Hardware Acceleration of Database Operations

Jared Casper and Kunle Olukotun
Pervasive Parallelism Laboratory
Stanford University
{jaredc, kunle}@stanford.edu

ABSTRACT

As the amount of memory in database systems grows, entire database tables, or even databases, are able to fit in the system's memory, making in-memory database operations more prevalent. This shift from disk-based to in-memory database systems has contributed to a move from row-wise to columnar data storage. Furthermore, common database workloads have grown beyond online transaction processing (OLTP) to include online analytical processing and data mining. These workloads analyze huge datasets that are often irregular and not indexed, making traditional database operations like joins much more expensive.

In this paper we explore using dedicated hardware to accelerate in-memory database operations. We present hardware to accelerate the selection process of compacting a single column into a linear column of selected data, joining two sorted columns via merging, and sorting a column. Finally, we put these primitives together to accelerate an entire join operation. We implement a prototype of this system using FPGAs and show substantial improvements in both absolute throughput and utilization of memory bandwidth. Using the prototype as a guide, we explore how the hardware resources required by our design change with the desired throughput.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles

Keywords

Database; FPGA; Hardware Acceleration; Join; Sort

1. INTRODUCTION

Database systems have historically been largely constrained by disk performance. Now, with advances of memory technology, the amount of main memory available in large database systems has grown enough that many large database tables now reside entirely in main memory. While Moore's law continues to hold and the number of transistors available to chip architects continues to increase, power constraints limit the number of logic transistors that can be active at any given time on a chip [3]. It is unlikely that general purpose processing elements will ever be able to fully utilize the amount of memory bandwidth available to a chip while performing all but the most basic database operations. As an example, studies have increased join performance into the 100s of million tuples per second [7, 6], with 64-bit tuples this corresponds to a data bandwidth of one to five gigabytes per second. Modern chips, conversely, can achieve memory bandwidth over 100 GB/s [1]. Clearly using general purpose compute is leaving performance on the table.

To achieve maximum performance for in memory database operations, it will thus be necessary to move to special purpose processors. The move to heterogeneity is not a new idea. Researchers at Intel have proposed that architectures move from optimizing for the 90% case, the traditional 90/10 approach, to spending new transistors on dedicated accelerators for multiple 10% cases, the "10x10" approach. [5, 2].

In this paper we propose hardware designs that accelerate three important primitive database operations: selection, merge join, and sorting. These three operation can be combined to perform one of the most fundamental database operations: the table join. The primary goal in our designs is to build hardware that fully utilizes any amount of memory bandwidth available. To that end our designs have as few limiters to scaling as possible, such that as logic density increases more hardware can be added to increase the throughput of the design with very little redesign of the architecture. Contribution of this this work include:

- We detail hardware to perform a selection on a column of data streamed at peak memory bandwidth.
- We describe hardware to merge two sorted data columns.
- We present hardware to sort a data column using a merge sort algorithm.
- We describe how to combine these hardware blocks to perform an equi-join entirely in hardware.
- We prototype all three designs on an FPGA platform and discuss issues faced when building the prototype.
- We analyze the performance of our prototype and identify key bottlenecks in performance.
- For each hardware design, we explore the hardware resources necessary and how those resources requirements grow with bandwidth requirements.

FPGA'14, February 26–28, 2014, Monterey, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2671-1/14/02 ...$15.00.
http://dx.doi.org/10.1145/2554688.2554787.

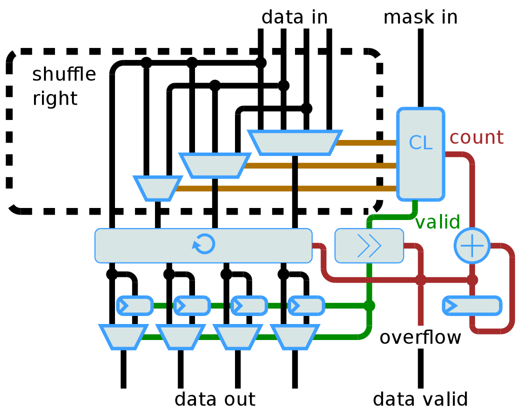

Figure 1: Data and control paths for selection of four elements.

2. HARDWARE DESIGN

2.1 Selection

In this paper we define the selection operation to take two inputs, a bit mask of selected elements and a column of data stored as an array of equal width machine data types. The inputs can either come from arrays laid out linearly in memory, or be produced by another operation which may be looking at a different column of data. In some cases the bit mask may be RLE compressed and must be decompressed before being used by the selection unit. A common case would have the bit mask coming from another operation and the data column being read from memory. The output of the operation is values from the input column that correspond to the true bits in the bit mask, in the same order that they appear in the original column. Like the input, the output data can be streamed to another processing unit or written sequentially into memory.

There are many ways to implement selection in software. One efficient implementation fills a SIMD register with the next values from the input column. A portion of the bit mask is used as an index into a look up table which contains indices for the SIMD shuffle operation to shuffle the selected data to one end of the SIMD register. The resulting SIMD register is written to the output array and the output pointer is incremented by the number of valid data elements that were written. This store is thus an unaligned SIMD memory access, which was added in SSE4, and has little performance impact when writing to the L1 cache. These unaligned stores are used to incrementally fill the output with compacted data. Parallel algorithms must first scan through the bit mask counting bits to determine the proper offset to begin writing each portion of the result. Once those offsets are calculated, the column can be partitioned for multiple threads to work on in parallel.

Hardware to perform this selection is presented in Figure 1. We call the number of elements consumed each pass through the hardware the "width" of the selection block. Assuming a fully pipelined implementation, the bandwidth of the block is fully determined by the width of the block and the clock speed. A barrel shifter can be efficiently implemented using multiple stages of multiplexors; however, such large barrel shifters must be pipelined to achieve high clock frequencies, so the datapath in Figure 1 was carefully designed to avoid feedback paths containing large barrel shifters which would necessitate pipeline stalls. As is, the only feedback path in the design is a very small addition (with width $log_2(W)$), allowing for a deeply pipelined design to achieve a high clock rate.

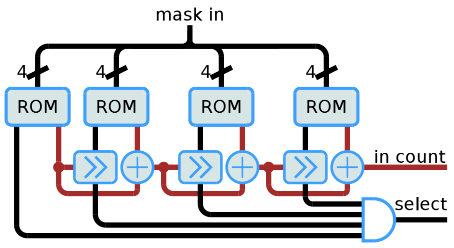

Figure 2: Control logic for the selection unit.

The first step is to produce a word array in which all selected words from the input are shuffled next to each other at one end of the array (in this case, the right side). A combinational logic block takes in a segment of the mask stream and produces a count of the number of selected elements in the segment, a bus of valid lines, and an index vector which specifies which word should be selected for each position in the shuffled word array.

For small input widths, this combination logic can simply be implemented as a single ROM. Such a ROM would have depth 2^W. This is clearly not feasible for any realistic input width. Using pure-combinational logic, such as a cascade of leading-1-detectors, would also not be feasible for larger input widths. We thus use smaller sections of the mask as addresses into multiple smaller ROMs. So for example, instead of using all 16 bits of a mask segment to address a 64k deep ROM, we can use each 4-bit nibble of the mask to address four 16 element ROMs. It is then necessary to shift the output of each ROM into the correct position of the final index vector, based on the accumulated count from the adjacent ROM. Figure 2 shows an implementation of this for an input width of 16. This datapath has no feedback paths and can thus be efficiently pipelined to achieve full throughput. Decreasing the size of the ROMS and including more of them results in lower total ROM space but higher latency and more adders, barrel shifters, and pipeline registers.

Once the selected values are shuffled to the right side, they are rotated left to a position indicated by the current number of saved values ready to be output. Values in the input that complete a full output are sent directly to the output and values that will make up a partial output are saved in registers. For example, if two values were previously saved in the registers, and three values are selected in the input, the input will be rotated right by two, such that the lowest (furthest right) two values fill the left two positions in the output, and the third input word is saved in the register furthest to the right, ready to be added to selected values from the next input.

2.2 Merge Join

The merge join operation takes two sorted columns of fixed-width keys as input, each with an associated payload column, and produces an output column which contains all the keys that the two columns have in common, together with the associated payload values. When there are duplicate matching keys, the cross product of all payload values

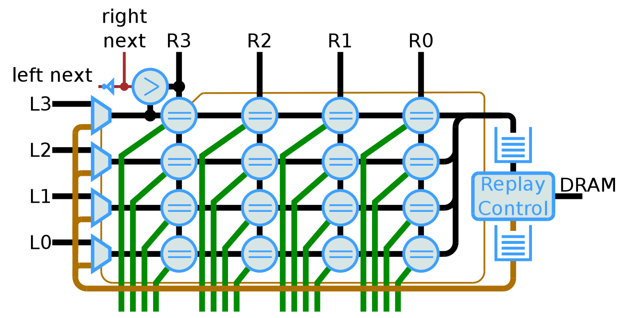

Figure 3: Hardware to perform the merge join operation. The green lines exiting diagonally from each comparator encompass the key, both values, and the result of the comparison.

are produced. For example, if there are four entries of a key x in one input column, and six entries of x in the other input, there will be 24 entries in the output with key x.

This operation can be performed in software by sequentially moving through each input column and advancing the pointer of the column with the lower value. When two keys match, an output row is written to the output array and the output pointer incremented. Care must be taken to handle the case of multiple matching keys and produce the correct cross-section output. The resulting code has a large number of unpredictable branches that result in a very low IPC and quickly becomes processor bound, not able to keep up with the memory bandwidth available to even a single core.

Our hardware design to perform this operation is laid out in Figure 3. The basic design is rather straightforward; all combinations of a section of keys from each of two inputs ("left" and "right") are compared. An array of possible output combinations with a bit mask indicating which should be used is produced. This output can then be sent into the selection unit from Section 2.1 to produce the actual output rows. The highest value from each input is compared, the input with the lower highest value is advanced, while the same selection from the other input remains. This ensures that any combination of input keys that could potentially match are compared.

Complications arise, however, when the highest value of each input selection is equal. In this case it is necessary to buffer the keys from the left input and advance through the left input until the highest keys no longer match. When that happens, it is guaranteed that the highest right input is lower than the highest left input, and the right input can be advanced. Any values buffered are then replayed and compared against the new selection from the right. When the replay buffer is empty, execution continues as normal.

Because the number of comparators grows quadratically with the width of input, it is difficult to implement hardware with a wide input array. An optimization to help increase the throughput of the design looks at a much wider selection of each input than the actual comparator grid. The input is partitioned into sections that fit into the comparator grid and the highest and lowest values are compared. Using those comparisons, only those cross sections with potential matches are sent into the comparator grid sequentially while the others are skipped.

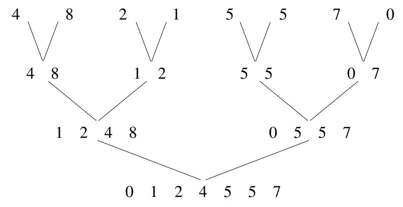

Figure 4: Sorting using a sort merge tree.

2.3 Sorting

Sorting an array, or column, of numbers has been and will continue to be a very active area of research and is an essential primitive operation in many application domains, including databases. Quicksort based algorithms have traditionally been considered to have the best average case performance among software sorting algorithms. However, recent advances in both CPU and GPU architectures have brought merge sort based algorithms, such as bitonic sort and Batcher odd-even sort, to the forefront of performance as they are able to exploit new architectures more effectively and better utilize a limited amount of bandwidth [4, 15, 17, 9]. Satish et.al.[16] provide a comprehensive overview of state of art sorting algorithms for general purposes CPU and GPU processors.

We present here a dedicated hardware solution to perform a merge sort entirely in hardware. The goal of this design is to sort an in-memory column of values while streaming the column to and from memory at full memory bandwidth as few times as possible. Figure 4 depicts the essence of a merge sort. We call the merge done at the individual node a "sort merge", which is distinguished from a "merge join" presented in Section 2.2. To accomplish this we implement a merge tree directly in hardware, stream unsorted data from memory into the merge tree and write out sorted portions of the column. Those sorted portions then become the input to each input leaf of the merge tree again, generating much larger sorted portions. This process is repeated until the entire column is sorted. The number of passes required through the tree is dependent on the width of the merge tree. Thus, if the width of the tree is relatively large, the number of passes required grows extremely slowly with the size of the input table and very large tables can be sorted in just two or three passes of the data.

Before we describe the design of the merge tree itself, we first look at an individual node in the merge tree. The maximum throughput of data through the merge tree will be ultimately limited by the throughput of data through the final node at the bottom of the tree. Depending on the data, other nodes of the tree can also become a bottleneck. For example, if the far left input on a second pass contains all of the lowest elements of the full column, then only the far left branches of the tree will be used until the entire portion is consumed. It is thus not practical to move only the lowest single value of the two inputs of a node to the output. This would result in the throughput of the tree being only one element per cycle. Multiple values must be merged every cycle.

Figure 6 gives a logical overview of how multiple values from the input are merged at a time. Each iteration, the lowest value of each input are compared and some number of values, in this case four, are removed from the input queue

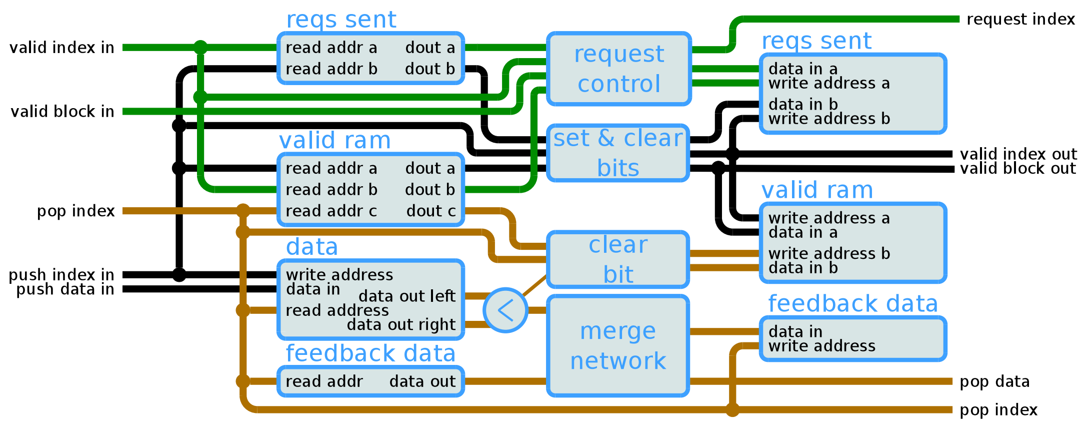

Figure 5: Sort merge unit. Note that for simplicity, ports to the same memory are separated.

with the lower lowest value. These four values are merged with the highest four values from the previous iteration. The four lowest values resulting from that merge are guaranteed to be lower than any other value yet to be considered since any values lower than the fourth would already have been pulled in. The highest four values, however, may be higher than and must therefore be fed back and merged with the next set of input values. In this way, four values are produced and four values are consumed from one of the inputs each iteration.

It is not necessary, however, to put a merge network like that in Figure 6 at each node of the tree. Each level of the tree need only supply values as fast as the level below it can consume values. Thus, each level need only match the throughput of the final node of the tree, which need only match the write memory bandwidth to keep up with memory. Figure 5 presents the hardware that encompasses a single level of a merge tree, which we call a “sort merge unit”. A data memory buffers the input data to the level. It is only necessary to hold as a single value for each input leaf to the level. The data memory is partitioned into “left” and “right” data so that both inputs to a particular node can be read at once, but each can be written separately. Another memory holds the feedback data from the previous merge of values for each node in the level. A valid memory holds a bit for each input leaf to indicate that the data for that leaf is valid, and a bit for each entry in the feedback memory. These valid bits are blocked in chunks, so a single read or write works on multiple values at once. Finally, a “request sent” memory, which is blocked like the valid memory, holds a single bit for each input leaf to indicate that a request has been sent up the tree to fill the data for that leaf. Note that there are no output buffers, as the outputs are buffered at the next level in the tree.

We now describe three operations performed on a sort merge unit: a push, a request, and a pop. A push, whose data path is black in Figure 5, is performed when input data comes from above the unit in the tree. First, the data is written to the data memory, which is known to be invalid because it was previously requested, and the valid and request outstanding blocks are read. The corresponding valid bit is set and the request outstanding bit is cleared, and the new blocks are written back. The new block of valid bits is also sent down to the lower level along with the index. If

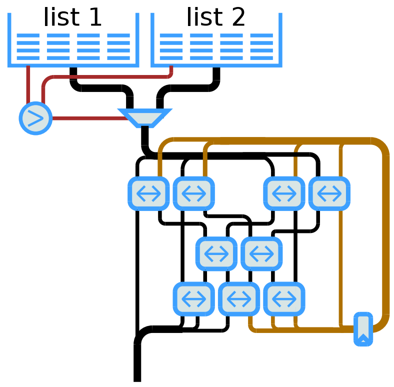

Figure 6: Merging multiple values at once.

nothing is being pushed in a particular cycle, a valid block (determined by an internal counter) is still read and sent down to the lower level, this is not shown in the figure and prevents deadlock in some cases.

When the valid block and associated index are sent to a sort merge unit, it initiates a request operation, which follows the green data path in Figure 5. First, the level’s own valid and request outstanding blocks corresponding to the valid bits received are read. The incoming valid block, which represent data valid at nodes above, and the local valid and request outstanding blocks are examined to to find invalid elements that have two valid parents and have not been requested. One such element is selected, a bit for it is set in the request outstanding memory, and the request is sent up to the parent.

Finally, an incoming request from below results in a pop operation, which follows the orange data path. Both data values, the feedback data, and corresponding valid block are read. The lowest values in each data buffer are compared. The block with the lowest is sent to the merge network along with the feedback data (if valid) and the valid bit corresponding to the consumed leaf is cleared while the valid bit for the feedback data is set. The lower values from the merge network are sent to the next level to pushed and the higher values are written back into the feedback memory.

As mentioned previously, the throughput of the entire merge tree is limited by the throughput of the final node

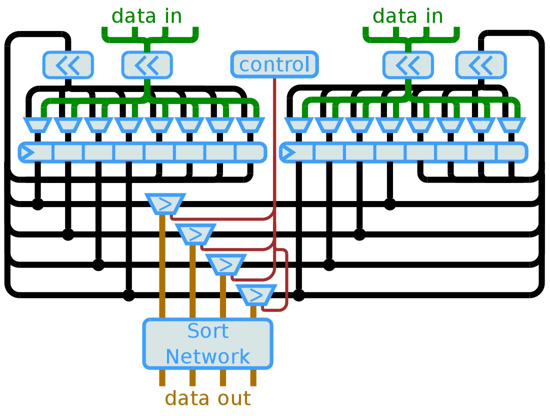

Figure 7: High bandwidth sort merge unit.

in the tree. The design in Figure 5 works well when there are plenty of inputs and outputs to fill the pipeline; merges of multiple nodes in the level are happening simultaneously. However, the final node of the tree has only two inputs. That means that an entire iteration must complete before the next merge can begin. It is thus insufficient for use in the final nodes of the tree.

Figure 7 presents a higher bandwidth sort merge unit which implements a single node of the tree, not an entire level with multiple nodes like Figure 5. Instead of consuming and merging a set number of values from one of the inputs, shift registers are used to consume a variable number from each input and new values are shifted in as space becomes available. Let W be the number of values to output each iteration. Let L_i and R_i be the values in the left and right shift registers, respectively, with i ranging from 0 to $2W - 1$. To determine the four lowest value from across both shift registers, each L_x is compared with $R_{(W-1-x)}$ for x between 0 and $W - 1$. The lower of the two in each case is advanced to the sort network while the higher remains in the shift register. For example, if $L_0 < R_3$, then at least one from the left and no more than three from the right are among the lowest, so L_0 is necessarily one of the lowest and R_3 is necessarily not. Likewise for L_1 and R_2, L_2 and R_1, and L_3 and R_0. The number taken from each side is counted and the shift register is shifted by that amount. If there is enough free space in the shift register, an input section is consumed, shifted, and stored into the correct position. The four lowest values are then sent into a full sort network and passed down to the next level. A merge network like that in Figure 6 is insufficient here since the input is not necessarily split into two equally sized, already sorted arrays.

The datapath in Figure 7 still has feedback paths which prevent a pipelined implementation from being fully utilized; the critical feedback path is a bit count, barrel shifter, and 2:1 multiplexor. This path is much shorter and grows much less quickly as the width increases than the feedback path of Figure 5 which include a full merge network.

Finally, Figure 8 shows the datapath for a full merge tree. A “tree filler” block has the same interface as a sort merge unit, but fulfills requests by fetching from DRAM. It continually sends blocks of “valid” bits which indicate that data is still available for a particular input, turns requests from the top level of the merge tree into DRAM requests, and turns replies from DRAM into pushes into the top sort merge unit. During the initial pass through the memory, the data for an input can come from anywhere, so the input column is read linearly and sent through a small initial bootstrap sort network since the sort merge units expect blocks of sorted data as input. To prevent very wide levels that make routing more difficult, the top levels of the tree are split into four sub-trees, which operate independently of each other. The final two levels of the tree are the high bandwidth merge sort unit to maintain the total throughput of the tree and merge the output of the four lower bandwidth trees to produce a single sorted output.

On passes after the initial pass through data, the tree filler must obtain data from the particular sorted portion that matches the tree input of the request. Depending on the number of portions remaining to be merged, the tree filler maps some number of inputs of the tree to each of the remaining portions. For example, if the full tree is 16k inputs wide and there are four portions remaining to be merged, the first portion is mapped to the first 4k inputs, the second to the next 4k, etc. This means that some values of the portion are re-merged, but also has the effect of using sections of the tree as an input buffer for each of the portions. The fewer portions that remain to be merge, the larger the “input buffer” for each portion is and the larger the requests to DRAM can be. When the number of portions remaining to be sorted is equal to the number of inputs to the tree, only a single chunk of a portion can be requested at a time, leading to inefficient use of the DRAM bandwidth. We see the results of this in Section 3.3.

To support using portions of the merge tree as an input buffer in subsequent passes, the tree filler keeps a bit mask of tree inputs that it has received a request for. When enough of the inputs mapped to a particular portion have been requested, a single large request for the next values in that portion are requested and all of the requests are fulfilled in bulk.

2.4 Sort Merge Join

A full join operation is the same operation as a merge join, described in Section 2.2, but does not require the input columns to be sorted. Two main algorithms are most often used to perform joins, a hash join and sort merge join [7]. A hash join builds a hash table of one of the two input columns, then looks each element of the other column up in the hash table to find matches. Modern hash join implementation use sophisticated partitioning schemes to parallelize the operation and utilize a processors cache hierarchy. A sort merge join simply sorts both input columns then performs a merge join on the sorted columns. Implementations leverage the massive amount of research to improve the performance of sorting.

Figure 8 shows how each of the three blocks previously described can be combined to perform an entire sort merge join in hardware. Two independent sort trees are used to sort each of the two input columns. On the final pass through each column, the sorted data is sent to the merge join block instead of back to DRAM. The merge join output is sent to the select block as before and only the result of the join operation is written back into DRAM. The design also include data paths that allow the sort, merge join, and select blocks to be used independently of each other.

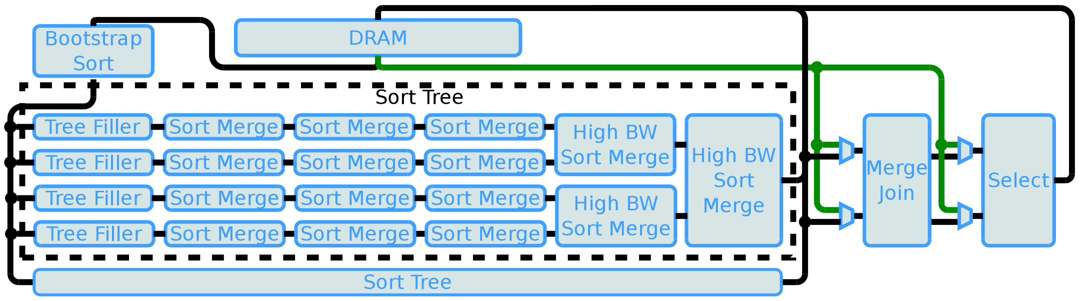

Figure 8: Full system block diagram and data paths.

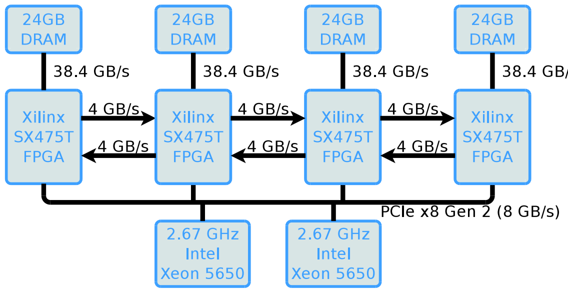

Figure 9: Block diagram of prototyping platform from Maxeler Technologies.

3. IMPLEMENTATION AND RESULTS

To prototype the design we used a system from Maxeler Technologies described in Figure 9. This system features four large Xilinx Virtex-6 FPGAs (XC6VSX475T). Each FPGA has 475k logic cells and 1,064 36 Kb RAM blocks for a total of 4.67 MB of block memory. Each FPGA is connected to 24 GB of memory via a single 384 bit memory channel capable of running at 400 MHz DDR, for a line speed of 307.2 Gbps, or 38.4 GB/s per FPGA. This gives a total line bandwidth between the FPGAs and memory of 153.6 GB/s, comparable to modern GPUs. The FPGAs are connected in a line with connections capable of 4 GB/s in each direction. For each design, we clocked the FPGA fabric at 200 MHz. Finally, each FPGA is connected via PCIe x8 to a host processor which is two 2.67 GHz Xeon 5650s, each containing 6 multi-threaded cores. These processor each have a line memory bandwidth of 32 GB/s.

Our purpose in prototyping the design was not entirely to determine the performance of the design, although we do provide performance numbers. As long as the components are able to match or exceed the memory bandwidth, the performance is largely determined by the memory system of the design, and thus many of the performance results are as much a test of Maxeler's memory system as they are of the acceleration design. Our main purpose in building the prototype was to drive the design using a real world implementation instead of what are often inaccurate simulations, and to be able to determine the challenging issues that arise as the hardware scales to higher bandwidths. Indeed, the final designs we have presented are fairly different from the original designs we came up with based on early simulations.

We chose the Maxeler platform for the large amount of memory capacity and bandwidth available to the FPGAs; we wanted to ensure that our prototype handled a sufficient amount of bandwidth to prevent masking any scalability issues. The largest performance bottleneck we faced using the platform is the relatively narrow intra-FPGA links, which prevented us from effectively emulating a single chip with a full 153.6 GB/s of memory bandwidth. Thus, for all but Section 3.4, we use a single FPGA, since using the narrow intra-FPGA links skews the results in terms of the memory bandwidth utilization.

Since many of the performance numbers are dominated by the performance of the memory system on the Maxeler platform, we also present percentage of the maximum memory throughput (by which we mean the line bandwidth of the memory interface) as a metric of comparison. Since our hardware is designed to scale with available bandwidth, these percentages give an idea of how the design would perform with different memory systems. They also provide a metric of comparison with previous work, as it is difficult to make a true "apples-to-apples" comparison when the hardware is so vastly different. We also give some intuition as to how the resource requirements of each design will scale to platforms with different memory bandwidths.

3.1 Selection

We implemented the software algorithm described in Section 2.1 and optimized at the assembly language level. On our system's host processor, this implementation is able to achieve a maximum throughput using 8 threads, with an average throughput of 7.4 GB/s and 6.0 GB/s as the selection cardinality moves from 0% to 100%. This corresponds to 23.1% to 18.8% of the 32 GB/s maximum memory throughput of the Xeon 5650. For reference, the STREAM benchmark[10] also achieves the maximum bandwidth with 8 threads and is able to copy memory at a maximum speed of 11.8 GB/s[1], about 36.8% of the line rate memory bandwidth of the Xeon 5650. Results reported on the STREAM benchmark website [10] indicate that this utilization of maximum memory bandwidth is typical for modern processors.

Our implementation uses three SIMD registers, one to hold the data to be shuffled, one to hold the bit mask, and one to hold the shuffle indices loaded from memory. Thus, the lack of available SIMD registers accounts for the inability of the processor to fully pipeline the selection process and achieve the throughput of STREAM. The Xeon's in our

[1]The STREAM benchmark reported 23.6 GB/s, but counts bytes both read and written, or the "STREAM" method; the number here is for the "bcopy" method, which counts total bytes moved, which is more aligned with our use of bandwidth in this work.

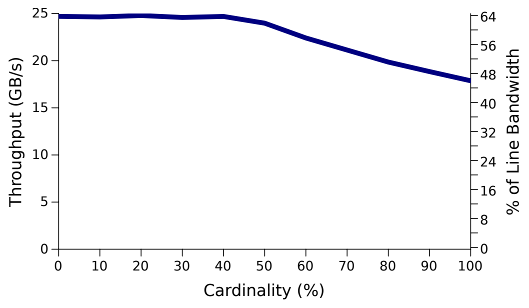

Figure 10: Measured throughput of the select block prototype.

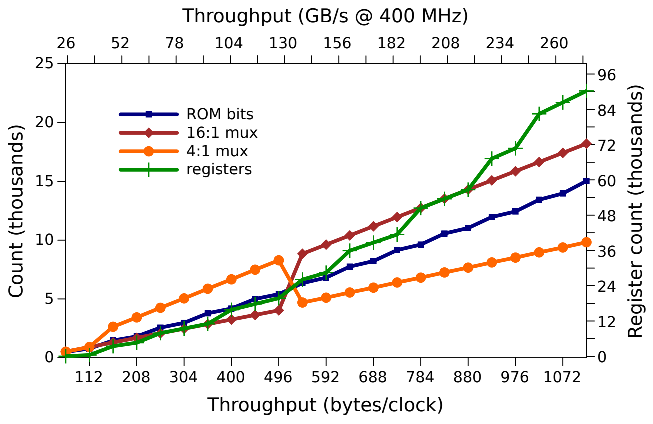

Figure 11: Amount of resources needed as the desired throughput of the select block increases.

test system support 16 byte wide SIMD instructions; using the 32 byte wide AVX2 integer instructions in the Haswell processors we would expect better performance and a highly tuned software selection algorithm to match the throughput of STREAM.

The design in Section 2.1 maps almost directly to the FPGA platform and we built a block that processes 72 64-bit values per clock cycle, for a maximum throughput of 14.4 billion values per second, or 115.2 GB/s. This is much more than the memory bandwidth available to a single chip; we will see in Section 3.2 why we made it that wide.

Figure 10 shows the measured throughput of the prototype. Throughout Section 3, bandwidth numbers are measured as the number of input bytes processed per second [2]. We could alternatively use total number of bytes read and written. This is pertinent here because a selection with cardinality of 0% transfers half the amount data as one with cardinality of 100%. With a constant amount of memory bandwidth that can be used for either reading or writing data, the 100% case will take longer to execute, but would have higher throughput if bytes both read and written were counted. Counting only bytes read, the cardinality of 100% case shows lower bandwidth since it takes longer to process the same amount of input data. This explains the nearly linear drop from 24.7 GB/s down to 17.8 GB/s as the cardinality moves from 40% to 100%. Below 40% the limits of a single port of the DRAM controller are reached and the full line rate of the memory interface is not realized. At 100% cardinality, the memory controller is more efficient with two streams of data (in and out) and is able to utilize 93% of the 38.4 GB/s of line bandwidth. This high utilization is achieved because of the very linear nature of the data access pattern and by putting the source and destination columns in different ranks of the DRAM, preventing them from interfering with one another.

At low cardinalities, the 24.7 GB/s achieved is 64.3% of the 38.4 GB/s maximum memory throughput of the FPGA. This represents a 2.8x increase in the memory bandwidth utilization over the 23.1% utilization of the software, and a 1.7x increase over the STREAM benchmark, which is as high as any software implementation could possibly achieve.

We now look at the number of resources required to scale the design. Figure 11 shows the resources used by the implementation as the width, and thus bandwidth, of the block increases (note the different scale for registers and the other components). We present throughput as bytes per clock to decouple the results from any particular frequency, but also present GB/s at 400 MHz for reference. The range in throughput represents the range in width from 8 to 144 64-bit words. In choosing the number of stages used in the initial shuffle control (see Section 2.1), we experimentally found a good number of stages to use is $W/4$, where W is the width in words of the selection block.

Note that the numbers in Figure 11 present resources at the bit level. So a multiplexor that select between 4 64-bit words requires 64 4:1 multiplexors. For convenience, we lump 2:1 multiplexors in with 4:1 multiplexors and 8:1 multiplexors in with 16:1 multiplexors. Any multiplexor wider than 16 inputs is split into multiple stages to ease routing congestion and maintain clock speed. The swap that occurs at 496 bytes/block (or 62 to 68 words) results from the second stage of an 68:1 multiplexor requiring 16:1 multiplexors instead of the 4:1 second stage of smaller widths ($W/16 > 4$ when $W > 64$).

The most dramatic increase in resources as throughput increases comes from the number of registers. This results from the additional pipeline stages needed as the width increases. In addition to addition stages in the shuffle multiplexor and barrel shifter, we added duplicate registers to reduce fanout for each 16 inputs to help with the routing on the FPGA.

3.2 Merge Join

The prototype of the design presented in Section 2.2 is designed to merge two streams of elements composed of 32-bit keys and 16-bit values. Because of the high demand for routing resources, the structure did not map well to the FPGA fabric and we were able to achieve a block with a width of eight words for each input. The output combinations, which are a 32-bit key and two 16-bit values, and equality bit vector are sent into a selection block, which is wide enough to accept all 64 64-bit inputs.

The throughput of the prototype for varying amounts of output vs the input table size is presented in Figure 12. The line labeled "m=1" is the raw comparison grid without the optimization of not examining unnecessary cross sections. The other line, "m=8" shows the throughput for looking at 8 chunks of each input and only actually comparing

[2]Also note that "GB" is here is really gigabyte, not gibibyte, making percentage of line bandwidth, which is also in GB, not GiB, make sense

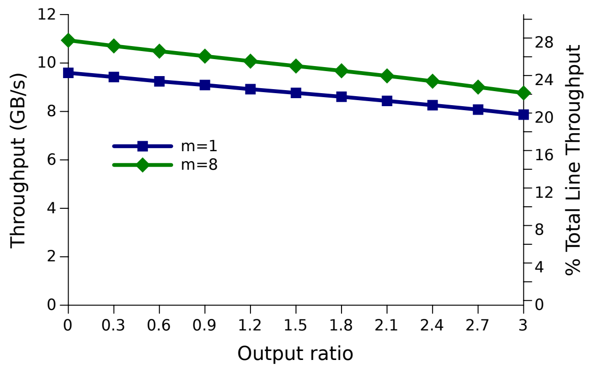

Figure 12: Throughput of the merge join prototype.

chunks with potential matches. The output ratio is the size of the output compared to the input table size (which is two equally sized tables). The keys are uniformly distributed within a range that is changed to vary the output ratio.

At low output ratios, the throughput is contrained by the throughput of the hardware block itself (eight six byte values at 200 MHz is 9.6 GB/s). As the output ratio increases, it is necessary to "replay" portions of the input more often (see Section 2.2) and the throughput decreases. Above a ratio of 1.5 (i.e. the output is 1.5 times the size of the input), the throughput is entirely limited by the write memory bandwidth. We looked at non-uniform distributions, but saw no variance in the throughput for any given output ratio. Most skewed data, such as data with a Zipf distribution used in the literature, produced a very large amount of output and were all limited by the write memory bandwidth.

We do not plot the required resources for the merge join block because it is dominated entirely by the comparators and routing resources and is simply a quadratic function of the bandwidth required. To consume N values from either input every cycle required N^2 comparisons. Higher bandwidth could be obtained by replicating the merge block and partitioning the data, but doing so is left for future work.

3.3 Sorting

Section 2.3's implementation handles 12 64-bit values every other 200 MHz cycle, providing a maximum throughput of 19.2 GB/s, which is able to keep up with the memory bandwidth of an individual FPGA (assuming a column is being read and written). One of the major challenges faced in implementing the low bandwidth merge sort unit was the number of memory ports needed. In particular, it was necessary to access five different addresses of the valid memory in any given cycle. The local memories on the FPGA have two full RW ports. To solve the issue we duplicated each valid memory and time multiplexed the ports, alternating between reading and writing (thus handling a new input every other cycle). Table 1 details how each port was used to achieve a virtual 5-port memory. Note that each copy must perform the same operation on the write cycle to maintain coherent duplication.

All the other structures mapped directly to the FPGA logic. To maintain 19.2 GB/s through the entire tree, the three high bandwidth sort merge units at the bottom of the tree were built to accept 24 values every four cycles to accommodate the feedback path. The most challenging aspect was getting the control for the fine grained communication

Memory	Port	Read Cycle	Write Cycle
valid copy 1	A	Read for push	Write for push
	B	Read for pop	Write for pop
valid copy 2	A	Read for request	Write for push
	B	Idle	Write for pop
request outstanding	A	Read for push	Write for push
	B	Read for request	Write for request

Table 1: Memory port usage in sort merge unit.

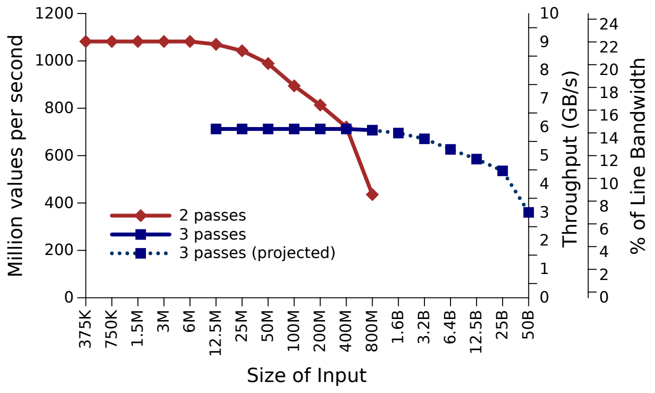

Figure 13: Throughput of the sort tree prototype.

between levels correct. As an example, the pop operation is pipelined to take six cycles: 1) start the read of data and valid blocks; 2) decode the index; 3) start the read of the feedback data; 4) the reads complete, compare the data; 5) multiplex the data based on the comparison result; 6) merge decoded index with read valid blocks, update the valid block, and send the feedback data and selected data to the merge network. At every other pipeline stage the index being pushed is compared with the incoming index and if the two fall within the same block, the decoded index, which indicates the valid bit to set, is updated and the incoming push is considered complete. The pipelines for the request and push operations are similar.

The memories on the FPGA provided enough space for 12 levels in the merge tree, with a top level 8k inputs wide. The data buffering alone for the merge tree (including the feedback data) occupied 18.6 Mbits, or 50%, of the 37.4 Mbits of block ram available on the device.

Figure 13 shows the throughput of the prototype as the size of the input column grows. Note that when performing two passes over the entire data set, the theoretical maximum throughput is one quarter of the maximum memory throughput (each value needs to be both read and written twice), or 9.7 GB/s in our case. At small input sizes, we achieve 8.7 GB/s, which is 22.7% of the maximum memory bandwidth, or 89% of the theoretical maximum with two passes. This high utilization is possible because there are fewer partially sorted portions to merge in the second pass and as a result each portion has a large virtual input buffer and the requests to memory can be large (see Section 2.3). For reference, recent work on sorting values on both CPUs and GPUs achieved rates as high as 268 million 32-bit values per second [16]. This corresponds to 1 GB/s of throughput, which is 3.9% of the 25.6 GB/s available to the Core i7 used (GPU performance was worse). We thus see a 5.7x improvement in terms of memory bandwidth utilization.

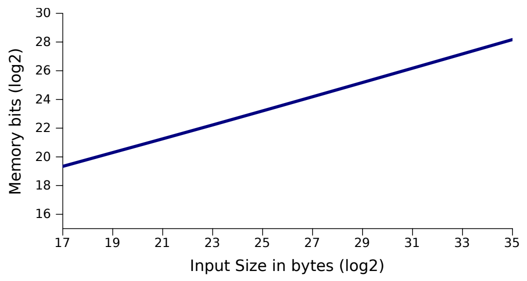

Figure 14: Memory bits required to achieve optimal throughput for a given input size. Note the log/log scale.

As the size of the input increases, the number of portions that must be merged on the second pass increases and the size of the requests to memory decrease. At an input size of 25M values, the memory requests are too small to fully utilize the memory bandwidth and performance begins to degrade. When the input size reaches 400M values, there are enough portions in the second pass that it is advantageous to perform a third pass. In this case, the portions from the first pass are partitioned into groups small enough that large memory requests can be used. Above 800M values, there was insufficient memory to hold both the input and output columns, we therefore projected the performance for larger columns using predictions based on the throughput seen on the second pass of smaller columns.

Unlike the previous sections, the interesting resource metric is not how the resource usage grows with desired bandwidth, but how the resource usage grows with input size, keeping bandwidth constant. A very small merge tree could maximize bandwidth for small inputs, but performance would rapidly decrease as input size grows. For example, our prototype was able to use the maximum amount of memory bandwidth until the input was over 12.5 million values. To see where this limit comes from, let N be the size of the input, in bytes, and let W be the width of the top level of the tree in bytes (in our prototype $W = 8k * 12\ records * 8\ bytes/record = 786432\ bytes$). The number of portions left after the first pass through the data is $L = N/W$ and the maximum size of each read on the second pass is W/L, or W^2/N. If the minimum read size for optimal memory throughput is M, the maximum input size that achieves optimal memory performance is W^2/M. For the Maxeler platform, M is measured to be 6144 bytes, which gives a maximum size of 100 MB, or 12.5M 64-bit values. Likewise, W must be $\sqrt{M * N}$ for a table of size N to fully utilize the memory bandwidth on the second pass. Figure 14 provides the number of memory bits needed to achieve maximum memory bandwidth efficiency for given input sizes, provided a minimum read size of 6144 bytes.

To obtain the highest throughput possible using our platform, we tested a prototype where one quarter of the input column was split onto FPGAs 0 and 2, while the remaining three quarters were put on FPGA 1. With this configuration, the two smaller portions were individually sorted then streamed to the FPGA with the bulk of the data, and we achieved a throughput of 1.4 billion values per second, or 11.2 GB/s. With the narrow intra-FPGA links in play, this is a much lower percentage of the memory bandwidth available to the three chips used (9.7%).

System	Clock Freq	Throughput/ Mem BW (GB/s)	% of BW
Multi FPGA	200 MHz	6.45 / 115.2	5.6%
Single FPGA	200 MHz	6.25 / 38.4	16.3%
Kim [7] (CPU)	3.2 GHz	1 / 25.6	3.8%
Kaldewey [6] (GPU)	1.5 GHz	4.6 / 192.4	2.3%

Table 2: Summary of sort merge join results.

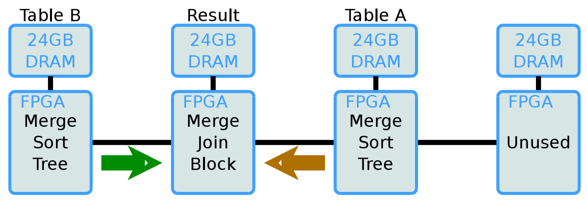

Figure 15: Full multi-FPGA join process. Each table is first sorted separately on the respective FPGA. Finally, both tables are sent to the FPGA containing the merge join block to be merged.

3.4 Sort Merge Join

Finally, we combine the selection, merge join, and sorting blocks to prototype the full design in Figure 8. The resources of a single FPGA were too constrained to fit all three blocks on a single FPGA, so we put the merge join and selection blocks on one FPGA and sort trees on the two adjacent FPGAs. Figure 15 outlines the process used to perform a full join. Each of the columns to be joined is held entirely on a seperate FPGA. Each table is individually sorted, except the output of the sort tree on the final pass is sent across the intra-FPGA links to the merge join block described in Section 3.2. These blocks are sufficiently wide to keep up with the bandwidth of the intra-FPGA links. Since the first sorting pass through the table has a constant throughput limited by the memory bandwidth, and the second and final pass through the data is limited by the intra-FPGA link, the end-to-end throughput of the whole design is a consistent 6.45 GB/s across all table sizes and output cardinality, or just over 800 million key/value pairs a second. This is slightly under the aggregate intra-FPGA bandwidth of 8 GB/s due to the initial pass through the data for sorting. The achieved 6.45 GB/s is 5.6% of the 115.2 GB/s of memory bandwidth available to the three chips. This lower utilization is due to the narrow intra-FPGA links.

If all three blocks were able to fit on a single chip, the second pass through the data would be constrained by the throughput of the merge-join block. In this case, the end-to-end throughput would be 6.25 GB/s, which is lower absolute throughput than the multi-FPGA design due to using only one FPGA's memory bandwidth, but is 16.3% of that FPGA's maximum memory throughput.

Table 2 summarizes our results and compares with other recent work on join processing. In Kim et. al.'s work [7], they used a Core i7 965 with 25.6 GB/s to achieve a join throughput of 128 million 64-bit tuples per second, or 1 GB/s and 3.9% of memory bandwidth. Our multi-FPGA design achieved a 40% increase over this utilization, and a single-chip design would provide a 4.1x increase in utilization. More recent work by Kaldewey et. al. [6] uses a GTX 580 GPU with 192.4 GB/s of memory bandwidth to achieve 4.6 GB/s of aggregate throughput. These results used UVA memory access over a PCIe link since their experiments showed that the computational throughput of the GPU was less then the PCIe data transfer throughput. This, even if the tables were contained in device memory, the join throughput would remain at 4.6 GB/s, or 2.3% of memory bandwidth of the device.

4. RELATED WORK

There has been a growing interest in using dedicated acceleration logic to accelerate database operations, specifically using FPGAs as an excellent platform to explore custom hardware options. Mueller et.al. proposed an FPGA co-processor that performs a streaming median operator which utilizes a sorting network [12]. This work performs a different operation and is directed at much smaller data sets and lower bandwidths than our work. In their design, it was only necessary to have single merge unit that data flowed through, sorting small eight word blocks in a sliding window independent of each other. Our design incorporates a full sorting tree that has many merge units coordinating the sorting of the entire memory stream. This same team has also proposed Glacier, a system which compiles queries directly to a hardware description [11, 13]. This is complimentary to our work as it looks at ways to incorporate accelerators into an overall database system.

Koch and Torrenson also propose an architecture for sorting numbers using FPGAs [8]. The design in this work has similarities to the sorting implementation presented here; however, they were constrained to a system with much lower memory bandwidth and capacity and thus achieve results on the order of 1 to 2 GB/s of throughput. They do not discuss scaling their results to higher bandwidths, which requires fundamental design changes as illustrated in our work. Our work builds on top of this work by presenting new designs that make use of a modern prototyping system with a large amount of memory capacity and bandwidth.

Researchers at IBM proposed an architecture to accelerate database operations in analytical queries using FPGAs [18]. Their work focuses on row decompression and predicate evaluation and concentrates on row based storage. Netezza, now part of IBM, provides systems that use FPGA based query evaluators that sit between disks and the processor [14]. Like Glacier, this work is complimentary and shows the possibilities of incorporating accelerators like those presented here into real database systems.

5. CONCLUSION

In this work we have presented three new hardware designs to perform important primitive database operations: selection, merge join, and sorting. We have shown how these hardware primitives can be combined to perform an equi-join of two database tables entirely in hardware. We described an FPGA based prototype of the designs and discussed challenges faced. We showed that our hardware designs were able to obtain close to ideal utilization of available memory bandwidth, resulting in a 2.8x, 5.7x, and 1.4x improvement in utilization over software for selection, sorting, and joining, respectively. We also present the hardware resources necessary to implement each hardware block and how those hardware resources grow with bandwidth.

6. ACKNOWLEDGEMENTS

This research is supported by DARPA Contract, SEEC: Specialized Extremely Efficient Computing, Contract # HR0011-11-C-0007; Army contract AHPCRC W911NF-07-2-0027-1; NSF grant, SHF: Large: Domain Specific Language Infrastructure for Biological Simulation Software, CCF-1111943; Stanford PPL affiliates program, Pervasive Parallelism Lab: Oracle, AMD, Intel, NVIDIA, and Huawei. Authors also acknowledge additional support from Oracle and the Maxeler University Program.

7. REFERENCES

[1] M. Bauer, H. Cook, and B. Khailany. CudaDMA: optimizing GPU memory bandwidth via warp specialization. In *High Performance Computing, Networking, Storage and Analysis*, SC '11.

[2] S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[3] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *Design Automation Conference*, June 2003.

[4] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proc. VLDB Endow.*, August 2008.

[5] A. A. Chien, A. Snavely, and M. Gahagan. 10x10: A general-purpose architectural approach to heterogeneity and energy efficiency. *Procedia Computer Science*, 4(0):1987 – 1996, 2011.

[6] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU join processing revisited. In *Workshop on Data Management on New Hardware*, DaMoN '12.

[7] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: fast join implementation on modern multi-core CPUs. *Proc. VLDB Endow.*, August 2009.

[8] D. Koch and J. Torresen. FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In *Field Programmable Gate Arrays*, FPGA '11.

[9] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. In *Parallel Distributed Processing*, IPDPS '10.

[10] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream/.

[11] R. Mueller, J. Teubner, and G. Alonso. Glacier: a query-to-hardware compiler. In *Conference on Management of data*, SIGMOD '10.

[12] R. Mueller, J. Teubner, and G. Alonso. Data processing on FPGAs. *Proc. VLDB Endow.*, August 2009.

[13] R. Mueller, J. Teubner, and G. Alonso. Streams on wires: a query compiler for FPGAs. *Proc. VLDB Endow.*, August 2009.

[14] Netezza. The Netezza FAST engines framework.

[15] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Parallel Distributed Processing*, IPDPS '09.

[16] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *Conference on Management of data*, SIGMOD '10.

[17] E. Sintorn and U. Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10), 2008.

[18] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad. Database analytics acceleration using FPGAs. In *Parallel Architectures and Compilation Techniques*, PACT '12.