Engineering a Multi-core Radix Sort

Jan Wassenberg[1] and Peter Sanders[2]

[1] Fraunhofer IOSB, Ettlingen, Germany
jan.wassenberg@iosb.fraunhofer.de
[2] Karlsruhe Institute of Technology, Karlsruhe, Germany
sanders@kit.edu

Abstract. We present a fast radix sorting algorithm that builds upon a microarchitecture-aware variant of counting sort. Taking advantage of virtual memory and making use of write-combining yields a per-pass throughput corresponding to at least 89% of the system's peak memory bandwidth. Our implementation outperforms Intel's recently published radix sort by a factor of 1.64. It also compares favorably to the reported performance of an algorithm for Fermi GPUs when data-transfer overhead is included. These results indicate that scalar, bandwidth-sensitive sorting algorithms remain competitive on current architectures. Various other memory-intensive applications can benefit from the techniques described herein.

1 Introduction

Sorting is a fundamental operation that is a time-critical component of various applications such as databases and search engines. The well-known lower bound of $\Omega(N \cdot \log N)$ for comparison-based algorithms no longer applies when special properties of the keys can be assumed. In this work, we focus on 32-bit integer keys, optionally paired with a 32-bit (or larger) value. This simplifies the implementation without loss of generality, since applications can often replace large records with a pointer or index [1]. The radix sort algorithm is commonly used in such cases due to its $O(n)$ complexity. In this report, we show a 1.64-fold performance increase over results recently published by Intel [2].

The remaining sections are organized in a bottom-up fashion, with Section 2 dedicated to the basic realities of current and future microarchitectures that affect memory-intensive programs and motivate our approach. We build upon this foundation in Section 3, showing how to speed up counting sort by taking advantage of virtual memory and write-combining. Section 4 applies this technique towards a novel variant of radix sort. The performance of our implementation is evaluated in Section 5. Bandwidth measurements indicate the per-pass throughput is nearly optimal for the given hardware. Its two CPUs outperform a Fermi GPU when accounting for data-transfer overhead.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6853, Part II, pp. 160–169, 2011.

2 Software Write-Combining

We begin with a description of basic microarchitectural realities that are likely to have a serious impact on applications with numerous memory accesses, and show how to avoid performance penalties by means of Software Write-Combining. These topics are not new, but we believe they are often not adequately addressed.

The first problem arises when writing items to multiple streams. An ideal cache with at least as many lines could exploit the writes' spatial locality and entirely avoid noncompulsory misses. However, perfect hit rates are not achievable in practice due to limited ways of associativity a [3]. Since only a lines can be mapped to a cache set, any further allocations from that set result in the eviction of one of the previous lines. If possible, applications should avoid writing to many different streams. Otherwise, the various write positions should map to different sets to avoid thrashing and conflict misses. For current L1 caches with $a = 8$ ways, size $C = 32$ KiB and lines of $B = 64$ bytes, there are $S = \frac{C}{a \cdot B} = 64$ sets, and bits $[\lg B, \lg B + \lg S)$ of the destination addresses should differ (e.g. by ensuring the write positions are not a multiple of $S \cdot B = 4$ KiB apart).

A second issue is provoked by a large number of write-only accesses. Even if an entire cache line is to be written, the previous destination memory must first be read into the cache. While the corresponding latency may be partially hidden via prefetching, the cache line allocations remain problematic due to capacity constraints and eviction policy. Instead of displacing write-only lines that are not accessed after having been filled, the widespread (pseudo-)Least-Recently-Used strategy displaces previously cached data due to their older timestamp. An attempt to avoid these evictions by explicitly invalidating cache lines (e.g. with the IA-32 CLFLUSH instruction) did not yield meaningful improvements. Instead, applications should use *non-temporal streaming store* instructions that write directly to memory. These are guaranteed to avoid cache pollution since they circumvent the cache.

This leads directly to the next concern: single memory accesses involve significant bus overhead. The architecture therefore combines neighboring non-temporal writes into a single burst transfer. However, currently microarchitectures only provide four to ten write-combine (WC) buffers [4]. Non-temporal writes to multiple streams may force these buffers to be flushed to memory via 'partial writes' before they are full. The application can prevent this by making use of Software Write-Combining [5]. The data to be written is first placed into temporary buffers, which almost certainly reside in the cache because they are frequently accessed. When full, a buffer is copied to the actual destination via consecutive non-temporal writes, which are guaranteed to be combined into a single burst transfer.

This scheme avoids reading the destination memory, which may incur relatively expensive Read-For-Ownership transactions and would only pollute the cache. It works around the limited number of WC buffers by using L1 cache lines for that purpose. Interestingly, this is tantamount to direct software control of the transparently managed cache.

We recommend the use of such Software Write-Combining whenever a core's active write destinations outnumber its write-combine buffers. Fortunately, this can be done at a fairly high level, since only the buffer copying requires special vector loads and non-temporal stores (which are best expressed by the SSE2 intrinsics built into the major compilers).

3 Virtual-Memory Counting Sort

We now review Counting Sort of N elements with keys in $[0, M)$ and describe an improved variant that makes use of virtual memory and write-combining.

The naïve algorithm first generates a histogram of the N keys. After computing the prefix sum to yield the starting output location for each key, each value is written at its key's output position, which is subsequently incremented.

Our first optimization goal is to avoid the initial counting pass. We could instead insert each value into a per-key container, e.g. a list of data blocks. However, this incurs some overhead for checking whether the current bucket is full. Preallocating space for M arrays of size N is more efficient, because items can simply be written to the next free position (c.f. Algorithm 1, introduced in [6]). This algorithm only writes and reads each item once, a feat that comes at

Algorithm 1: Single-pass counting sort

```
storage := ReserveAddressSpace(N · M);
for i := 0 to M − 1 do next[i] := i · N;
foreach key,value do
    storage[next[key]] := value;
    next[key] := next[key] + 1;
```

the price of $N \cdot M$ space. While this appears problematic in the Random-Access-Machine model, it is easily handled by 64-bit CPUs with paged virtual memory. Physical memory is only mapped to pages when they are first accessed,[1] thus reducing the actual memory requirements to $O(N+M \cdot \text{pageSize})$. The remainder of the initial allocation only occupies address space, of which multiple terabytes are available on 64-bit systems.

Having avoided the initial counting pass, we now show how to efficiently write values to storage using the write-combining technique described in Section 2. Our implementation initializes the next pointers to consecutive, naturally aligned, cache-line-sized buffers. A buffer is full when its (post-incremented) position is evenly divisible by its size. When that happens, an unrolled loop of non-temporal writes copies the buffer to its key's current output position within storage. These output positions are also stored in an array of pointers.

[1] Accesses to non-present pages result in a page fault exception. The application receives such events via signals (POSIX) or Vectored Exception Handling (Microsoft Windows) and reacts by committing memory, after which the faulting instruction is repeated.

4 Radix Sort

After a brief review of radix sorting, we introduce a new variant based on the virtual-memory counting sort described in Section 3.

A radix sort successively examines D-bit 'digits' of the K-bit keys. They are characterized by the order in which digits are processed: starting at the Least Significant Digit (LSD), or Most Significant Digit (MSD).

An MSD radix sort partitions the items according to the current digit, then recursively sorts the resulting buckets. While it no longer needs to move items whose previously seen key digits are unique, this is not especially helpful when the number of passes K/D is small. In fact, the overhead of managing numerous (nearly empty) buckets makes MSD radix sort less suited for relatively small N.

By contrast, each iteration of the LSD variant partitions *all* items into buckets by the current key digit. This amortizes the bucket setup cost over the number of elements and avoids the possibility of load imbalance for parallelization at the price of increased data copying.

To reduce this overhead and also parallel communication, we make use of "reverse sorting" [7], in which one or more MSD passes partition the data into buckets, which are then locally sorted via LSD. This turns out to be even more advantageous for Non-Uniform Memory Access (NUMA) systems because each processor is responsible for writing a contiguous range of outputs, thus ensuring the OS allocates those pages from the processor's NUMA node [8].

Let us now examine the pseudocode of the radix sort (Algorithm 2), choosing $K = 32$ for brevity and $D = 8$ to allow extracting key digits without masking. Each Processing Element (PE) first uses counting sort to partition its items into local buckets by the MSD (digit = 3). Note that items consist of a key and value, which are adjacent in memory (ideally within a native 64-bit word, but larger combinations are possible in our implementation via larger user-defined types). After all are finished, the output index of the first item of a given MSD is computed via prefix sum. Each PE is assigned a range of MSD values, sorting the buckets from all PEs for each value. Skewed MSD distributions can cause load imbalance. However, this could be resolved via special treatment of large buckets[2]. The local sort entails $K/D-1$ iterations in LSD order. The first copies all other PEs' buckets into local memory. The second to last pass also computes the last digit's histogram, thus allowing writing directly to the output positions in the final pass. Note that three sets of buckets are required, which makes heavy use of virtual memory ($3 \cdot 2^D \cdot |\mathrm{PE}| = 6144$ times the input size). While 64-bit Linux grants each process 128 TiB address space, Windows limits this to 8 TiB, which means only about 1.4 GiB of inputs can be sorted[3].

We briefly discuss additional system-specific considerations. The radix 2^D was motivated by easy access to each digit, but is also limited by the cache

[2] Sorting buckets larger than $N/|\mathrm{PE}|$ using multiple PEs.

[3] This limitation could be circumvented by estimating bounds for bucket sizes via sampling. In the unlikely case that they are exceeded, a new sample would be drawn and the process repeated.

Algorithm 2: Parallel Radix Sort

```
parallel foreach item do
    d := Digit(item, 3);
    buckets3[d] := buckets3[d] ∪ {item};
Barrier;
foreach i ∈ [0, 2^D) do
    bucketSizes[i] := Σ_PE |buckets3[i]|;
outputIndices := PrefixSum(bucketSizes);
parallel foreach bucket3 ∈ buckets3 do
    foreach item ∈ bucket3 ∀ PE do
        d := Digit(item, 0);
        buckets0[d] := buckets0[d] ∪ {item};
    foreach bucket0 ∈ buckets0 do
        foreach item ∈ bucket0 do
            d := Digit(item, 1);
            buckets1[d] := buckets1[d] ∪ {item};
            d := Digit(item, 2);
            histogram2[d] := histogram2[d] + 1;
    foreach bucket1 ∈ buckets1 do
        foreach item ∈ bucket1 do
            d := Digit(item, 2);
            i := outputIndices[d] + histogram2[d];
            histogram2[d] := histogram2[d] + 1;
            output[i] := item;
```

and TLB size. Because of the many required TLB entries, we map the buckets with small pages, for which the Intel i7 microarchitecture has 512 second-level TLB entries. To increase TLB coverage, we use large pages for the inputs. The working set consists of 2^D buffers, buffer pointers, output positions, and 32-bit histogram counters. This fits in a 32 KiB L1 data cache if the software write-combine buffers are limited to a single 64-byte cache line. To avoid associativity and aliasing conflicts, these arrays are contiguous in memory. Interestingly, these optimizations do not detract from the readability of the source code. Knowledge of the microarchitecture can also be applied towards middle-level languages and enables principled design decisions.

5 Performance Evaluation

We characterize the performance of our sorting implementation by its throughput, defined as $\frac{N}{t_1 - t_0}$, where N is the number of items and t_0 and t_1 are the earliest and latest start and finish times reported by any thread. The test platform consists of dual W5580 CPUs (3.2 GHz, 48 GiB DDR3-1066 memory) running Windows XP x64. Our implementation is compiled with ICC 11.1.082 `/Ox /Og /Oi /Ot /Qipo /GA /GR- /GS- /EHsc /Qopenmp /QaxSSE4.2`. When sorting 350 M

uniformly distributed 32-bit keys generated by the WELL512 algorithm [9], the basic algorithm ('VM only') reaches a throughput of 391 M items/s, as shown in the second column of Table 1. After enabling write-combining ('VM+WC'), performance nearly doubles to 657 M/s.

Intel has reported 240 M/s for the same task and a single but identical CPU [2]. For a fair comparison with our dual-CPU system, we double their throughput, which optimistically assumes their algorithm is NUMA-aware, scales perfectly and is not running at a lower memory clock (since our DDR3-1066 is at the lower end of currently available frequencies). We must also divide by the given speedup of 1.2 due to hyperthreads, since those are disabled on our machine. This ('Intel x2') yields 400 M/s; the proposed algorithm is therefore 1.64 times as fast. A separate publication has also presented results [10] for the Many Integrated Cores architecture. The Knights Ferry processor provides 32 cores, each with 4 threads and 16-wide SIMD. The simulation ('KNF MIC') shows a throughput of 560 M/s. Our scalar implementation is currently 1.17 times as fast when running on 8 cores.

Recently, a throughput of 1005 M/s was reported on a GTX 480 (Fermi) GPU [11]. However, this excludes driver and data-transfer overhead. For applications in which the data is generated and consumed by the CPU, we must include at least the time required to read and write data over the PCIe 2.0 bus. Assuming the peak per-direction bandwidth of 8 GB/s is reached, the aggregate throughput ('GPU+PCIe') is 501 M/s. Our implementation, running on two CPUs, therefore outperforms this algorithm on a current top-of-the-line GPU by a factor of 1.31 despite lower transistor counts ($2 \cdot 731$ M vs. 3000 M) and thermal design power ($2 \cdot 130$ W vs. $275 - 300$ W).

Table 1. Throughputs [million items per second] for 32-bit keys and optional 32-bit values

Algorithm	K=32,V=0	K=32,V=32
VM only	391	238
Intel x2	400	307
GPU+PCIe	501	303
KNF MIC	560	(?)
VM+WC	657	452

Similar measurements and extrapolations for the case of 32-bit keys associated with $V = 32$-bit values are given in the third column of Table 1. Since the slowdown is less than a factor of two, the implementations are at least partially limited by computation instead of bandwidth. Intel's algorithm is more efficient in this regard, with only a 1.3-fold decrease vs. our factor of 1.45. The additional data transfers over PCIe render the GPU algorithm uncompetitive.

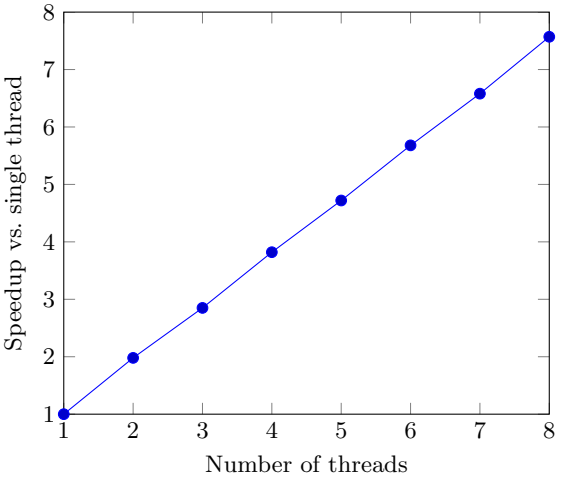

Fig. 1. Linear scalability on two quad-core CPUs with a NUMA factor of 1.5

Since radix sort is bandwidth-sensitive, it is also interesting to examine performance for a varying number of processors. We manually distribute OpenMP threads across CPU packages and cores (in that order) to make use of all available memory controllers. Our NUMA-aware implementation scales linearly with the number of threads, as shown by Figure 1.

To explain the 95% parallel efficiency, we measured the total traffic at each socket's memory controller. Since this information is not available from current profilers such as VTune (which use per-core performance counters), we have developed a small kernel-mode driver to provide access to the model-specific performance counters in the Intel i7 uncore[4]. Uncached writes constitute the bulk of the write combiners' memory traffic and are therefore of particular interest. They are apparently reported as Invalid-To-Exclusive transitions and can thus be counted as the total number of *reads* minus 'normal' reads [12]. We find that 2041 MiB are written, which corresponds to 64 Mi items · 8 bytes per item · 4 passes (slightly less because our final pass cannot use non-temporal writes when the output position is not aligned). Surprisingly, 2272 MiB are read – about 10% more than expected. This amount seems to be influenced by the number of threads. Possible causes may include coherency traffic or page walks and will be investigated in future work. However, we can provide a conservative estimate of the bandwidth utilization. Given the pure read and write bandwidths (38687 MB/s and 28200 MB/s) measured by RightMark [13], the minimum time required for 4 reads and writes of 175 M 8-byte items is 343 ms, which is 89% of the total measured time. This calculation does not include write-to-read turnaround [14, p. 486], so there is even less room for improvement than indicated.

[4] The part of the socket not associated with a particular core.

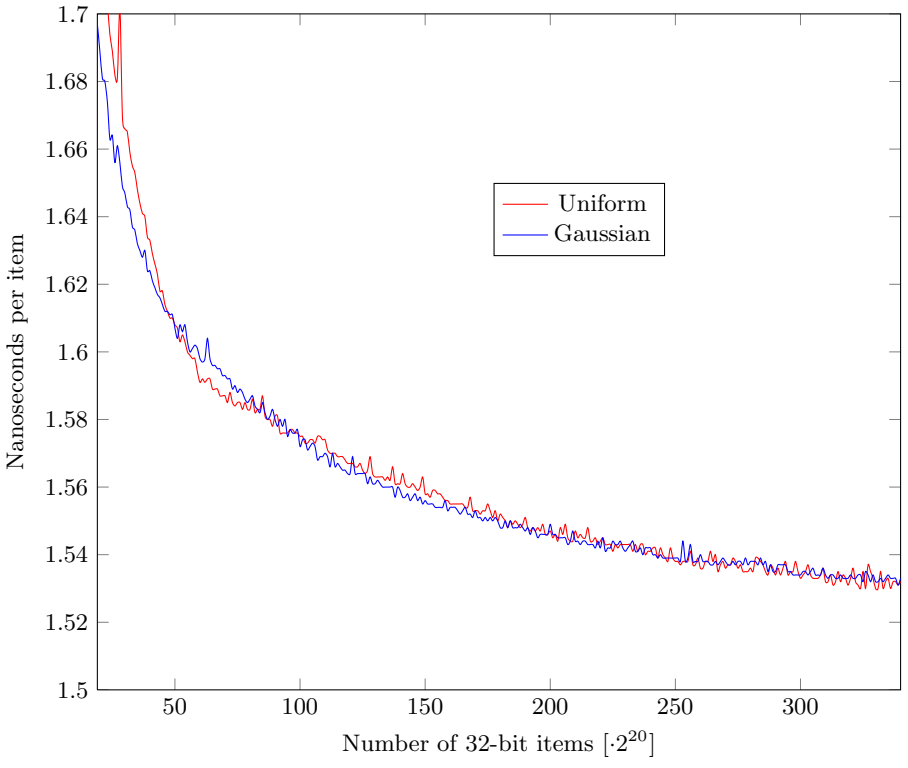

Fig. 2. Time per item for various input sizes and distributions

The previous measurements concern large numbers of items. We now study performance over a wider range of input sizes. The elapsed time per item, shown in Figure 2, varies inversely with the number of items N due to amortization of thread-startup overhead. Performance is within 10% of the best measurement when $N \geq 26 \cdot 2^{20}$, or $N \geq 21 \cdot 2^{20}$ in the case of the approximated Gaussian distribution [15]. It is initially surprising that this distribution does not require more time to sort than uniformly distributed numbers. However, interleaving buckets in the LSD passes (successive buckets are assigned to different threads) avoids load imbalance, and increased occupancy of the central buckets improves locality at the memory page level.

6 Conclusion

We have introduced improvements to counting sort and a novel variant of radix sort for integer key/value pairs. Bandwidth measurements indicate our algorithm's throughput is within 11% of the theoretical optimum for the given hardware. It outperforms the recently published results of Intel's radix sort by a factor of 1.64 and also outpaces a Fermi GPU when data transfer overhead is included.

These results indicate that scalar, bandwidth-sensitive sorting algorithms still have their place on current architectures. However, achieving this level of performance requires awareness of the underlying microarchitecture and some degree of tuning. Our implementation encompasses 5700 lines of C++ (including tests), plus 40,000 lines of shared infrastructure. A demo executable [16] capable of generating or reading 32-bit integers, sorting and efficiently writing them to disk is being made available so that our measurements may be reproduced.

Future Work: While carefully engineered, our implementation is not yet a general solution for all possible sorting applications. Radix sort is limited to relatively small integer keys, and we also assume at least one of the key digits (the MSB) is reasonably equally distributed. Skewed (e.g. constant) distributions currently result in load imbalance. This could be avoided by sorting extremely large buckets from the MSD phase using multiple processors.

We are also interested in testing on larger multi-socket machines with higher NUMA factors and investigating details of the memory subsystem that reduce effective bandwidth. Finally, we believe the general software write-combining technique can provide similar speedups for other memory-intensive applications. In particular, comparison-based sample sort is also expected to benefit from our implementation techniques.

References

1. Bohannon, P., McIlroy, P., Rastogi, R.: Main-memory index structures with fixed-size partial keys. In: SIGMOD Conference, pp. 163–174 (2001), http://www.acm.org/sigs/sigmod/sigmod01/eproceedings/papers/Research-Bohannon-et-al.pdf
2. Satish, N., Kim, C., Chhugani, J., Nguyen, A., Lee, V., Kim, D., Dubey, P.: Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In: Elmagarmid, A., Agrawal, D. (eds.) SIGMOD Conference, pp. 351–362. ACM Press, New York (2010), http://doi.acm.org/10.1145/1807167.1807207
3. Mehlhorn, Sanders: Scanning multiple sequences via cache memory. Algorithmica 35 (2003)
4. Intel. Intel Architecture Software Developer Manual (2010), System Programming Guide, http://www.intel.com/Assets/PDF/manual/253668.pdf
5. Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual (November 2007), http://www.intel.com/design/processor/manuals/248966.pdf
6. Wassenberg, J., Middelmann, W., Sanders, P.: An efficient parallel algorithm for graph-based image segmentation (June 2009), http://algo2.iti.uni-karlsruhe.de/wassenberg/wassenberg09parallelSegmentation.pdf
7. Jimenez-Gonzalez, D., Navarro, J., Larriba-Pey, J.: Fast parallel in-memory 64-bit sorting. In: Proceedings of the 2001 International Conference on Supercomputing (15th ICS 2001), Sorrento, Napoli, Italy, pp. 114–122. ACM, New York (2001)

8. an Mey, D., Terboven, C.: Affinity matters! OpenMP on multicore and ccNUMA architectures. In: Parallel Computing: Architectures, Algorithms and Applications, vol. 15, Forschungszentrum Jülich and RWTH Aachen University (Febuary 2008), http://www.compunity.org/events/pastevents/parco07/AffinityMatters_DaM.pdf
9. Panneton, F., L'Ecuyer, P., Matsumoto, M.: Improved long-period generators based on linear recurrences modulo 2. ACM Transactions on Mathematical Software 32 (2006)
10. Satish, N., Kim, C., Chhugani, J., Nguyen, A., Lee, V., Kim, D., Dubey, P.: Fast sort on CPUs, GPUs and intel MIC architectures. Technical report, Intel (2010), http://techresearch.intel.com/userfiles/en-us/FASTsort_CPUsGPUs_IntelMICarchitectures.pdf
11. Merrill, D., Grimshaw, A.: Revisiting sorting for GPGPU stream architectures. Technical Report 3, University of Virginia (February 2010), http://www.cs.virginia.edu/~dgm4d/papers/RadixSortTR.pdf
12. Levinthal, D.: Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors. Intel, http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf
13. Besedin, D.: RightMark memory analyzer, http://cpu.rightmark.org (accessed January 9, 2009)
14. Jacob, B., Ng, S., Wang, D.: Memory systems: cache, DRAM, disk. Morgan Kaufmann, San Francisco (2007)
15. Helman, D., Bader, D., JáJá, J.: A randomized parallel sorting algorithm with an experimental study. J. Parallel Distrib. Comput. 52(1), 1–23 (1998)
16. Wassenberg, J.: Vmcsort demo (May 2011), http://algo2.iti.kit.edu/wassenberg/vmcsort/demo.html