Fast Sort on CPUs, GPUs and Intel MIC Architectures

Technical Report

Intel Labs

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

Throughput Computing Lab.

Executive Summary

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 [29], we had earlier presented a competitive analysis of comparison and non-comparison based sorting algorithms on CPUs and GPUs. In this report, we extend this comparison to the Intel Many Integrated Core (MIC) architecture. We evaluate radix sort on Knights Ferry (an implementation of Intel MIC architecture), obtaining a performance gain of 2.2X and 1.7X over the best sort performance on the Intel Core i7 CPU and GTX 280 respectively. We also improve the performance of GPU radix sort by 1.6X over previous results.

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 previous work, [29], we evaluated how two main memory sorting algorithms, a radix and a merge sort, performed on CPUs and GPUs.

In this work, we evaluate radix sort on the upcoming Intel MIC architecture [28], a Larrabee [23] based silicon platform for many-core research and software development. In particular, we implement our radix sort on Knights Ferry (KNF), an implementation of this architecture that has 32-cores each with 512-bit SIMD units. The implementation of merge sort on KNF is work-in-progress. We also improve the GPU radix performance by 1.6X from that reported in [22] using techniques to improve SIMD efficiency and Instruction-Level Parallelism.

2. Radix Sort

In this section, we describe our changes to the GPU radix sort algorithm as well as the radix implementation on the Intel MIC Architecture. Both these implementations use the SIMD-friendly 1-bit split algorithm described in [29].

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 2D 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

2.1 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.

2.1.1. Detailed Implementation

In this section, we describe our GPU radix implementation.

We describe the GPU algorithm using the notations described above. 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 B. Assign each block to a GPU thread block. Each block computes a local histogram $H_m(d)$ for each bin d by incrementing appropriate bins for each input. This is achieved by maintaining per-thread local histograms in local shared memory, and using gathers/scatters to update these histograms.

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

Step 3: Each thread block first rearranges data locally on the basis of D bits by using D 1-bit stream splits operation. 1-bit splits are implemented using scan operations on the GPU [22]. To avoid the SIMD inefficiency of scans, it is best to perform scans in scalar code as much as possible. This can be achieved by a ***vertical*** scan organization – each thread in different SIMT lanes in a warp scans its own set of data items. Each thread produces the sum of its local values as it is done scanning its values. Threads within each warp then cooperatively use a SIMD scan algorithm to scan these sum values. The scan values corresponding to each thread are then independently added back by the thread to each of its individual values.

On the GTX 280, C = 16KB. For best efficiency, we found that each thread within a thread warp must independently scan at least 13 elements. This corresponds to a total of 13*32 = 416 elements/warp (~1.664 KB). We can then run a total of 16 KB/1.66 ~ 9.5 warps. We found using 8 warps (arranged as 2 blocks of 4 warps each) yields the best performance. 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.

2.1.2. Analysis

Step 1 is the histogram computation. Each thread performs spends 4 ops in reading the element and computing the index and 2 ops to scatter into shared memory. Histogram computations can be organized to have no shared memory bank conflicts - hence gets good IPC. The per-thread histograms are then reduced into a per-data block histogram using 32 scalar operations per histogram bin, for a total of 32*64 ops per data block. Thus in total **Step1 takes** 6*416 + 32*64 ops/block, or **~** 12 scalar ops/element. This translates to **less than 0.3 SIMD ops/element** (using 32-wide SIMD). From a bandwidth perspective, each element requires 4 bytes to be loaded and 64 bytes per block (416 elements) for histogram (1 byte per histogram bin). This is a total of ~ 4.25 bytes per element. GPUs get a high bandwidth of about 2.8 bytes/cycle/core – but **Step 1 is still bandwidth bound** due to the low computation performed – the expected bandwidth bound time is about 1.4 cycles per element.

Step 2 takes negligible time. Step 3 involves a local 4-bit sort of data using 4 1-bit stream splits.

Each split operation involves 14.5 ops per element (4 ops to read data from shared memory and compute the bit, 1.5 in the local thread scan, 1 to perform the coordinated warp scan, 1 to add back the warp scan results to the local thread scan, 6 to compute the rank of each element and 1 to move the values according to the rank). In addition, the first split requires a memory to buffer read consuming 4 ops, and the last split also requires 4 ops to write back data from shared to global memory. In addition, we measure about 6 extra ops of overhead in setting up registers before the compute loop. Step 3 thus takes a total **of 70 ops for the 4-bit split (or 2.2 SIMD ops per element)**. In terms of bandwidth, each data element needs to be read and written once, for a total of 8 bytes/element, which requires 2.9 cycles on the GTX 280. Step 3 is expected to be compute bound, and should take about 9 cycles per element. Most of this time is spent in the **split operations**, which take 59 scalar/1.84 SIMD ops, which is about 1.84/(2.2 + 0.3) = 73% of the total operations. In practice, since Step 1 is bandwidth bound, we expect the **split operations to take about 65% of overall time**. The split operation thus has significant impact on performance; for instance, introducing hardware to double the speed of the split operation will increase the overall sorting rate by about 1.5X.

2.2 Radix sort on MIC Architecture

We study how radix sort would perform on the Intel Many Integrated Core (MIC) Architecture [28]. The MIC architecture is an x86-based many-core processor architecture based on small in-order cores that uniquely combines full programmability of today's general-purpose CPU architectures with compute-throughput and memory bandwidth capabilities of modern GPU architectures. Each core is a general-purpose processor, which has a scalar unit based on the Pentium processor design, as well as a vector unit that supports 16 32-bit float or integer operations per clock. The MIC architecture has two levels of cache: low latency L1 cache and larger globally coherent L2 cache that is partitioned among the cores. Knights Ferry (KNF) [28] (an implementation of the MIC architecture), has a 32 KB L1 cache and 256 KB partitioned L2 cache. To further hide latency, each core is augmented with 4-way multithreading.

Since MIC has a large 16-wide SIMD, we adopted the SIMD-friendly split-based sorting technique. We follow the same 3-step radix sort algorithm as for CPUs and GPUs, with a histogram, prefix sum and scatter stages. Given the larger cache size of 256 KB per core on KNF as compared to the 16 KB shared memory per core on GPUs, we perform 6 bits per pass rather than 4 (expect the last stage when we handle 8 bits per pass). Each pass is implemented using a 1-bit split approach where efficient SIMD operations for gathering the 0's and 1's are used. This helps save bandwidth compared to the GPU. Our KNF radix sort is about 1.7X faster than the GTX 280.

3. Performance Evaluation

Machine Configuration: We now evaluate the performance of our radix sort implementations on GPUs and the Intel MIC architectures with the CPU results reported in [29]. The CPU is an Intel quad-core 3.2 GHz Core i7 CPU and the GPU is an NVIDIA GTX 280 GPU running at 1.3GHz. Both the CPU and GPU systems have 4 GB RAM and run Linux. The MIC architecture is a Knights Ferry (KNF) system running at 1.2 GHz.

3.1 Comparative Analysis across architectures

We first present our sorting performance results on a random distribution of 32-bit integers. Figure 1 presents our results of (a) the best radix sort implementation and (b) the best merge sort on Core i7 and the GTX 280. We additionally report our radix sort performance on the Intel MIC architecture. The merge sort implementation on Intel MIC is work-in-progress. The best radix sort implementation on Core i7 is our scalar buffer version while the best one on the GTX 280 and MIC 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 logN) 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 1million.

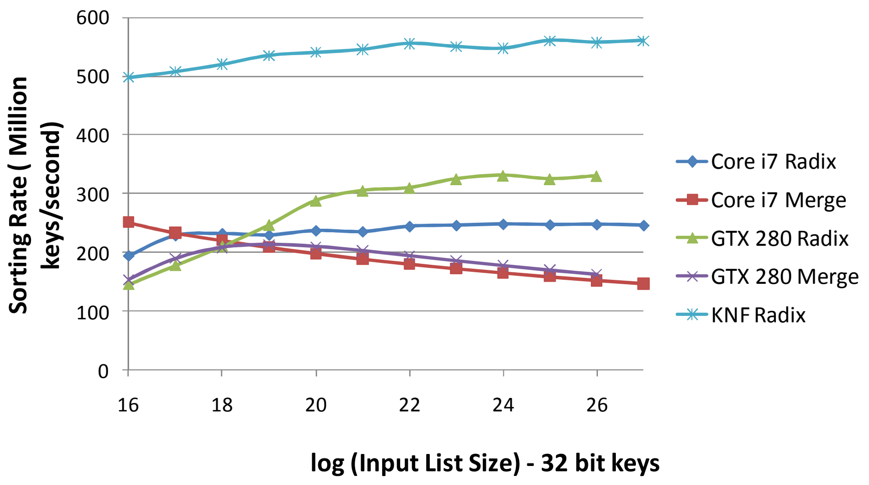

Figure 1: Sorting performance of radix and merge sorts on Intel Core i7, NVIDIA GTX 28 architectures. Radix sort performance on the Intel KNF architecture is also shown.

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 takes2.5 cpe (which is a 3.8X speed-up over scalar code). Radix sort always takes 4 passes, and hence 52 cpe. Merge sort takes logN passes and hence 2.5logN cpe. Our radix sort performance is the best reported performance on CPU architectures to date.

GPU performance: On the GTX 280 GPU, radix sorts is faster than merge sort for large N. Radix sort takes about 115 cycles/element, which is spent in performing 32 1-bit splits (each takes about 2.3 cycles per element), plus the time taken for histogram computations (2*8 = 16 cycles/element) and writing out local sorted data to global memory (3*8 = 24 cycles/element). The throughput of the split code is about 325 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.

MIC performance: On the MIC architecture, our best radix sort implementation uses the large L2 cache to buffer more data than possible on GPUs; and hence performs fewer passes over the data. This allows for a 1.8X gain in performance over the GTX 280. Our sort runs at about 560 M elements/second.

Comparing CPUs and GPUs: In terms of absolute performance of CPU versus GPU, we find that the best radix sort, the GPU radix sort outperforms the best CPU sort by about 20%. The reason for the relatively low speedup 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. 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.

3.2 Comparison to Analytical Model

We analyze the performance of our new radix sort algorithm on GPUs and compare it to the analytical model presented earlier.

GPU Radix Sort: The number of SIMD instructions in each step closely matches expected numbers. Step 1 is bandwidth bound, but with each pass taking about 2 cycles/second rather than the expected 1.4 cycles/second in bandwidth. Step2 takes negligible time. Each of the 4 1-bit splits of Step 3 takes about 0.52 instructions/element, closely following our analysis – for a total of ~ 2.08 instructions/element. Each SIMD instruction takes 4 cycles; hence the 1-bit splits take a total of 8.32 cycles/element per pass. Step 3 also involves global memory reads and writes, which take an additional 4 cycles/element per pass (there is some impact due to memory latency). Overall, our algorithm takes about 14.8 cycles/element per pass, or a total of 14.8*8 = 118.4 cycles/element for 8 passes. This accounts for our performance.

3.3 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.

Table 1 compares the best reported running times of comparison and non-comparison sorts on CPU and GPU platforms. On the CPU platform, a 3.2 GHz Corei7, 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.

	CPU (Core i7)				**GPU (GTX 280)**		
	Radix	IPP [2]	Merge [9]	*Radix*	*Radix*	*Merge*	Sample [18]
256K	*1.1*	1.2	1.2	*2.1*	*1.1*	*1.3*	2.5
1M	*4.4*	5.2	5.3	*9.6*	*3.5*	*5.0*	9.1
4M	*17.2*	25.4	23.3	*40.9*	*12.4*	*21.6*	38.1
16M	*67.6*	160.7	101.5	*185.7*	*49.5*	*94.5*	139.8
64M	*271.0*	550.5	439.7	*835.5*	*193.3*	*381.8*	524.3

Table 1: Performance comparison of the best performing sorting algorithms across platforms. We italicize our sorting implementations. Running times are in milliseconds, so lower numbers are better.

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 our optimized radix sort implementation. Our merge sort is slightly faster than radix up to 256 K elements, but the complexity of radix sort takes over at larger inputs.

4. 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 ids 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 rids 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].

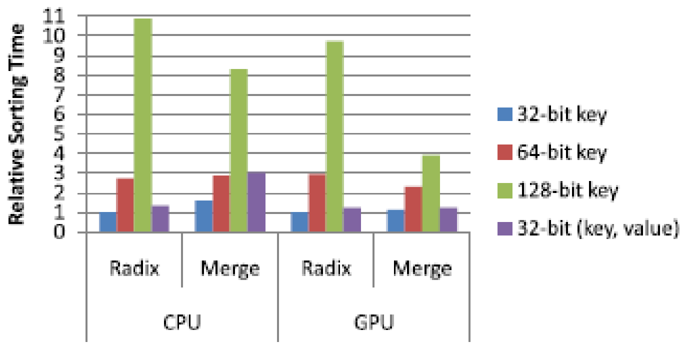

Figure 2: Performance of radix and merge sort on large keys (relative to 32-bit radix performance) on Intel Core i7 and NVIDIA GTX 280.

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 2 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 2 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.

5. Conclusions

This paper presents new results for GPU radix and Intel MIC architecture. We improve the GPU radix sort performance by 1.6X. We present analytical models for analyzing the performance. We evaluate radix sort on Knights Ferry (an implementation of Intel MIC architecture), obtaining a performance gain of 2.2X and 1.7X over the best sort performance on the Intel Core i7 CPU and GTX 280 respectively.

6. Future Work

As future work, we intend to implement merge sort on the Intel MIC architecture and evaluate both merge and radix sorts on the NVIDIA Fermi architectures.

7. 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. McIlroy, 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-AVXProgramming- Reference-31943302.pdf.

[15] D. Jiménex-González, J. J. Navarro, and J.-L. Larriba-Pey. CC-Radix:a Cache Conscious Sorting Based on Radix sort. Euromicro Conference on Parallel, Distribu

ted, 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.

[28] Kirk Skaugen, "Petascale to Exascale: Extending Intel's HPC commitment", ISC 2010 keynote. http://download.intel.com/pressroom/archive/reference/ISC_2010_Skaugen_keynote.pdf

[29] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey, "Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort". Accepted at ACM SIGMOD 2010.