort drops significantly with longer keys.

We conclude with the potential benefits of the recently introduced AVX512 instruction set [18] that introduce 512-bit registers. Such architecture allows for operating on twice as many elements in parallel. In addition, AVX512 introduces new

instructions that can store sparse elements sequentially in memory (VCOMPRESSPS and VCOMPRESSPD), using a special mask register. This eliminates the need for using lookup tables, while the AVX512 comparison instructions can receive a special mask register as the destination, and the output is exactly the required mask. Therefore, the AVX-512 architecture has the potential to further speed up our quicksort algorithm by a significant amount.

ACKNOWLEDGEMENTS

We thank two anonymous referees for their constructive comments. The work of V.K. was carried out during his Intel employment period.

FUNDING

S.G. was partially funded by the European Commission Horizon 2020 Research Programme, grant #645622 for the PQCRYPTO project

REFERENCES

[1] de Wiel, M.V. and Daer, H. (2005) Sort Performance Improvements in Oracle Database 10g Release 2, An Oracle White Paper, Oracle.

[2] Fuguo, D., Hui, F. and Da, Y. (2010) A novel image median filtering algorithm based on in-complete quick sort algorithm. *Int. J. Digital Content Technol. Appl.*, **4**, 79–84.

[3] Hoare, C.A.R. (1962) Quicksort. *Comp. J.*, **5**, 10–16.

[4] Parikh, R. (2008) Accelerating QuickSort on the Intel Pentium 4 Processor with Hyper-Threading Technology, Intel.

[5] Tsigas, P. and Zhang, Y. (2003) A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 1000. *Proc. 11th Euromicro Conf. on Parallel Distributed and Network based Processing*, Genoa, Italy, February 5–7, pp. 372–384. IEEE Computer Society Washington, DC, USA.

[6] Kaligosi, K. and Sanders, P. (2006) How Branch Mispredictions Affect Quicksort. *Proc. 14th Annual European Symposium*, Zurich, Switzerland, September 11–13, pp. 780–791. Springer, Berlin, Heidelberg.

[7] Sanders, P. and Winkel, S. (2004) Super Scalar Sample Sort. *European Symposium on Algorithms*, Bergen, Norway, September 14–17, pp. 784–796. Springer LNCS.

[8] Cederman, D. and Tsigas, P. (2009) GPU-Quicksort: a practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, **14**, 4–24.

[9] Intel® Advanced Vector Extensions (Intel® AVX), Intel. http://software.intel.com/en-us/intel-isa-extensions (accessed 28 May, 2015).

[10] Inoue, H., Moriyama, T., Komatsu, H. and Nakatani, T. (2007) AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors. *Proc. 16th Int. Conf. on Parallel Architecture and Compilation Techniques*, Brasov, Romania, September 15–19, pp. 189–198. IEEE Computer Society, Washington, DC, USA.

[11] Chhugani, J. *et al.* (2008) Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture. *Proc. of the VLDB Endowment*, Auckland, New Zealand, August 24–30, Vol. 1, 1313–1324. VLDB Endowment. http://pcl.intel-research.net/publications/sorting_vldb08.pdf (accessed 28 May, 2015).

[12] Satish, N. *et al.* (2010) Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. *Proc. 2010 ACM SIGMOD Int. Conf. on Management of Data*, Indianapolis, IN, USA, June 6–11, pp. 351–362. ACM, New York, NY, USA.

[13] Furtak, T., Amaral, J.N. and Niewiadomski, R. (2007) Using SIMD Registers and Instructions to Enable Instruction-level Parallelism in Sorting Algorithms. *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, San Diego, CA, USA, June 9–11, pp. 348–357. ACM, New York, NY, USA.

[14] Buxton, N. (2011) Haswell new instruction descriptions now available. Intel. http://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available (accessed 28 May, 2015).

[15] Intel® Integrated Performance Primitives (IPP), Intel. https://software.intel.com/en-us/intel-ipp (accessed 28 May, 2015).

[16] The GNU C Library (glibc), http://www.gnu.org/software/libc/libc.html (accessed 28 May, 2015).

[17] The GNU C++ Library, http://gcc.gnu.org/onlinedocs/libstdc++/ (accessed 28 May, 2015).

[18] Reinders, J. (2013) AVX-512 instructions. Intel. http://software.intel.com/en-us/blogs/2013/avx-512-instructions (accessed 28 May, 2015).

APPENDIX

```
.Lqsort_int_avx_loop:
    cmp $4, n
    jl  .Lqsort_int_avx_loop_exit
    # Load the next 4 elements
    vmovdqu 16*0(array), A0
    # Find elements lesser-than-equal to P
    vpcmpgtd    A0, P, M0
    # Extract mask to gpr
    vmovmskps   M0, r0
    # Multiply by 16  the size of each mask
    shl $4, r0
    # First shuffle
    vpermilps .LPermTableLesser(,r0), A0, M0
    # Second shuffle
    vpermilps .LPermTableGreater(,r0), A0, A0
    # Count number of elements
    popcnt  r0, r0
    mov $4, r1
    sub r0, r1
    vmovdqu M0, (bottom)
    vmovdqu A0, (top)
    lea (bottom,r0,4), bottom
    lea (top,r1,4), top
    sub $4, n
    add $4*4, array
    jmp .Lqsort_int_avx_loop
.Lqsort_int_avx_loop_exit:
```

FIGURE A1. A code snippet (AT&T asm syntax) that demonstrates the main loop of our quicksort implementation using AVX instructions.

```
.align 16                 |   align 16
.LPermTableLesser:        |   .LPermTableGreater:
.long 0,0,0,0 #0          |   .long 0,1,2,3 #0
.long 0,0,0,0 #1          |   .long 1,2,3,0 #1
.long 1,0,0,0 #2          |   .long 0,2,3,0 #2
.long 0,1,0,0 #3          |   .long 2,3,0,0 #3
.long 2,0,0,0 #4          |   .long 0,1,3,0 #4
.long 0,2,0,0 #5          |   .long 1,3,0,0 #5
.long 1,2,0,0 #6          |   .long 0,3,0,0 #6
.long 0,1,2,0 #7          |   .long 3,0,0,0 #7
.long 3,0,0,0 #8          |   .long 0,1,2,0 #8
.long 0,3,0,0 #9          |   .long 1,2,0,0 #9
.long 1,3,0,0 #10         |   .long 0,2,0,0 #10
.long 0,1,3,0 #11         |   .long 2,0,0,0 #11
.long 2,3,0,0 #12         |   .long 0,1,0,0 #12
.long 0,2,3,0 #13         |   .long 1,0,0,0 #13
.long 1,2,3,0 #14         |   .long 0,0,0,0 #14
.long 0,1,2,3 #15         |   .long 0,0,0,0 #15
```

FIGURE A2. The lookup tables, used for sorting 32-bit elements.