# Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort

Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen,
Victor W. Lee, Daehyun Kim, and Pradeep Dubey

nadathur.rajagopalan.satish@intel.com

Throughput Computing Lab,
Intel Corporation

## ABSTRACT

Sort is a fundamental kernel used in many database operations. In-memory sorts are now feasible; sort performance is limited by compute flops and main memory bandwidth rather than I/O. In this paper, we present a competitive analysis of comparison and non-comparison based sorting algorithms on two modern architectures - the latest CPU and GPU architectures. We propose novel CPU radix sort and GPU merge sort implementations which are 2X faster than previously published results. We perform a fair comparison of the algorithms using these best performing implementations on both architectures. While radix sort is faster on current architectures, the gap narrows from CPU to GPU architectures. Merge sort performs better than radix sort for sorting keys of large sizes - such keys will be required to accommodate the increasing cardinality of future databases. We present analytical models for analyzing the performance of our implementations in terms of architectural features such as core count, SIMD and bandwidth. Our obtained performance results are successfully predicted by our models. Our analysis points to merge sort winning over radix sort on future architectures due to its efficient utilization of SIMD and low bandwidth utilization. We simulate a 64-core platform with varying SIMD widths under constant bandwidth per core constraints, and show that large data sizes of $2^{40}$ (one trillion records), merge sort performance on large key sizes is up to 3X better than radix sort for large SIMD widths on future architectures. Therefore, merge sort should be the sorting method of choice for future databases.

## Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

## General Terms

Performance, Algorithms

## 1. INTRODUCTION

Sorting is of fundamental importance in databases. Common applications of sorting in database systems include index creation, user-requested sort queries, and operations such as duplicate removal, ranking and merge-join operations. Sorting on large databases has traditionally focused on external sorting algorithms. However, the rapid increase in main memory capacity has made in-memory sorting feasible.

In-memory sorts are bounded by compute, bandwidth and latency characteristics of processor architectures. Recent and future trends in modern computer architectures are therefore of primary importance for high performance sort implementations. Compute capacity has increased through a combination of having more cores (thread-level parallelism) with each core having wide vector (SIMD) units to exploit data-level parallelism. Core counts will increase rapidly as Moore's law continues to increase the number of on-chip transistors. The SIMD width of modern CPU and GPU processors has been steadily increasing - from 128-bit in SSE architectures, 256-bit in AVX [14] to 512-bit in the upcoming Larrabee [23] architecture. GPUs have a logical 1024-bit SIMD with physical SIMD widths of 256-bits on the latest NVIDIA GTX 200 series, increasing to 512-bits on the upcoming Fermi architecture [19]. Memory bandwidth is increasing at a slower pace than compute. Algorithms that are bound by memory bandwidth will not scale well to future architectures.

A variety of algorithms have been developed for sorting a list of numbers. Sorting algorithms can be broadly classified as either comparison based or non-comparison based sorts. Comparison based sorting algorithms rearrange the data items based on the results of comparing pairs of elements at a time. Non-comparison based sorts rely on using the absolute values of the data items, rather than comparisons, to rearrange the data. A common example of a non-comparison based sort is radix sort. Radix sort is a multiple pass sort algorithm that buckets data according to individual digits of the data items. Sorting algorithms differ in their **computational complexity**, which dictates the inherent amount of computation required by the algorithm, and also differ in their **architectural friendliness**, or how well they can use current and future architectural trends, such as increasing thread-level and data-level parallelism on modern architectures. There is often a trade-off between these factors. For instance, **radix sorts** are not naturally data-parallel, unlike comparison sorts that can use data parallel merging networks. Further, radix sort needs many passes over each data item - resulting in high bandwidth utilization. **Merge sort**, on the other hand, can be made bandwidth friendly (and is used in external disk sorts for this reason). Opposing these architectural inefficiencies of radix sort is its **lower computational complexity** of $O(N)$, as against the lower bound $\Omega(N \log N)$ of comparison based sorts. The right choice of sorting algorithms becomes a trade-off between computational complexity and architectural efficiency; such trade-offs are architecture-dependent. There has, so far, not

*SIGMOD'10,* June 6–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0032-2/10/06 ...$10.00.

been much work in analyzing the efficiency with which sorting techniques utilize modern architectural features. In this work, we evaluate these trade-offs on two sorting algorithms - radix sort with a low $O(N)$ computational complexity, and a SIMD-efficient and bandwidth oblivious merge sort with a complexity of $O(N \log N)$. We investigate these trade-offs on the latest CPU and GPU architectures, which are commercially widespread and hence of interest to the database community. The Intel Core i7 CPU has 4 cores each with 128-bit SSE width, while the NVIDIA GTX 280 has 30 SMs each with 256-bit SIMD units.

In order to make such an investigation fair, the algorithms must be implemented as efficiently as possible on each architecture. We, therefore, identified bottlenecks found in common implementations of each sorting algorithm (such as irregular memory accesses, lack of SIMD use, conflicts in local storage such as cache or shared memory) and optimized our algorithms to avoid such bottlenecks. Our novel CPU radix sort algorithm uses a buffer stored in cache to localize scatters; this avoids capacity and conflict misses, resulting in the **fastest** reported CPU sorts. We implemented an optimized merge sort on the GPU which is only 10% off the best known sorting algorithm (a radix sort) on the GPU. We adopted highly optimized codes where available, including a CPU merge sort and a GPU radix sort. Our resulting implementations are the best performing implementations of these sorting algorithms on these architectures, and include the highest performing sort on each architecture. We provide a **performance model** for each of our resulting implementations in terms of the compute, bandwidth and SIMD usage of our algorithms. Our analysis results match well with the actual performance results.

The influence of architectural awareness on sorting algorithms is clear from our investigations. The best implementation of the same radix sort algorithm is very different on CPUs and GPUs - a SIMD friendly 1-bit split based code is best on GPUs that heavily rely on data-level parallelism, while a scalar buffer-based scatter approach works best on CPUs with lower SIMD widths. However, we show that both versions of radix sort have heavy bandwidth demands due to their multi-pass nature. The bandwidth demand of radix sort increases for large key sizes - radix becomes **bandwidth bound** on even the latest CPUs on **6-byte keys**. GPUs have much higher bandwidth - however, even then a part of the radix sort algorithm is bandwidth bound. Algorithms that are bandwidth bound cannot use compute resources effectively. On the other hand, merge sort uses bandwidth resources efficiently, and is compute-bound. Merge sort hence becomes faster than radix on GPUs on keys larger than 8-bytes, and faster on CPUs for keys greater than 9-bytes. In addition, merge sort can also utilize SIMD resources efficiently. While the higher computational complexity of merge sort does make it slower than radix on current architectures for large data sets of 4-byte keys, the gap narrows from 1.7X on CPUs with 128-bit SIMD to only about 10% on GPUs with 256-bit SIMD.

Having identified the bottlenecks of these algorithms on current hardware, we project the performance of our algorithms on future architectures with wider SIMD and lower bandwidth-to-compute requirements. The bandwidth-oblivious SIMD-friendly merge sort will perform better on such architectures. We confirm our projections by simulating our algorithms on architectures with varying SIMD widths, and show that as SIMD widths increase to 2048-bit and beyond, SIMD merge sort performance for 8-byte keys is 1.5X faster than radix sort on data sizes as large as $2^{40}$ (one trillion records). For 16-byte keys, the performance ratio further increases to 3X better than radix sort. The **bandwidth-oblivious SIMD-friendly** merge sort, should, therefore, be the sorting method of choice for future databases.

## 2. RELATED WORK

Several implementations of comparison and non-comparison sorts on multi-core and many-core processors with SIMD support have been reported in the literature. Bitonic sort [4], which allows for a straightforward use of SIMD (albeit at a higher algorithmic complexity of $O(N \log^2 N)$ as compared to the $O(N \log N)$ of merge and quick sorts), has been implemented on many architectures, including GPUs (one example is GPUTeraSort [12]). Before the advent of scatter functionality and local stores on GPUs, bitonic sort was well suited for GPU implementations.

After recent improvements in GPU technology, other comparison sorts with lower algorithmic complexity of $O(N \log N)$ such as merge sort and sample sort have become viable. Implementations of sample sort [18], merge sort [22] and hybrids of radix and merge sort [25] has been reported on GPUs. A combination of merge sort and comb-sort (AA-Sort) has also been developed for the PowerPC 970MP [13]. AA-Sort consists of two passes: in the first pass, each thread performs a local comb-sort, the results of which are merged in the second pass. On multi-core CPUs, the merge sort implementation by Chhugani et al. [9] has the best reported comparison sort performance to date. The authors utilize an efficient $O(N \log N)$ algorithm and use available parallelism such as thread-level and data-level parallelism to achieve their performance. No analogous algorithm has been reported yet for GPUs. In this work, we adopt their algorithm as our basis and develop an optimized GPU implementation, resulting in the fastest reported GPU comparison sort.

Radix sort is a non-comparison based sort that has a lower algorithmic complexity of $O(N)$. However, radix sort has an irregular memory access pattern that results in misses to the cache and page hierarchy on modern CPUs, leading to wastage of memory bandwidth and increased memory latency. A cache-aware version of radix sort (CC-Radix sort) [15] was reported on the MIPS R10000 processor. The Intel Performance Primitives library [2] has a highly optimized version of radix sort for the Intel x86 architecture. However, both these implementations while cache-conscious do not account for cache conflict misses. LaMarca et al. [17] show that conflict misses become a significant source of cache misses in traditional radix sort, especially at large radix values. In this work, we report an optimized version of radix sort that maintains a software-managed buffer in cache and is managed to avoid cache conflict misses; this is the fastest reported sort on CPU architectures.

Scatter based radix sorts, even to a local buffer, cannot efficiently use the data-level parallelism available in modern architectures. On GPUs, where using data-level parallelism is critical to performance, a different version of radix sort has been reported by Satish et al. [22]. This implementation reduces memory scatters based on locally sorting data in blocks; this sort is based on a sequence of 1-bit stream split operations that can be efficiently implemented on GPUs using scan primitives. This makes it the fastest reported GPU sort. There are tradeoffs between the buffer and local sort implementations of radix sort based on the SIMD width of the processor.

Recent advances in CPU and GPU architectures have resulted in sorting algorithms that have been optimized for each architecture. In this work, we evaluate the tradeoffs between comparison and non-comparison sorts on different architectures with varying SIMD width, project these findings to future architectures, and make a case for the SIMD-friendly bandwidth-oblivious merge sort.

## 3. PERFORMANCE OF SORTING ALGORITHMS ON MODERN PROCESSORS

While the theoretical computational complexity of different sorting algorithms has a definite impact on performance, the efficiency with which they use different hardware features is often decisive in terms of performance. With increasing memory sizes, in-memory databases are now common, and hence I/O is not a major constraint. The performance limitations have now shifted to the compute and main memory resources available in the architecture. We now illustrate some of the recent architectural advancements and their impact on sorting performance in the context of database operations.

## 3.1 Thread-Level and Data-Level Parallelism

Modern processors have increased compute power by (a) adding more cores to exploit thread-level parallelism, and (b) adding SIMD units to exploit data-level parallelism. Many thread parallel implementations of sorting algorithms have been proposed for different multi-core architectures. A number of sorts such as merge and quick sort naturally involve combining or splitting different blocks of data in parallel. Other sorts, such as radix sort, can be parallelized through blocking the input data and using global histogram updates to coordinate between the blocks (see Blelloch [6]). These have been shown to scale well on CPU and GPU architectures.

Data-level parallelism is harder to exploit in certain sorting algorithms. For efficient SIMD execution, the data to be operated on has to be contiguously laid out in memory. In the absence of contiguous accesses, gathers/scatters[1] to memory are required, which are slow on most architectures. Radix sort, for instance, generally involves a common histogram update and irregular memory accesses to rearrange the data; both of these are inherently unsuited to SIMD execution. Variants of radix sort that are more SIMD friendly have been proposed [27]; however they require many more instructions to execute. On the other hand, merge sort can use sorting networks that are SIMD friendly [9]. As SIMD widths have been increasing, SIMD-friendly sorts become increasing more efficient.

## 3.2 Memory Bandwidth

Sorting fundamentally involves rearranging data resident in memory and as such is typically memory intensive. The memory bandwidth has not improved as much as the computational capacity of modern processors, and the per-core bandwidth is expected to further drop in the future [21]. Algorithms that require high memory bandwidth are likely to become bandwidth bound and hence stop scaling with respect to number of cores and SIMD width.

In order to bridge the gap between application bandwidth requirements and available bandwidth, architectures have introduced on-chip local storage in the form of cache hierarchies on CPUs and shared memory on GPUs. If the data set being processed fits in this storage, no main memory bandwidth is utilized; otherwise data is read and written from main memory. Sorting algorithms have been redesigned to use such storage areas. Merge sort can be made very efficient in terms of bandwidth use, requiring just two reads and two writes of data through a multi-way merge implementation [11] that limits the working set to fit in caches. Radix sort is not as bandwidth-friendly. If the input data is much larger than cache size, each input key must therefore be read and written in each pass.

## 3.3 Latency Effects/ILP

Instructions with high latency can lead to low utilization of functional units since they block the execution of dependent instructions. This is typically due to long latency memory accesses caused by last-level cache misses. In addition to cache misses, misses to an auxiliary structure called the Translation Lookaside Buffer (TLB) – used to perform a conversion from virtual to physical memory addresses, can also result in significant performance degradation.

Caches and TLBs are organized such that they have minimal misses when physically contiguous regions of memory are accessed (called *streaming accesses*). Caches are organized into cache lines of 64 bytes, which is the unit of data transfers between the cache and main memory. When the access is streaming, consecutive accesses will belong to the same cache line and memory page, resulting in cache and TLB hits. Streaming accesses also minimize cache conflict misses, which occurs when different memory regions are mapped to the same cache line due to cache associativity and are alternately accessed - resulting in a series of cache misses. Caches are organized to minimize such misses when a contiguous region of memory of less than the cache size is accessed.

Algorithms that have streaming access patterns therefore have minimal impact from cache and TLB misses. Among sorting algorithms, merge sort has a streaming access pattern, resulting in low misses. Radix sort, on the other hand, has a data rearrangement step where contiguous inputs are written into widely scattered output locations. This can cause page misses, since each of these locations is likely to belong to different pages, as well as cache misses, since they will also belong to different lines. Cache-conscious radix sorts have been designed where the memory scattered writes are limited to a small number of pages and cache lines [15]. However, cache conflict misses are still possible. We propose and discuss a buffer scheme in this work to minimize such misses.

# 4. RADIX SORT

In this section, we describe the basic radix sort algorithm and its parallel implementation in Section 4.1. In Section 4.2, we then propose a *buffer-based* scheme for making radix sort architecture-friendly and also describe a previous local sort approach based on *stream splits*. We then describe the best CPU and GPU implementations of radix sort in Sections 4.3 and 4.4 and present a performance model to analyze their performance.

Radix sort is a non-comparison based sort algorithm which was used as far back as Hollerith's tabulating machines in the early 1900's. In radix sort, the values to be sorted are broken up into digits, and are sorted a digit at a time. Radix sorts can proceed either from least to most significant or vice-versa, We consider the variation of radix sort starting from least to most significant digits. A similar analysis also applies to the other variant.

In order to sort the keys according to a given digit, a counting sort [10] is used: a count (or histogram) of the number of keys with each possible digit value (or *radix*) is obtained. Then for each key, we compute the number of keys that have smaller radixes than the current radix (sum of all preceding histogram values), and add the number of keys with the same radix that are before this key in the sequence. This gives the output location of the key; and finally, the key is written to that location.

We now describe efficient parallel radix sort implementations. We use the following symbols in Sections 4 and 5.

$N$ - number of elements to be sorted
$K$ - data size of key in bits
$T$ - number of threads
$C$ - size of local storage – cache (CPUs) / shared memory (GPUs)
$S$ - SIMD width – number of 32-bit keys stored in a SIMD register

The following symbols are specific to radix sort:
$D$ - radix size (number of bits/digit)
$B$ - buffer size (in number of elements) per each of the $2^D$ radixes
$H$ - histogram of radixes, $H(k)$ is count of elements in radix $k$
$M$ - the number of blocks into which data is grouped
$H_m$ - a local histogram for block $m$

[1] In this paper, we use the term "gather/scatter" to represent read-/write from/to non-contiguous memory locations.

## 4.1 Basic Parallel Algorithm

Thread-level parallelism is easy to extract - the input to be sorted is broken up into blocks, and a local histogram obtained for each block. These local histograms are then combined into a global histogram, and the rearrangement is then done using the global histogram. The process of converting the local to a global histogram is a prefix sum (scan) operation that is easily parallelized [6, 27].

For all digits in the number:

**Step 1:** Divide the input evenly among $T$ threads. For each thread, compute a local histogram $H_t$ for each thread $t$.

**Step 2:** Compute a global histogram from the local histograms using a parallel prefix sum (scan) operation. We now have the starting offset to which each radix for each thread should be written.

**Step 3:** Each thread then computes the write offset of each key in its partition as the sum of: (1) the starting offset computed in Step 2 for the current radix, and (2) a local histogram update within the partition to compute the number of keys with the current radix that are before the current key. The key is then scattered to the correct position.

## 4.2 Architecture-Friendly Implementation Alternatives

The main bottleneck to efficient radix sort implementation is irregular memory accesses, both in computing histograms and in the rearrangement step. In particular, Step 3 involves rearranging a large array (typically residing in main memory) in a bandwidth-unfriendly manner, since consecutive writes may be to widely spread out regions of memory. Further, such scatters also result in increased latency of memory accesses due to cache misses and page misses in the TLB, since each access will potentially be to a different page of memory.

We can reduce memory bandwidth utilization as well as latency of memory accesses by improving the locality of the scatters. By utilizing **local storage** (caches in CPU architectures or shared memory on GPUs), we can transform global memory scatters to local storage scatters. This can be achieved in two different ways: (1) buffering up writes to different cache line wide regions (64-bytes) of main memory in local stores (Section 4.2.1) and writing them in a contiguous manner to main memory, or (2) locally sorting small blocks of data that fit into local memory (Section 4.2.2) according to the considered radix.

### *4.2.1 Buffer based scheme*

The basic idea in the buffer based scheme is to collect elements belonging to the same radix into buffers in local storage, and write out the buffers from local storage to global memory only when enough elements have accumulated. This has two advantages: (1) all writes to global memory are now at a larger granularity than individual elements, resulting in better bandwidth utilization, and (2) at any time, there are only as many memory pages being written to as there are radixes. Since all writes to each radix are to consecutive locations, each page of memory, once opened to write a key with a particular radix continues to be reused until either the page is full or all values belonging to that radix are written out. This scheme results in few page misses as long as the number of radixes are tuned to suit the size of the TLB in the architecture.

On cache-based architectures, a straightforward implementation of radix sort can perform buffering implicitly, since caches are arranged into cache lines that map to memory bus width, and are written out only when a write to a different region of memory accesses the same cache line. However, in our experiments, we found that a naive implementation resulted in a significant number of **cache conflict misses**. This phenomenon is observed for a wide variety of input distributions. Such misses occur because the buffers belonging to different radixes are not present in contiguous memory locations, and hence can map to the same cache line depending on the exact memory addresses. A fully associative cache, (where there is no restriction on where individual memory addresses are mapped) can eliminate conflict misses. However, fully associative caches are architecturally too expensive to be built. To eliminate cache conflicts, we propose and use a *software managed buffer* and allocate a contiguous region of memory where all buffers are present and manage it using software.

**Software managed buffer:** Our software managed buffer scheme is illustrated in Figure 1. We maintain an area of $B \cdot 2^D$ elements in local storage, in which $B$ elements are buffered for each radix. The input is streamed through and appended to the correct buffer depending on its radix (Figure 1(a)). Software then checks if the buffer has filled up. Once a buffer is full, it is emptied by writing out the buffered elements to global memory, as in Figure 1(b). The buffer can then be reused for further entries (Figure 1(c)). Our software buffer has $B$ is sized to be a multiple of 64 bytes, which is the unit of main memory transfers.

The choice of the number of bits considered in a single pass depends on the size of the local store, and the number of page translation entries (TLB size). For optimum utilization of memory bandwidth, there must be at least 64 bytes (memory transfer size) per radix in the local buffer, placing an upper limit on the number of radixes. Further, the number of radixes must also be less than the number of TLB entries, so that accesses to buffers do not suffer from TLB misses. Thus, the overall number of radix values is chosen from the minimum of these two constraints. Since the entire buffer fits in cache and is consecutively stored in memory, there are **no conflict misses**.

The scheme outlined above is efficient in terms of memory transfers and scalar instructions. However, it cannot utilize the data-level parallelism available in modern architectures. This is because the buffer scheme still involves scatters - to the local buffer rather than global memory. In the absence of gather/scatter[2] support, data-level parallelism requires that memory reads and writes be to consecutive memory locations. On GPU architectures, scatters to local (shared) memory are cheaper because they have multiple banks. However, even if the process of scattering is cheap, the buffer management (determining when a particular radix needs to be written out etc) can overwhelm any SIMD gains.

### *4.2.2 Local sort based scheme*

We now describe a scheme based on Blelloch [6] which is a SIMD-friendly approach to radix sort. This scheme blocks the input data into local storage (caches/shared memory) such that multiple operations can be performed on them before a global write.

**Motivation:** While the buffer-based scheme can be made efficient in terms of bandwidth use and cache misses, it cannot utilize SIMD efficiently since it requires atomic histogram updates as well as scatters to buffer entries in different cache lines. A key insight by Blelloch [6] was that SIMD can be much better utilized if the sorts are done a single bit at a time.

**SIMD-friendly sort:** Sorting on a single bit can be viewed as *splitting* up an input stream into two output streams - one for the '0' case and one for the '1' case. In this case, the write offsets of each input key can be computed in two steps: (1) compute a prefix sum (or scan) of the *negated* bit for each key. This gives the

[2] We use the term "gather/scatter" to represent read/write from/to non-contiguous memory locations

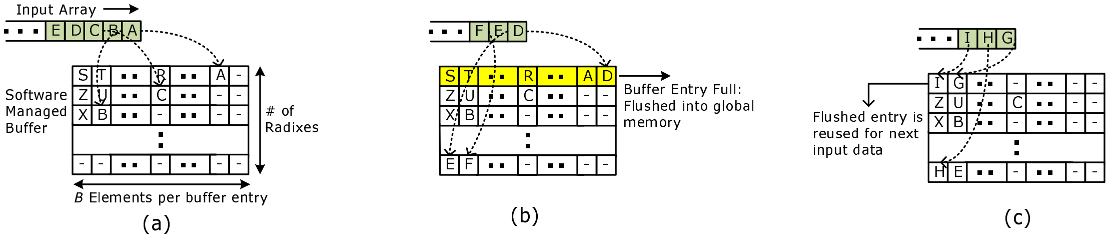

Input Array
Software Managed Buffer
# of Radixes
B Elements per buffer entry
(a)
Buffer Entry Full: Flushed into global memory
(b)
Flushed entry is reused for next input data
(c)

**Figure 1:** **Different stages of buffer-based radix sort (a) local scatter to different cache lines (b) a full cache line that is flushed into global memory and (c) reuse of the cache line after flushing.**

offsets for all keys with a bit value of '0'. (2) for keys with a bit value '1', we compute the difference between the total number of keys with bit '0' and the prefix sum value for the key as the offset. These two steps can be efficiently done with SIMD operations: (1) using SIMD versions of prefix sums, and (2) as a SIMD subtraction operation. SIMD-friendly prefix sums can be implemented using tree scans with two phases - a down-propagation phase performing a reduction of data and an up-propagation phase to propagate the reduced results [6, 22, 24]. The offsets found through the scan are then used to scatter the data. However, such a scatter is fast, because it only involves accesses to at most two cache lines - one for '0' and one for '1'.

However, sorting one bit at a time is highly inefficient in terms of memory bandwidth if we perform one pass over the entire data per bit. Therefore an efficient implementation of this scheme necessarily involves **blocking the input data** into chunks that fit in **local storage** (cache/shared memory), sort on multiple bits (one at a time), and then write out the partially sorted data to global memory. For instance, if a 32-bit value is divided into 4 digits of 8-bits each, then there will be 8 splits in the local sort. At the end of these 8 splits, the local data is sorted on the basis of the 8-bit digit. We then follow Steps 1 and 2 of Section 4.1 on the sorted data to obtain a global histogram of the data on the basis of the 8-bit digit, followed by a write to the computed global offsets in Step 3. Note that unlike the buffer scheme, the local sort scheme does not stream through the input data in Step 1. Instead, it has to block the input data so that it can hold it for multiple operations on different bits. For a more elaborate treatment of SIMD-friendly radix sort, please refer to Satish et al. [22].

The main disadvantage of this scheme is that we must perform as many local splits as the key width in bits. This means that we will have to perform many more operations than in the buffer version. The use of SIMD will result in a win only when the SIMD width of given architecture is sufficiently high to overcome the higher number of operations. We now go into details for a CPU and a GPU architecture.

## 4.3 Implementation on CPUs

We use the quad-core Core i7 CPU with each core having 4-wide SSE (SIMD) in our experiments. Each core has two SMT threads to hide latency. The two threads on a core share on-die L1 and L2 caches of 32 KB and 256 KB respectively. In our experiments, SMT had a impact of about 20% compared to running one thread per core.

Of the two radix sort implementation alternatives, split radix sort does not perform well due to the low SIMD width. The large number of instructions required for extracting bits and shuffling elements in each iteration overcomes any benefits due to the use of SIMD. The most efficient radix implementation on CPUs is consequently is the buffer implementation of Section 4.2.1. For the buffer based radix sort, neither the histogram update nor the buffer scatter can be written in a SIMD friendly fashion (due to the absence of gather/scatter SSE instructions), Radix sort is consequently implemented in scalar code without SSE instructions. We present the details below.

### *4.3.1 Detailed Implementation*

Radix sort is carried out in $P = K/D$ passes. Each pass involves the following steps:

**Step 1:** First, we divide the data evenly into blocks of size $M = N/T$. In parallel, each thread iterates through its block $m$ and computes a local histogram $H_m(d)$ for each bin $d$ by sequentially incrementing the appropriate bins for each input.

**Step 2:** We update $H_m(d)$ to the starting write offset for each bin $d$ of each block $m$, computed as $\sum_{m' \in M, d' < d} H_{m'}(d') + \sum_{m' < m, d' = d} H_{m'}(d')$.

**Step 3:** For each thread in parallel, we iterate through the input elements. For each element $n$, we compute its radix digit $d$. We then read off the updated $H_m(d)$ computed in Step 2 as the write offset $W(n)$ of element $n$. $H_m(d)$ is then incremented. Element $n$ is written to the buffer at position $(d \cdot B + W(n)\%B)$. If $((W(n)\%B) == (B\text{-}1))$, then we have filled up the buffer for bin $d$, and we write out the buffer elements $d \cdot B$ to $d \cdot B + B\text{-}1$ to global memory at positions $W(n) - (B\text{-}1)$ to $W(n)$ in a coherent manner.

**Choice of $B$ and $D$:** We choose $B$ such that $B \cdot K$ is a multiple of 512 bits (64 bytes), the data bus size of main memory. The exact choice of $B$ and $D$ depend on (1) the cache size of the architecture, and (2) the number of TLB entries. The Core i7 CPU has a two-level TLB, with a first level fully-associative TLB size of 64 entries and a second level 4-way associative TLB of 512 entries. We found that using $D = 8$ resulted in few TLB misses, while $D > 8$ resulted in increasing TLB misses. To avoid expensive L2 cache misses, both the buffer and histograms must fit in the 128 KB on-die L2 cache per thread. The buffer is of size $B \cdot 2^D \cdot (K/8)$ bytes, and the histogram is of size $4 \cdot 2^D$ bytes. $B \cdot 2^D \cdot (K/8) + 4 \cdot 2^D \leq 128K$. For 32-bit keys, $K = 32$, and with $2^D = 256$, this yields $B \leq 128$. Since our scheme is memory bandwidth efficient as long as $B \geq 16$, we picked $B = 16$, the memory bus width size.

### *4.3.2 Analysis*

Radix sort, for the CPU platform we analyzed, is done in $P = K/D$ passes = 4 passes for a 32-bit key. Both the compute and bandwidth requirements remain the same for all passes, and so we analyze a single pass. The operations involved in Step 1 are computing the radix of each element, and updating the corresponding histogram(reading, adding one and writing the updated histogram value). Let $O_{rad}$ refer to the cost of computing the radix (this takes 1 op[3] per element) and $O_{hist}$ refer to the cost of updating the histogram (this takes 3 ops per element). The compute cost of Step 1 is then $O_{S1} = O_{rad} + O_{hist} = 4 \cdot N$. In terms of bandwidth, the entire data has to be read once, and local histograms written out per thread. The local histograms are negligible in size compared to the input data. For $D = 32$ bits (or 4 byte data), this takes $4/BW$ cycles, where $BW$ is the bandwidth in bytes/cycle. For the Core i7, Step 1 is usually bound by memory bandwidth.

[3] 1 op implies 1 operation or 1 executed instruction.

Step 2 reads each histogram entry, updates it and writes it back. The cost for each such update is similar to $O_{hist}$, but only one update is required per histogram entry. Moreover, each histogram entry is read and written once. There are a total of $2^D \cdot T$ local histogram entries, which is negligible for $N >> 2^D \cdot T$. Step 2 takes very little time.

Step 3 computes the radix of each element, updates the histogram, writes each element to its buffer position (computes buffer position and writes it there), and has a buffer management cost associated with checking whether each buffer is full and if so, writing it back to global memory. The total cost of Step 3 is $O_{S3} = O_{rad} + O_{hist} + O_{write} + O_{buf}$. Here, $O_{write}$ is the cost of computing the buffer index and writing to the buffer, which is 4 ops/element, and $O_{buf}$ is the cost of detecting whether the buffer written to is full, reading all the elements and writing them to global memory. This also takes 4 ops/element. Overall, $O_{S3} = 12 \cdot N$ operations. Step 3 reads the input (as well as the small global histogram) and writes out the sorted results 64-bytes at a time in a bandwidth-friendly manner. However, in a cache-based architecture, any write of data must read the data into the cache and subsequently must be flushed out to memory later. Consequently, there are a total of 2 reads and 1 write of the input data in Step 3, leading to a total of 12 bytes of memory traffic per element for 4-byte data.

The overall compute cost for Steps 1, 2 and 3 is $O_{comp} = O_{S1} + O_{S2} + O_{S3} \sim 17 \cdot N$ ops. The overall algorithm involves $P$ passes, (where $P = 4$ for 32-bit key data), hence a total of $17 \cdot N \cdot 4 = 68 \cdot N$ ops. The total bandwidth required would be 16 bytes/element per pass, for a total of 64 bytes for 4 passes. This results in a required 64/68 bytes/cycle (assuming IPC=1). Current CPUs provide much higher bandwidth, and hence radix sort is compute bound.

## 4.4 Implementation on GPUs

The NVIDIA GPU architecture consists of multiple cores (called shared multiprocessors, or SMs). The GTX 280 has 30 such SMs. GPUs hide memory latency through multi-threading. Each GPU SM is capable of having more multiple threads of execution (up to 32 on the GTX 280) simultaneously active. Each such thread is called a **thread block** in CUDA.

Each GPU core has multiple scalar processors that execute the same instruction in parallel. In this work, we view them as SIMD lanes. The GTX 280 has 8 scalar processors per SM, and hence an 8-wide SIMD. However, the logical SIMD width of the architecture is 32. Each GPU instruction works on 32 data elements (called a *thread warp*), which are executed in 4 cycles. As a consequence, scalar code is 32X off in performance from the peak compute flops. For radix sort, this means that the scalar buffer version of radix sort performs badly. Consequently, the best version of radix sort is the split-based local sort that can use SIMD. We describe the details of this scheme next.

### 4.4.1 Detailed Implementation

In this section, we describe our GPU radix implementation based on the CUDPP implementation [1] (development branch, release 1.1) based on the description in Satish et al. [22].

We describe the GPU algorithm using the same notations as the CPU code (described in Section 4). Radix sort is carried out in $P = K/D$ passes.Each pass involves the following steps:

**Step 1:** First, we divide the data evenly into a set of blocks $M$, each of size $C/T$. Assign each block to a GPU *thread block*.

**Step 1a:** Each thread block locally sorts the data on $D$ bits by using $D$ 1-bit stream splits operation. 1-bit splits are implemented using scan primitives on the GPU [22].

**Step 1b:** Each thread block computes a histogram of $2^D$ bins on the basis of the bits used in the local sort. Each element of the sorted data is checked for equality against its predecessor, and markers are set when the equality check fails. These markers give the starting addresses of different histogram bins. The histogram is computed as the difference between consecutive starting addresses. Both the histogram and the sorted list are written to global memory.

**Step 2:** We compute $H_m(d)$, the starting write offset for bin $d$ of block $m$, as $\sum_{m' \in M, d' < d} H_{m'}(d') + \sum_{m' < m, d' = d} H_{m'}(d')$. This step is a global prefix sum on the $2^D$ histogram entries of each of the $M$ blocks.

**Step 3:** Each thread block $m$ writes the values corresponding to each of bin $d$ into its location $H_m(d)$. These writes involve coalesced memory accesses since each value belonging to the same bin is written to consecutive memory locations.

On the GTX 280, $C$ = 16KB, and the best implementation chose $T$ = 8, which means each block was of size 2 KB (storing 512 32-bit keys). The choice of $D$ was picked to be 4, since that resulted in most bins having more than 16 elements, resulting in efficient coalesced writes in Step 3.

### 4.4.2 Analysis

Overall, GPU sorts on 32-bit keys are compute bound. Step 1a involves a local $D$-bit sort of data using $D$ 1-bit stream splits. Each split operation involves 28 SIMD operations (4 in computing the bit and writing it to shared memory, 12 to perform a SIMD scan of the bits, 4 to compute the position of each element in the split array based on this scan, and 8 in writing each element to its correct position and writing back to global memory). Thus Step 1a takes $28 \cdot D$ ops. Step 1b computes the histogram based on the sorted array, and involves reading each sorted value (3 ops), checking it for equality with the previous value (7 ops), storing all indexes where the values differ to obtain histogram offsets (4 ops) and finding differences between the offsets stored to get a histogram count (8 ops), a total of 22 ops. Step 1 thus takes a total of $22 + 28 \cdot D$ ops. This produces 32 elements, hence $(22 + 28 \cdot D)/32$ ops/element. In terms of bandwidth, Step 1 reads and writes the data once for a total of 8 bytes (histograms are negligible in size). Since GPUs offer a high achievable bandwidth of about 2.8 bytes/cycle/core, reading and writing data only takes about 2.9 cycles/element. For $D = 4$, Step 1 is bound by compute operations and not by bandwidth.

Step 2 takes negligible time for large $N$ and is ignored. Step 3 requires about 16 SIMD operations, spent in reading the histograms and input keys (4 ops), computing the radix (4 ops), computing the global offset from the histogram result of Step 2 (6 ops), and storing the keys (2 ops). This produces 32 elements; hence a total of 0.5 ops/element. However, Step 3 needs to read and write the data once more: a bandwidth use of 8 bytes/element, which requires 2.9 cycles on the GTX 280. Step 3 is bandwidth bound.

## 5. MERGESORT

Mergesort is a comparison based algorithm that is based on merging two sorted lists of size $L$ into a sorted list of size $2L$. In the next step, it merges two lists of size $2L$ to obtain a list of size $4L$, and so on, until there is only one sorted list. This requires $O(\log N)$ steps, where $N$ is the size of the list to be sorted. Each step requires $O(N)$ work to perform the merge, hence resulting in an overall $O(N \log N)$ complexity.

Merge sort can be mapped well to modern many-core processors. In particular, apart from TLP, it is also capable of utilizing DLP and utilizes bandwidth efficiently. While the performance of the scalar version of merge sort suffers from branch mispredictions, the data-parallel variant uses merging networks in which branches are infrequent. Consequently the impact of branch misprediction is

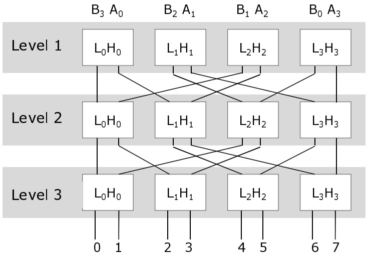


**Figure 2:** **A 4x4 bitonic merge network.**

low. Further, SMT has a low impact of less than 10%. Although an efficient implementation on CPUs is known, it has not been adapted to GPUs. We now describe the high level merge sort algorithm and how it is mapped to many-core processors and GPUs.

## 5.1 Parallel MergeSort Algorithm

We adopt the basic bitonic merge network algorithm by Chhugani et al. [9]. The algorithm works by successively merging lists at SIMD width granularity to exploit data-level parallelism. To exploit thread-level parallelism, the set of lists to be pairwise merged is evenly split among the threads. The overall algorithm is to merge successive pairs of lists using two steps:

**Step 1:** As long as each thread has one or more pairs of lists to merge, threads work independently to merge their allotted pairs. Each thread merges a pair of lists using a data parallel bitonic merge network. A bitonic merge network can be implemented as a sequence of comparison and shuffle operations. as shown in Figure 2. At the end of each step, the size of each sorted list doubles, and the number of lists to be merged halves.

**Step 2:** Eventually, there will be fewer lists to merge than threads. Multiple threads must now cooperate to merge two lists. This can be done by partitioning each pair of lists into multiple chunks that can be merged independently. Francis et al. [11] proposed a scheme based on finding quantiles to partition the lists evenly. Once partitioning is done, these partitioned lists can be independently merged using Step 1. Continue Step 2 until the entire list is sorted.

**Multi-way merge:** Merge sort can be made bandwidth-oblivious by adopting multi-way merge [11]. Instead of merging two lists at a time, this technique merges *all* the sorted lists from Step 1 to create the final sorted list. It is implemented in a bandwidth friendly fashion by using a binary tree, where the leaves correspond to the cache-sized sorted lists, and the internal nodes correspond to the partial results of merging its children nodes. Only a fixed sized chunk of elements is brought into the cache from the leaves, and the output is finally written to the main memory (as the final sorted list). The intermediate nodes are always cache resident and do not contribute to the bandwidth requirement.

For particular architectures, specific optimizations and parameter choices (such as the width of the bitonic merge kernel) may be required. We discuss these details next.

## 5.2 Implementation on CPUs

We briefly discuss the mergesort implementation described in Chhugani et al. [9] for multi-core CPUs with $S = 4$-wide SSE.

### 5.2.1 Detailed Implementation

**Step 1** is implemented in two phases – a first phase where each thread sorts data up to its cache size (requiring a single read and a final write of data from/to main memory), and a second phase where the data sorted by each thread is merged (using multi-way merge, this also only requires only a single read and write of data from main memory). Both phases use a 4-way bitonic merge network using $S = 4$-wide SSE instructions. We choose to view the bitonic merge kernel as a min/max/interleave network as shown in Figure 2. The inputs to the merge network are *two* 32-bit sorted lists of length four each, and the output is a merged array of length eight. Merging arrays of length greater than four is done by a sequence of calls to this network. After each call, the smaller four elements are written back to the output, and the rest are retained. Four elements are then picked from the input array with the smallest remaining element. These are then merged with the retained elements. Merging two lists of total size $L$ can be accomplished in $(L-1)/S$ calls to the bitonic merge network, where $S$ is the size of the merging network (same as SIMD width). The bitonic merge network itself assumes that one of the inputs is sorted in decreasing order - so the four values read are reversed using shuffles prior to each call. A code snippet for a single level (innermost loop) of the 4-way bitonic network is shown in Figure 3.

```
// The following code is run three times for a 4x4 network

L_1 = sse_min(A,B);    // A and B are the two input SSE registers
H_1 = sse_max(A,B);
L_1p = sse_shuffle(L_1, H_1, imm2);
H_1p = sse_shuffle(L_1, H_1, imm3);
```

**Figure 3:** **SSE code snippet for one level of the bitonic merge network. The overall network requires 6 SSE min/max and 7 shuffle instructions.**

In a straightforward implementation, the latencies of the SSE instructions will be exposed due to dependencies between the min/max and shuffle instructions. On out-of-order CPUs, this can be avoided by merging multiple lists in parallel.

**Step 2** is implemented by generating independent work per thread by partitioning a large list. For instance, to merge two large lists using 2 threads, the median element in the merged list is found. While finding the median, the starting location for the second thread in both lists can also be found. Similarly, to merge two lists using 4 threads, the 1/4th, 2/4th and 3/4th quantiles of the merged list can be found, and the starting elements for the threads also computed. These partitioned lists are then independently merged using Step 1.

### 5.2.2 Analysis

**Bandwidth Considerations:** It is important to avoid becoming bound by memory bandwidth since this can dominate the compute requirements of the merge network. If each of the $\log N$ passes of merge sort needs to read and write the entire list of 4-byte keys from main memory, it would require 8 bytes of memory traffic (per level) per element to be merged. The resultant bandwidth requirements would exceed available bandwidth on Core i7.

However, merge sort can avoid this by (1) performing the first few iterations in cache (when the sizes of the arrays to be merged is small). These passes only read and write data to cache, with a single read and write of data from/to main memory. (2) Beyond this point, the lists to be merged are too large to reside in cache. However, using the multi-way merge scheme, data only needs to be read and written once more from main memory. Using these two optimizations, the entire merge sort algorithm can be implemented with just two reads and two writes of data to memory, and is far from bandwidth bound.

**Compute Requirements:** We build up a model for the time taken by merge sort based on the time taken for a bitonic merge step. For 4-wide SSE ($S$ = 4), the bitonic merge takes 6 min/max and 7 shuffle ops, a total of 13 ops. Merging $N$ elements in a single call to Step 1 requires $N/S$ calls to the merge network, for a total of $13 \cdot N/S$ ops. Let Step 1 be called $P_1$ times.

Each of the $\log N - P_1$ remaining iterations call Step 2. Each call to Step 2 can, in addition to the $N/S$ calls to the bitonic merge network as in Step 1, also needs to partition the data. If the number of operations to partition is $O_{part}$, then the total cost per call is $13 \cdot N/S + O_{part}$ ops. In addition, for each step, there is an overhead of a executing a barrier call $O_{sync}$ due to the fact that all threads of merge sort must synchronize between iterations.

The overall time for merge sort is therefore $(13 \cdot N/S + O_{sync}) \cdot P_1 + (13 \cdot N/S + O_{sync} + O_{part}) \cdot (\log N - P_1)$. The partition time $O_{part}$ is generally small enough to be neglected, hence the above expression simplifies to $(13 \cdot N/S + O_{sync}) \cdot (\log N)$. $O_{sync}$ may be non-negligible can be a source of overhead.

To summarize, merge sort on CPUs is SIMD-friendly and bandwidth-oblivious.

## 5.3 Implementation on GPUs

Previous work on merge sort on GPUs has used parallel binary search to perform the merge operations at each merge step [22]. We propose to instead use the efficient bitonic merge scheme that can efficiently exploit thread-level and data-level parallelism. Due to the wider SIMD width of GPUs, we adopt a wider merge network. We highlight the main features of the algorithm and the differences from the CPU algorithm below. The resulting implementation is the *fastest* comparison based GPU sort.

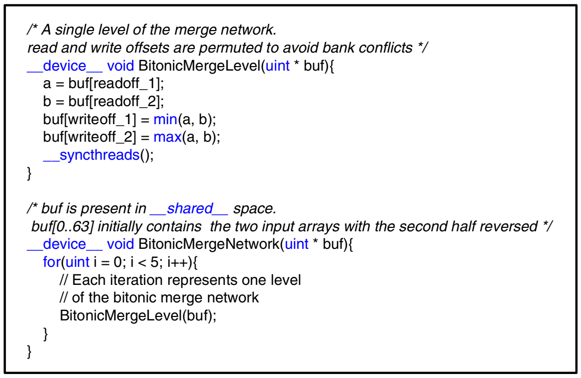
### 5.3.1 Detailed Implementation

**Size of the merge network:** GPUs have 8 execution units per processor (hence 8-wide SIMD), but a logical SIMD width of 32. All instructions execute on 32 elements and take 4 cycles. To avoid wasting compute resources, we must use all 32 logical SIMD lanes.

For merge sort, the number of operations per element sorted by the bitonic merge network increases as the network width $S$ increases. It is therefore better to use as small a merge network as possible. Instead of implementing a 32-wide network, it is better to instead use the 32 lanes to implement $U$ parallel $32/U$-wide networks (each sorting independent lists). We would then use a condition on the *threadId* (SIMD laneid) to decide which list each SIMD lane will process. There are two main constraints: (1) the physical SIMD width of 8 provides an upper bound of 4 on $U$, since each of the 8 physical lanes must run the same instruction. (2) sorting independent lists will require the inputs to be gathered into SIMD registers and outputs to be scattered back to global memory. However, since memory gathers and scatters only occur at a half-warp granularity of 16 threads, using $U = 2$ avoids any gathers/scatters.

**Step 1** is therefore performed using a 16-wide sorting network. The sorting network, as before, can be implemented as a min/max/interleave sequence of instructions. However, the key difference comes due to the **absence of a single-instruction shuffle operation** on GPUs. Instead, shuffles need to be implemented by scattering elements to shared memory (to their correct output positions), perform a flush to shared memory to ensure values have been written, and reading them back consecutively from shared memory.

The performance of the shared memory scatter operation depends on the pattern of access. Shared memory on recent GPUs is arranged into 16 banks, and accesses that hit different banks (among the 16-threads of a half-warp) can occur simultaneously with no gather overhead. However, accesses that hit the same bank are serialized. Since a majority of the time in merge sort is spent in the merge network, this can result in a significant performance overhead. In the merge network of Figure 2, the output of each min instruction $L_i$ goes to index $2i$, and the output of each max instruction $H_i$ goes to index $2i+1$. Such accesses are known to create two-way bank conflicts on GPUs [20], leading to a substantial loss of performance. We can avoid bank conflicts by **permuting** the lists to be merged. The permutation index is chosen such that the reads and writes to shared memory are to different banks. This technique is applicable to GPUs of any SIMD width. A code snippet of the merge network in CUDA is shown in Figure 4.



```
/* A single level of the merge network.
read and write offsets are permuted to avoid bank conflicts */
__device__ void BitonicMergeLevel(uint * buf){
   a = buf[readoff_1];
   b = buf[readoff_2];
   buf[writeoff_1] = min(a, b);
   buf[writeoff_2] = max(a, b);
   __syncthreads();
}

/* buf is present in __shared__ space.
 buf[0..63] initially contains  the two input arrays with the second half reversed */
__device__ void BitonicMergeNetwork(uint * buf){
   for(uint i = 0; i < 5; i++){
      // Each iteration represents one level
      // of the bitonic merge network
      BitonicMergeLevel(buf);
   }
}
```

**Figure 4: CUDA code snippet for bitonic merge.**

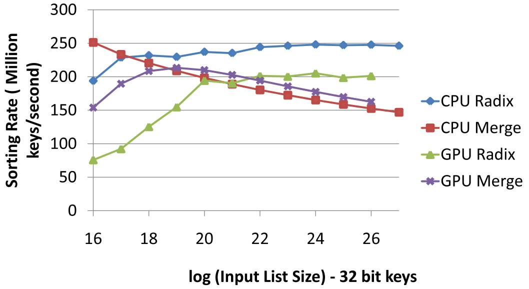


**Figure 5: Sorting performance of radix and merge sorts on Intel Core i7 and NVIDIA GTX 280.**

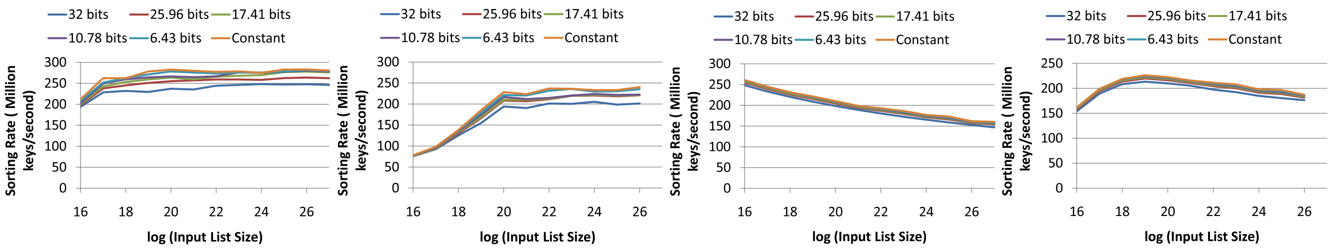
As in the CPU case, we found it necessary to merge multiple lists in parallel to avoid instruction dependency overheads. We found that we needed to merge 4 lists in parallel for optimal performance.

**Step 2** is performed as in the CPU, by finding the quantiles of the arrays to be merged and the starting offsets of each block.

### 5.3.2 Analysis

**Bandwidth Considerations:** Unlike on the CPU platform, there is no need to implement caching or multi-way merge on current GPUs. There are two reasons for this: (1) the number of compute instructions is higher on the GPU due to the overhead of the shuffle operation, and (2) the bandwidth-to-compute ratio of the GPU platforms is higher than CPUs. Even a simple implementation is about 4X away from bandwidth bound; this means that the bandwidth could decrease by a factor of 4 without impact in performance. However, future GPU platforms may see lower bandwidth-to-compute ratios; if this ratio decreases by more than 4X, merge sort may become bandwidth bound and require multi-way merge.

**Compute Requirements:** The bitonic merging network takes 8 ops/level (2 loads from shared memory, a min, max, 2 writes to shared memory, and 2 for synchronization). A $S$-wide merging network has $\log 2S$ levels. A 16-wide network thus has 5 levels, hence a total of 40 ops. This produces 32 results to be written into the output buffer; hence it takes 40/32 = 1.25 ops/element for the bitonic merge. Additionally, each bitonic merge requires reading global to shared memory and a reversal of one array ($O_{overhead}$). Merge sort requires a total of $\log N$ such passes. For the last $\log T$ levels when there are fewer lists to merge than the number of thread blocks $T$, there is also a cost of partitioning large arrays (as per Step 2)($O_{part}$). On the GPU, $T = 240$, with 8 thread blocks/core and 30 cores - which means that the last 8 levels need partitioning. The overall time for merge sort, $O_{overall} = (\log N) \cdot (1.25 + O_{overhead}) \cdot N + 8 \cdot O_{part}$.



(a) (b) (c) (d)

**Figure 6: Performance Variation of different sorts with varying entropy (a) CPU Radix (b) GPU Radix (c) CPU Merge and (d) GPU Merge.**

# 6. PERFORMANCE EVALUATION

**Machine Configuration**: We now evaluate the performance of radix and merge sort on an Intel quad-core 3.2 GHz Core i7 CPU and an NVIDIA GTX 280 GPU running at 1.3 GHz. Both the CPU and GPU systems have 4 GB RAM and run Linux. Peak flops and peak bandwidth of the two platforms are shown in Table 1.

## 6.1 Comparative Analysis on CPUs and GPUs

We first present our sorting performance results on a random distribution of 32-bit integers. Figure 5 presents our results of (a) the best radix sort implementation and (b) the best merge sort on Core i7 and the GTX 280. The best radix sort implementation on Core i7 is our scalar buffer version while the best one on the GTX 280 is the stream split version. We present the sorting throughput (in millions of elements/second) versus $\log_2 N$, where $N$ is the length of input to be sorted. $N$ varies from $2^{16}$ to the largest input that fits in memory on the respective platforms (our sorts are out-of-place; we must store the input, output and some temporaries in memory). The effect of the computational complexity of merge and radix sort can be seen from the figure. The sorting rate of CPU radix sort **is constant** at sizes above 128K elements, in line with the linear $O(N)$ complexity of radix sort. On the other hand, the sorting rate of merge sort **reduces with** $N$ from about 240 M elements/second at $N = 128$ K to 140 M elements/second at $N = 128$ M. This is the effect of the $O(N \log N)$ complexity of merge sort. GPU trends are also similar at large $N$ over one million. However, synchronization costs and high kernel call overheads on GPUs leads to slower performance of both radix and merge sort for $N$ smaller than 1 million.

**CPU performance:** On the Core i7, radix sort has a throughput of 240M elements/second and outperforms merge sort for most values of $N$ starting from 128K elements. At $N = 2^{27}$, the radix sort throughput is **1.7X** the merge sort throughput of 144M elements/second. In terms of cycles per element (cpe), a single pass of radix sort is written using scalar instructions and takes about 13 cpe, while a single pass of merge sort (using SSE instructions) only takes 2.5 cpe (which is a 3.8X speed-up over scalar code). Radix sort always takes 4 passes, and hence 52 cpe. Merge sort takes $\log N$ passes and hence $2.5 \log N$ cpe. Our radix sort performance is the best reported performance on CPU architectures to date.

**GPU performance:** On the GTX 280 GPU, merge and radix sorts have comparable performance. Radix sort is still slightly faster than merge at large $N$, but the gap narrows to only about **10%** at $N = 2^{26}$. Radix sort takes about 195 cycles/element, which is spent in performing 32 1-bit splits (each takes about 3.75 cycles per element), plus the time taken for histogram computations and writing out local sorted data to global memory. The throughput of the split code is about 200 M elements/second. A single pass of merge sort takes only about 1.4 instructions per element in the bitonic merge network (this is about half the instructions as on the Core i7 due to the SIMD width being twice as large). This would mean that for $N = 2^{26}$ elements, merge sort could potentially run at 268 M elements/second. However, we found that the merge code suffers from overheads due to index computations and array reversal, resulting in an achieved performance of only 176 M elements/second. On an architecture where these overheads are lower, merge sort can *perform better* than the split radix sort.

**Comparing CPUs and GPUs:** In terms of absolute performance of CPU versus GPU, we find that the best radix sort, the CPU radix sort outperforms the best GPU sort by about 20%. The primary reason is that scalar buffer code performs badly on the GPU. This necessitates a move to the split code that has many more instructions than the buffer code. This is enough to overcome the 3X higher compute flops available on the GPU[4]. On the other hand, the GPU merge sort does perform slightly better than the CPU merge sort, but the difference is still small. The difference is due to the absence of a single instruction scatter, and the additional overheads, such as index computations, affecting GPU performance.

**Different distributions:** We now show the effect of different input distributions on radix and merge performance on CPU and GPU architectures. We create distributions with 32-bit keys of different entropy based on the method proposed in Thearling et al. [26]. Figure 6 shows that sort performance is stable, and improves only slightly with decreasing entropy. As the key entropy drops from 32 bits to 0 (constant data), the radix sort performance improves by about 12% on the CPU and about 20% on the GPU. The improvement is because of the improved pattern of memory access in the rearrange step (Step 3) of the respective sorts. For constant data, all keys belong to the same histogram bin and will all be written to consecutive memory locations. This results in streaming write accesses that are very efficient on CPU and GPU architectures.

Memory access patterns in merge sort do not change when the entropy of the keys decreases. As we move from keys of 32-bit entropy to those with 0-bit entropy, merge sort performance improves very marginally by less than 5% on both CPUs and GPUs.

## 6.2 Comparison to analytical model

We analyze CPU code using the VTune Performance Analysis Tooland compare it to our performance model. For GPU code, we use the CUDA profiler [20] for comparative purposes.

**CPU Radix Sort:** The number of instructions (per element) in each step found from VTune matches closely with the number of operations predicted in Section 4.3.2. Step 1 takes 5 instructions/element/core, as predicted in our model. In terms of bandwidth, Step 1 reads 4 bytes/element. The measured bandwidth on the Core

[4] Of the 933 actual GFlops on the GTX 280, 622 are available through madd and Special Purpose Unit flops that sorting algorithms cannot readily use.

i7 processor is about 7.2 bytes/cycle[5]. Thus it takes 4/7.2 = 0.55 cycles/element. The actual run-time is 0.57 cycles/element, **which is very close**; hence Step 1 is bandwidth bound. Step 2 takes negligible time; matching our performance model.

Step 3 computes a histogram, as in Step 1, but also writes data to the buffer and performs buffer management to flush buffer lines that are full. $O_{write} = 4$ instructions (3 for computing indexes and 1 for writing the data), and $O_{buf} = 3$ instructions (compare for full buffer, branch and write). Hence, Step 3 takes a total of 12 instructions/element/core, or a total of 4 instructions/element on 4 cores. This is very close to the 4.3 instructions/element actually found. By a combination of loop unrolling and the use of SMT, we are able to hide latency. CPU architectures can process multiple instructions in parallel, and our code retires 1.6 instructions per cycle. Step 3 takes a total of 2.7 cycles/element. Overall, we therefore take 3.3 cycles/element per radix sort pass. The run-time of 4 passes of radix sort is 13.1 cycles/element. On 4 cores, we get near perfect scaling of 3.9, leading to a throughput of 240 M elements/second.

**CPU Merge Sort:** Each 4-wide bitonic merge step takes about 17 SSE instructions, about 30% more instructions than expected due to register spills. Overall, this produces 4 elements, hence a total of 17/4 = 4.25 instructions/element. By software pipelining [3], instruction latencies are hidden; an IPC of 1.7 is obtained, leading to a performance of 2.5 cycles/element/iteration on 1 core, which reasonably matches our model.

**GPU Radix Sort:** The number of SIMD instructions in each step closely matches expected numbers. Step 1 is indeed compute bound, with each pass taking about 4.8 instructions/element for 4-bit sort and histogram update, about 15% off the expected number of 4.2 instructions/element. Each SIMD instruction takes 4 cycles; hence Step 1 takes a total of 19.2 cycles/element per pass. Step 2 takes negligible time. Step 3, as expected is bandwidth bound; however, it takes 5.3 cycles/element, which is significantly higher than the expected bandwidth bound number of 2.9 cycles/element. We found the main reason to be that the local sort method suffers some uncoalesced accesses when the distribution of elements to different radixes within each block is not uniform. Each consecutive 64-byte region can have elements belonging to more than 1 radix, leading to scatters and higher bandwidth consumption. On average, the bandwidth consumption is about 1.75X the expected value. Accounting for this, our model closely matches the results.

**GPU Merge Sort:** Each 16-wide bitonic network takes 1.4 instructions/element, which is only ∼10% off the predicted 1.25 instructions/element. $O_{overhead}$, the time to compute indices for global memory addresses, read global to shared memory, and reversal of one of the arrays read, takes about 0.6 instructions/element. Overall, merge sort takes about 2 instructions/element. With perfect latency hiding, each instruction takes 4 cycles to execute, a total of 8 cycles. However, merge sort suffers a small latency impact (∼10%) due to shared memory synchronization costs, and takes a total of 9 cycles/element/iteration on 1 core, resulting in a performance of 180 M elements/second for $N = 2^{26}$.

## 6.3 Comparison to other sorts

We implemented the state-of-the-art algorithms on the latest CPU and GPU platforms. Our CPU radix sort and GPU merge sort implementations improved on existing best known non-comparison CPU sorts and comparison based GPU sorts respectively.

| | CPU | | | | GPU | | |
|---|---|---|---|---|---|---|---|
| **BW** | 30.0 | | | | 141.7 | | |
| **GFlops** | 103.0 | | | | 933.3 | | |
| | *Radix* | IPP [2] | Merge [9] | AA [13] | Radix [22] | *Merge* | Sample [18] |
| 256K | 1.1 | 1.2 | 1.2 | 2.1 | 2.1 | 1.3 | 2.5 |
| 1M | 4.4 | 5.2 | 5.3 | 9.6 | 5.4 | 5.0 | 9.1 |
| 4M | 17.2 | 25.4 | 23.3 | 40.9 | 20.8 | 21.6 | 38.1 |
| 16M | 67.6 | 160.7 | 101.5 | 185.7 | 81.7 | 94.5 | 139.8 |
| 64M | 271.0 | 550.5 | 439.7 | 835.5 | 333.4 | 381.8 | 524.3 |

**Table 1: Performance comparison of the best performing sorting algorithms across platforms with various peak bandwidth (GB/sec) and peak flops (GFlops). We italicize our sorting implementations. Running times are in milliseconds, so lower numbers are better.**

Table 1 compares the best reported running times of comparison and non-comparison sorts on the latest CPU and GPU platforms. On the CPU platform, a 3.2 GHz Core i7, we compare our radix sort implementation with the best known radix implementation, the Intel IPP radix sort [2] and state of the art comparison based sorts - our implementation of the merge sort by Chhugani et al [9], and AA-Sort [13]. All results were collected on the same platform. Our radix sort is up to 1.7X faster than the best reported sort so far (the merge sort based on Chhugani et al. [9]), and up to 3.8X better than AA-Sort. The main reason is that the radix sort algorithm has fewer operations than merge sort due to computational complexity, and the small SSE width of the CPU is insufficient to compensate for the higher complexity. We are also **2X** better than the IPP radix sort at larger data sizes. This is because IPP radix sort suffers from cache misses, in particular conflict misses, and becomes bound by main memory bandwidth and latency for large data sets.

On the GPU, an NVIDIA GTX 280, we compare our GPU merge implementation versus a recent comparison based sample sort [18] (the best reported GPU comparison sort). The performance results for sample sort is taken from [18]. We are about 1.5-2X better than their recent implementation. While their implementation is on a Tesla C1040 platform, this has the same compute flops as the GTX 280. Their results show that their implementation is compute bound, and performance is not expected to improve on the GTX 280. Our merge sort is the **fastest** reported comparison based sort on GPUs. We also compare our merge sort implementation to the radix sort implementation based on Satish et al. [22]. Our merge sort is faster than radix up to 2M elements since each pass of merge sort has fewer instructions than the split version of radix sort, but the complexity of radix sort takes over at larger inputs.

# 7. LARGE KEYS AND PAYLOADS

We have so far reported sorting performance on 4-byte keys. While 4-byte keys are useful in many database applications, larger keys are used in certain contexts, like sorting on names/dates [5, 7]. Databases also frequently need to sort entire records, in which case we need to sort keys with payloads.

**Handling variable length keys and payloads:** We advocate using fixed length keys and using record id's instead of entire records while sorting. This is to maximize the use of SIMD and to minimize bandwidth requirements while reordering data. In order to handle variable length data, we use the work of Bohannon et al [7] to remap variable length keys to fixed length keys of 4 or 8 byte keys. We investigate both these cases in this section. To handle payloads, we use fixed length rid's during the sort and perform a epilogue to rearrange entire records once sorting is done. Such a technique is also used in Kim et al. for join algorithms [16].

[5] The peak bandwidth is about 9 bytes/cycle of which 7.2 bytes/cycle is achieved in practice.

**Impact on Radix Sort:** Increasing the key length from 4-bytes to larger keys leads to increase in both compute and bandwidth resources during sort. For radix sort, as the key-width doubles, each pass will need to read and write twice the data that it did previously. Moreover, since radix sort considers a constant number of bits per pass, the number of passes also doubles as we double key-widths. The bandwidth requirement of radix sort therefore scales **quadratically** with key-width.

For the scalar CPU buffer radix sort, the original code to sort 32-bit keys does not use SSE registers. When we sort larger keys, we utilize these unused SSE registers to store keys up to 16 bytes, and run the scalar buffer code on data in SSE registers. The number of instructions per pass, is therefore constant; compute only increases linearly with the number of passes. For the GPU split code, we already utilize SIMD registers. Parts of the algorithm such as local scatters need to store the entire keys in SIMD; these portions scale linearly with key width per pass; and since the number of passes increases linearly with key width, the net effect is a *quadratic scaling*. There are however, other parts of the algorithm (histogram update and 1-bit scans) that only work with a few bits at a time and are unaffected by key width. These only *scale linearly*, and hence overall compute does not scale quadratically. Since compute scales slower than bandwidth requirements of radix sort with increasing key widths, radix sort gets bandwidth bound on large keys.

Figure 7 shows the time taken to run radix and merge sort with increasing key widths relative to the 32-bit key radix sort times on each architecture (lower bars indicate better performance). For radix sort, we see that CPU performance on 64-bit and 128-bit keys is 2.7X and 10.9X slower than 32-bit keys. CPUs become bandwidth bound for larger than 6 byte keys, and thereafter slow down quadratically with key width. On GPUs, the slowdown on 64-bit and 128-bit keys is 3.0X and 9.7X. Only parts of the GPU implementation are bandwidth bound; hence the performance drop is more than linear but less than quadratic in key width.

*Handling payload:* Figure 7 also shows that the performance of 32-bit key and rid pairs is only 1.3X worse than 32-bit key sort on CPUs and only 1.2X worse on GPUs. This is because the number of passes of radix is the same as for keys only; the only performance loss is due to the instructions and extra bandwidth required for moving the rids.

**Impact on Merge Sort:** The bandwidth requirement of merge sort only increases **linearly** with key width. Our implementation of merge sort is not bandwidth bound even on 128-bit keys on either platform. The compute requirements of merge sort on larger keys depends on: (1) the data types on which comparisons are naturally supported on each architecture. (2) the number of keys compared in a single SIMD instruction, which is decided by the expense of gathering 32-bit portions of large keys into SIMD registers (we could either use SIMD registers to store the entire large key or a number of 32-bit partial keys). (3) gain in performance by using smaller merge networks in case SIMD is used to store the entire key.
On CPUs, 64-bit SIMD comparisons are efficiently supported with only a 2X performance loss over 32-bit comparisons. However, there is no support for 128-bit compares, which must be implemented using multiple 64-bit comparisons using:

$$
\begin{aligned}
(a_{[127..0]} > b_{[127..0]}) \equiv (a_{[127..64]} > b_{[127..64]}) \vee \\
((a_{[127..64]} == b_{[127..64]}) \wedge (a_{[63..0]} > b_{[63..0]}))
\end{aligned}
$$

Since CPUs do not have efficient gather support, SSE registers are used to store the entire keys: 2 64-bit keys/32-bit (key,rid) pairs or a single 128-bit key (leading to a 2X or 4X drop in performance). Some of this drop is gained back by the use of smaller and more efficient sorting networks. The net effect is that performance drops by about 1.8X for key-value pairs and 64-bit keys, and a further sharp drop of 2.9X from 64-bit to 128-bit keys (shown in Figure 7). GPUs do not have support for 64-bit compares; however, gathering 32-bits out of a 64-bit or 128-bit key can be done through efficient gathers in shared memory. Thus we must construct large-key comparison functions using 32-bit compares, but all comparisons can work on a 16-wide network.After a 2.1X drop from 32-to 64-bit keys, there is only a further 1.7X drop from 64-bit to 128-bit keys. Key value pairs can be efficiently compared by extracting out only keys; this only leads to a 1.2X performance drop.

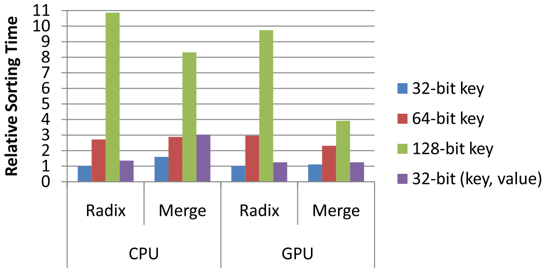


**Figure 7: Performance of radix and merge sort on large keys (relative to 32-bit radix performance) on Intel Core i7 and NVIDIA GTX 280.**

**Summary:** The overall trends for radix and merge sort scaling means that merge sort (which is 1.7X slower than radix sort on 32-bit keys) becomes comparable in performance to radix on 64-bit keys and 1.3X better on 128-bit keys. On GPUs, where merge performance on 32-bit is only 10% off radix, merge sort becomes 1.3X and 2.5X faster than radix for sorting 64-bit and 128-bit keys. The fundamental reason is the high bandwidth utilization of radix; making merge sort superior for sorting large keys on current hardware.

## 8. FUTURE ARCHITECTURES

In this section, we take into account the growing trends towards single-chip many-core architectures, with each core having wider SIMD to predict the best performing sort version. Future projections point to increasing core count as the major way of improving performance under power constraints. However, the bandwidth available is unlikely to scale with the increased core count; the **bandwidth per core** will drop in the future [21].

Recent trends have pointed to increasing SIMD width from 4-wide in SSE to 8-wide in AVX and current GPUs. Recently announced architectures by Intel (Larrabee) [23] and NVIDIA (Fermi) [19] have 16-wide SIMD units. The logical SIMD width of NVIDIA GPUs is already 32; we project this may increase to 64-wide and beyond. Further, in order for the increased compute power due to core and SIMD to be useful, applications must not be bandwidth bound. As in other applications, sorting algorithms that are SIMD-friendly and use less bandwidth should perform well in the future.

In order to project the performance of radix and merge sort on future architectures, we perform a simulation of our different sorts on a generic *future* many-core processing platform. We model a Chip Multi-Processor architecture with high core count: 64-cores and increasing SIMD widths from 16-wide to 128-wide. We enhance each core to have distinguishing architectural features: arbitrary gather/scatter, general interleave instructions and unaligned memory access support, while retaining typical characteristics of a many-core processor such as coherent caches and in-order instruction processing, as on recently announced parts by Sun [8] and Intel [23]. Although we expect the per core bandwidth to decrease, the extent of the decrease is unclear. We therefore maintained it **constant** in our experiments. This gives an **upper bound** on radix

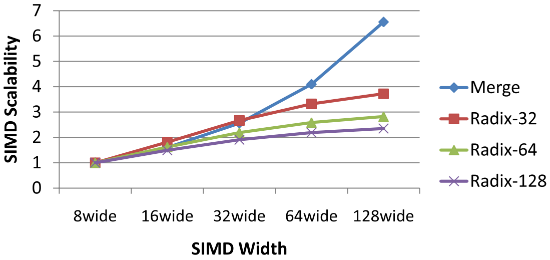


**Figure 8: Simulated performance of merge sort on large keys (relative to 32-bit keys) for future architectures with varying SIMD width. SIMD widths are shown on log scale.**

sort performance (which has high bandwidth requirements). Our per-core bandwidth assumption is the same as that on existing GPU architectures. Each core runs at similar frequency to current GPUs.

## 8.1 Effect of increasing SIMD

We investigate how radix and merge sort performance scales with SIMD width. We simulate the performance of both sorts on 32-bit, 64-bit and 128-bit keys. As SIMD widths increase, the best implementation of radix sort will rely on the SIMD-friendly split version and not the buffer version. Even at the current effective 8-wide SIMD on GPUs, we find that the best performing radix sort is the split based code, and therefore we consider only the SIMD scalability of split radix code and merge sort. Our simulations indicate that both algorithms scale similarly with number of cores; hence we do not include core scalability. Figure 8 shows the results of our simulations as SIMD widths increase from 8-wide to 128-wide[6]. We see that the performance of merge sort scales about 6.5X with an 16-fold increase in SIMD width. This is independent of key size and is the result of moving to wider merge networks and overheads that are not SIMD friendly. Radix sort, however, scales only by 3.7X from 8-wide to 128-wide SIMD at 32-bit keys. This is the result of portions of radix sort being bandwidth bound - and hence unable to utilize SIMD resources. The bandwidth effect becomes even larger for wider keys. Radix sort only scales 2.8X and 2.3X for 64-bit and 128-bit keys as SIMD increases from 8-wide to 128-wide. Furthermore, SIMD gains saturate as radix sort gets more bandwidth bound at large SIMD widths. The gain of moving from 64-wide to 128-wide SIMD is only 1.1X for 64-bit keys.

## 8.2 Effect of increasing memory size

As memory capacity keeps increasing, we can store databases with increasing number of tuples. As the number of tuples increase to beyond $2^{32}$, we expect that key sizes will also grow to 64-bits and beyond. It is a common belief that the computational complexity of merge sort will overwhelm any architectural advantages as the number of tuples grows large. However, our simulations indicate this is not a problem. Assuming a 64-wide SIMD architecture, merge sort is 2.2X faster than radix. Even at $N = 2^{40}$ (1 trillion records), merge sort will only slow down by 40/26 = 1.5X, which will still be 2.2/1.5 = **1.5X** faster than radix sort. At even larger key sizes of 128-bits, merge sort is **3X** faster at 1 trillion keys. Note that this is under our constant bandwidth per core assumption; the merge speed-ups will be *even higher* at lower bandwidth per core.

**Summary:** For 64-bit keys, radix sort only scales 2.8X from 8-wide to 128-wide SIMD and the scaling further drops to 2.3X for 128-bit keys. In contrast, SIMD-friendly merge sort is bandwidth oblivious and scales well (around 6.5X from 8-wide to 128-wide) and the scaling is independent of key width. While merge sort continues to scale with SIMD even at large SIMD widths, the scaling of radix sort continues to fall. Therefore merge sort should be the sorting method of choice for future databases.

## 9. CONCLUSIONS

We present an comparative analysis of comparison and non-comparison based sorting algorithms on CPUs and GPUs. We propose efficient CPU radix sort and GPU merge sort implementations that are 2X faster than published results. We used the best comparison sort (merge sort) and best non-comparison sort (radix sort) and found that while radix sort is faster on current architectures, the gap narrows from CPUs to GPUs. Merge sort performs *better* than radix sort for sorting keys of larger sizes, which will be required for future databases with larger cardinality. We present analytical models for analyzing the performance, which points to merge sort winning over radix sort on future architectures due to its efficient utilization of SIMD parallelism and low bandwidth utilization. We conclude that *SIMD-friendly bandwidth-oblivious merge sort* should be the sorting method of choice for future databases.

## 10. REFERENCES

[1] CUDPP: CUDA Data Parallel Primitives Library. gpgpu.org/developer/cudpp/.

[2] Intel Performance Primitives. http://software.intel.com/en-us/intel-ipp/.

[3] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3):367–432, 1995.

[4] K. E. Batcher. Sorting networks and their applications. In *Spring Joint Computer Conference*, pages 307–314, 1968.

[5] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for column stores. In *SIGMOD*, pages 283–296, 2009.

[6] G. E. Blelloch. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990.

[7] P. Bohannon, P. Mcllroy, and R. Rastogi. Main-memory index structures with fixed-size partial keys. In *SIGMOD*, pages 163–174, 2001.

[8] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, et al. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, 2009.

[9] J. Chhugani, A. D. Nguyen, V. W. Lee, et al. Efficient implementation of sorting on multi-core SIMD CPU architectures. *VLDB*, 1(2):1313–1324, 2008.

[10] T. Cormen, C. Leiserson, and R. Rivest. *Intro. to Algorithms*. MIT Press, 1990.

[11] R. S. Francis, I. D. Mathieson, and L. Pannan. A fast, simple algorithm to balance a parallel multiway merge. In *Proceedings of PARLE*, 1993.

[12] N. Govindaraju, J. Gray, R. Kumar, et al. GPUTeraSort: High Performance Graphics Co-processor Sorting. In *SIGMOD*, pages 325–336, 2006.

[13] H. Inoue, T. Moriyama, H. Komatsu, et al. AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors. In *PACT*, pages 189–198, 2007.

[14] Intel Advanced Vector Extensions Programming Reference. 2008, http://softwarecommunity.intel.com/isn/downloads/intelavx/Intel-AVX-Programming-Reference-31943302.pdf.

[15] D. Jiménez-González, J. J. Navarro, and J.-L. Larriba-Pey. CC-Radix:a Cache Conscious Sorting Based on Radix sort. *Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, 0:101, 2003.

[16] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, et al. Sort vs. hash revisited: Fast join implementation on multi-core cpus. *PVLDB*, 2(2):1378–1389, 2009.

[17] A. Lamarca and R. E. Ladner. The Influence of Caches on the Performance of Sorting. In *Journal of Algorithms*, pages 370–379, 1997.

[18] N. Leischner, V. Osipov, and P. Sanders. Gpu sample sort, 2009.

[19] NVIDIA. Fermi Architecture White Paper, 2009.

[20] NVIDIA. *NVIDIA CUDA Programming Guide 2.3*. 2009.

[21] M. Reilly. When multicore isn't enough: Trends and the future for multi-multicore systems. In *HPEC*, 2008.

[22] N. Satish, , M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *IPDPS*, pages 1–10, 2009.

[23] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, et al. Larrabee: A Many-Core x86 Architecture for Visual Computing. *SIGGRAPH*, 27(3), 2008.

[24] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan Primitives for GPU Computing. In *Graphics Hardware 2007*, pages 97–106, Aug. 2007.

[25] E. Sintorn and U. Assarsson. Fast Parallel GPU-Sorting Using a Hybrid Algorithm. In *Workshop on GPGPU*, 2007.

[26] K. Thearling and S. Smith. An improved supercomputer sorting benchmark. In *Proceedings of Supercomputing '92*, pages 14–19, 1992.

[27] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings of Supercomputing '91*.

[6] an $S$-wide SIMD can store $S$ 32-bit values, $S/2$ 64-bit values or $S/4$ 128-bit values.