
ULTRA-SORT

EXTREMELY PARALLEL HARDWARE OPTIMIZED SORTING

15-618 FINAL PROJECT REPORT

Dee Dong

ddong1@andrew.cmu.edu

Saatvik Shah

saatviks@andrew.cmu.edu

1 INTRODUCTION

Just as a recap, as part of our 15618 project, we propose to build a production level DSL-style sorting system efficiently utilizing the hardware its running on. Here is what it should be capable of:

1. Leveraging SIMD parallelism for the latest hardware - optimized for the latest AVX512 registers.
2. Cache friendly
3. Efficient in space and time.
4. Generalized: Support all numeric data types.
5. Generalized: Supporting sorting n-ary tuples by different sort-keys.
6. Bonus goal: Multi-core.
7. Bonus goal: NUMA Awareness.
8. Bonus goal: Peloton Integration.

We are happy to report that apart from the additional bonus goals of *NUMA awareness and Peloton Integration* we have achieved all our primary goals and the bonus goal of multicore SIMD sorting. We are able to achieve a best-cast speedup of 8x on single-core and 17x on multicore AVX512.

The entire implementation is available on Github at <https://github.com/PatwinchIR/ultra-sort>.

The sections have been arranged as follows: Section II provides a literature survey, Section III provides a background and detailed introduction to SIMD and all the components/operations needed within the main algorithm, Section IV covers our main algorithm approach, variants we've setup and the DSL architecture. In Section V, we present our benchmarking results and the corresponding analysis in Section VI. In Section VII we'll conclude our report.

2 LITERATURE REVIEW

Sorting is one of the most important phases in many engineering applications. As part of our project, we have decided to experiment with it keeping in mind the DBMS as an application. There has been a lot of research to exploit the parallelism one can get out of a simple operation as sorting: SIMD intrinsics, multicore and even NUMA-awareness. Prior literatures have some exciting performance results regarding SIMD sorting. C. Kim & Dubey (2009) have suggested that, once hardware provides support for vector instructions that are sufficiently wide (SIMD with 256-bit AVX and wider), sort-merge joins would easily outperform radix-hash joins. C. Balkesen (2013) experimented with AVX2 (256 bit) and found out that radix-hash join is still superior than sort-merge join in many cases, but with the access to wider SIMD this might no longer hold. They also implemented NUMA-aware algorithms in which sort-merge join performed better. H. Inoue (2015) proposed NUMA-aware and cache-friendly sorting algorithm but with AVX2, which also indicates the need for updating the state of art using AVX512.

3 BACKGROUND

3.1 NOTATION

1. **Workload:** A workload refers to the input to our SIMD sorting algorithm. It may consist of numerical, KV pair or record level inputs of different types. The only expectation of our algorithm is that it process such a workload and provide a correctly sorted output.
2. **Scalar vs. SIMD sort:** As expected, a scalar sort is one which does not use any SIMD/vectorized instructions while a SIMD sort does.
3. **Horizontal vs. Vertical SIMD operations:** Horizontal operations apply on a SIMD register on its own, while Vertical operations apply to a pair or group of registers.
4. **Run-size or Chunk size:** A Run or chunk represents a group on numbers which are being operated on. A `unit run size` refers to a minimal such size.
5. **Comparison vs. Movement operations:** A SIMD operation has been referred to as a comparison operation if the actual values within the register are being compared. It is a movement operation if there is horizontal or vertical movement of values independent of the magnitude of values.
6. **Domain of Elements:** By domain we mean the range in which the actual values to sort lie. i.e. We can sort 1024 elements which lie between -10 to 10 or 1024 elements which lie between -10K to 10K. We hope to investigate later how the domain of a workload has an effect on the overall time for the same number of elements.

3.2 USEFUL RESOURCES

In this subsection we have layed out a number of useful resources which we've ended up using throughout our project. This simply serves as a future reference to the authors of this document as well as interested readers.

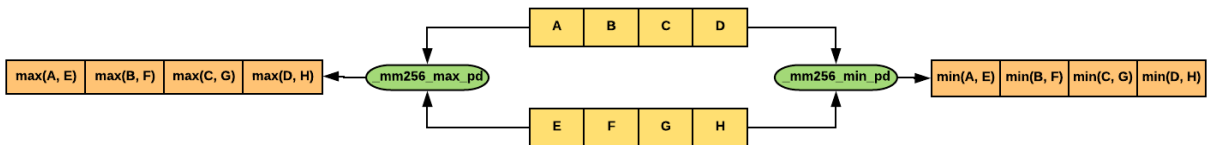
1. [Introduction to Intel Xeon Phi and AVX512](#)
2. [Excellent Introduction to AVX](#)
3. [AVX Intrinsics Guide](#)
4. [Automatic Sorting Network Design](#)

3.3 BASIC SIMD INSTRUCTIONS

We have used a number of AVX intrinsics to achieve relatively complex operations in a vectorized manner. In this subsection we provide insight into some of the building blocks to these more complex operations.

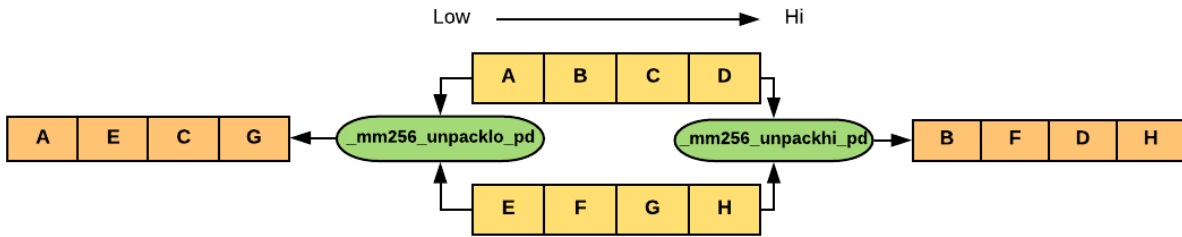
1. **Min/Max:** Directly takes the minimum or maximum and stores into the output register. Below example shows min/max instructions for 4 double precision numbers

Figure 1: Min-Max



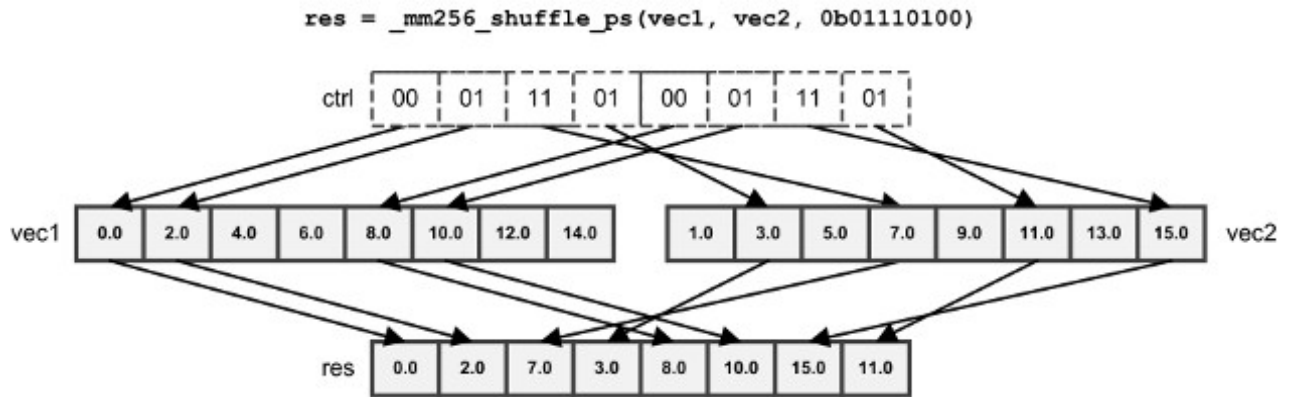
2. **Unpack Lo/Hi:** Unpack-Lo first forms **two groups** of equal numbers, and extracts the lower order numbers from each group sequentially into the output register. Unpack-Hi takes the higher order numbers instead.

Figure 2: Unpack Lo-Hi



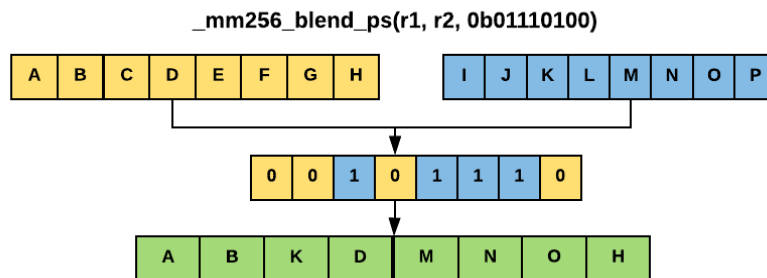
3. **Shuffle:** The control bits are divided into groups of size two(8 bits). Then alternately each group selects elements from the first half of first and second vector, and then from the second vector. i.e. first group selects from first half of vec1, second group from first half of vec2, first group from second half of vec1, second group from second half of vec2.

Figure 3: Shuffle



4. **Blend:** The blend instruction can be used to select one of a specific value from one of the two registers depending on the control signal. Only those number of control bits are used as there are numbers of the specific type in the register.
If we were working with 64-bit double precision numbers then only the first 4 control bits

Figure 4: Blend



would have been used in the above figure.

It's important to keep in mind that all these components have different intrinsic functions for different types and one has to be careful to use the correct type to ensure efficiency and correctness. We'll discuss this more in a later section.

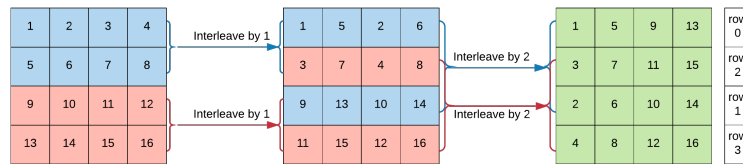
3.4 SORTING NETWORKS

Sorting networks which have been well summarized in Knuth (1997) are one of the best ways to get huge speedups via SIMD. They allow for defining custom sized networks based on the AVX register size and data type available. Once the data has been loaded into registers a set of vectorized min-max operations can be run without branches and with high potential for Instruction Level Parallelism. We used this [handy utility](#) to obtain the most optimal sorting network configuration for different sized networks. An example of the input and output to/from a 4x4 sorting network has been visualized through Figure 7.

3.5 COMPLEX SIMD OPERATIONS

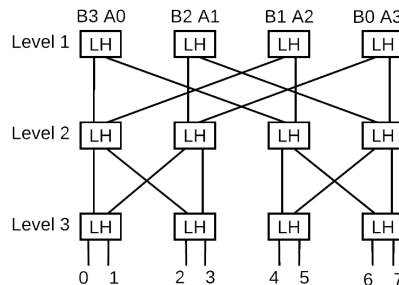
In this subsection we describe more complex operations achieved using SIMD required to efficiently implement our sorting algorithm described later.

Figure 5: 4x4 Transpose Example



1. **Transpose:** The basic idea for transpose a matrix stored in SIMD registers is, suppose N by N matrix:
 - (a) Interleave N rows by Unit 1. (32 bits for `int`, 64 bits for `int64_t`).
 - (b) Interleave N rows by Unit 2. (64 bits for `int`, 128 bits for `int64_t`).
 - ...
 - (c) Interleave N rows by $N/2$.
 the transpose can be done in $N \log_2 N$ operations

Figure 6: Bitonic Merge Network



2. **Horizontal Intra-Register Sort:** The idea behind intra-register sort is bitonic merging network. For a merging network, two sorted sequences of length K each are fed as inputs, and a sorted sequence of length $2K$ is produced. Figure ?? shows the idea of merging 2 sequences of 4 elements. Initially A and B are held in SIMD register and are both sorted

in the ascending order. Bitonic merge requires one of the sequences sorted in descending order, so a reverse of B is done on register. Each level of the network use **Min/Max** to do comparison, after this the data in register is permuted for the appropriate comparison for the next level. Figure ??(in Appendix) showed an example of merging two 16-value registers using SIMD instructions. With Blue box indicating smaller values to Red box indicating larger values, shallow color boxes are redundant duplicated values resulted from permutation. The register on the right is already flipped, the output is a sorted 32-value sequence stored in two registers respectively, with the smallest 16 values in the left register and the largest 16 values in the right register. The steps are also shown in the example to indicate the dependencies. Steps which don't have any dependencies can be executed parallelly.

4 APPROACH

In this section we will describe the main approach including the algorithms used in each step, its variants and along the way we motivated the need for a **DSL for SIMD Sorting**.

4.1 ALGORITHMS

We summarize the steps of sorting an N elements sequence here, in the subsections we will introduce our building blocks with two phases. To sort N elements, for example, consider N `int` values, each value is of 32 bits, in AVX512, one register can represent 16 such values. We pack 16 `int` values into one register, and prepare 16 such registers to form a $16 \times 16 = 256$ values, we call this a block, and then apply a 16 by 16 sorting network to this block to get 16 column-wise(vertically) sorted registers. Once we transpose these 16 registers, we can get the new 16 registers in which the 16 values are sorted. This process is done repeatedly on the N elements, 256 values at a time, with every 16 values sorted.

Next, to get a completely sorted array, a merging process is performed for those 16-value sorted sequences. This process is highly iterative, for example, the first iteration for the example above would be merging every two 16-value sorted sequences to get a 32-value sorted sequence. Next iteration would be merging every two 32-value sorted sequences to a 64-value sorted sequence, algorithm used for using 16-value register to merge 32-value sequence is detailed in the subsection. Such and so on until the whole sequence is merged. This process requires N to be the power of 2.

4.1.1 PHASE 1: BUILDING SORTED RUNS

The first step in our algorithm is to obtain sorted runs of fixed size on which the next phase will operate. In order to obtain such sorted runs we can apply the following steps:

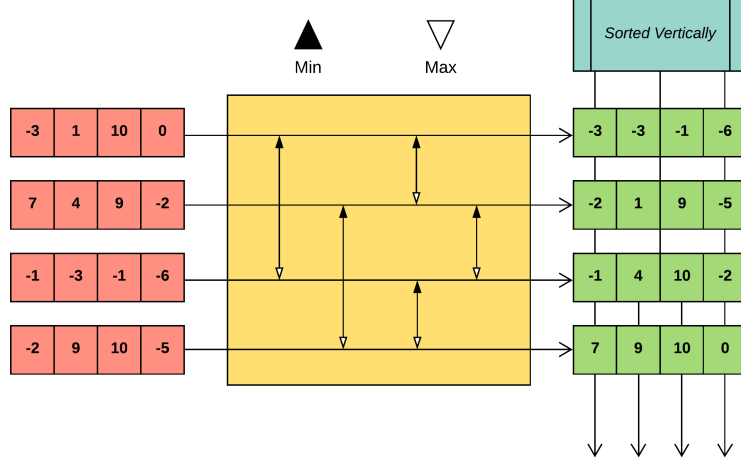
1. Based on the data type to sort, a custom best-case sorting network has to be selected(For example for `Int32` on AVX2, we can select a network of 8×8 in the best case.) We then apply this sorting network on a block of numbers(eg. 64 in the example from above). This has been visualized in Figure 7.
2. When applied on a block of numbers the sorting network returns the numbers sorted but between same positions of different registers - i.e. vertically not horizontally sorted. Thus we need a Transpose operation - This has again been implemented as described in Section III. It's again important to keep in mind that depending on the sorting network size the correct transpose operation should be performed.

At this stage we have a huge array with sorted runs of `unit run size`(which in the example above is 4).

4.1.2 PHASE 2: MERGING SORTED RUNS

After completing phase 1 we will have an array with sorted runs. The next step would be to efficiently merge sorted runs. In the explanations that follow, `unit run size` refers to the unit length on which an operation can apply. For example the `BitonicMerge8` merges 2 registers of

Figure 7: Sorting Network In/Out Example



size 8, so the `unit_run_size` is 8. The algorithm we follow is similar to merge-sort and has the following building blocks:

1. **Merge-Run-Pass Block:** In this block, the `BitonicMerge` which we defined earlier is used as a kernel on runs of variable size (multiple of the `unit_run_size`). A high level overview of this operation is available in figure 8 and has been described in H. Inoue (2015). In the figure, we use the `BitonicMerge4` as a kernel. We start by loading two unit size runs into registers. Then at each step we apply the bitonic merge kernel, and store the smaller output to an output buffer. We then load the next sorted run in. This is continuously done until all the runs have been processed. In practice this is implemented slightly differently:
 - (a) Split the input set of runs into two streams A and B.
 - (b) Load R_a and R_b with the first run from A and B.
 - (c) If $R_a[0]$ is less than $R_b[0]$ we select the next run to be loaded from A, else B.
 - (d) Apply bitonic merge and store the smaller half out and larger half in R_b .
 - (e) Load the next run in (from (c)).
 - (f) Go to (c). Continue until either of the streams has been completely processed. Then just merge in any remaining portions of the stream still having unprocessed runs.

The final output has chunks of merged/sorted runs of $2 * \text{run_size}$.
2. **End-To-End Merging:** We can now treat the `Merge-Run-Pass-Block` as a kernel itself to get finally merged runs. This is done by starting to merge `unit_run_size` of chunks of sorted runs and doubling run sizes to be merged from there onwards. Thus $\log(\text{seq_length})$ passes would be needed to complete the entire sorting procedure. This has been demonstrated in Figure 8 and has been derived from the work of C. Kim & Dubey (2009). For example if we used a sorting network of size 8×8 we will have a `unit_run_size` of 8. In the first pass we will merge 2 chunks of size 8 to merged-sorted runs of size 16. In the second pass we would merge chunks of length 32. This would go on till we reach the length of the sequence.

4.2 SORTING KEY-VALUE PAIRS

A widely used application of sorting algorithms is to sort key-value pairs. In this subsection we addressed the methodology we used to achieve this objective. When we say "key-value sort", we mean that we want to sort the values by the key. Following are the key ideas:

1. **Keep Key-Value Together:** Amongst the papers we read as part of our Literature survey, we only saw Furtak et al. (2007) which has specifically proposed an algorithm for KV

Figure 8: Merging Sorted Chunks

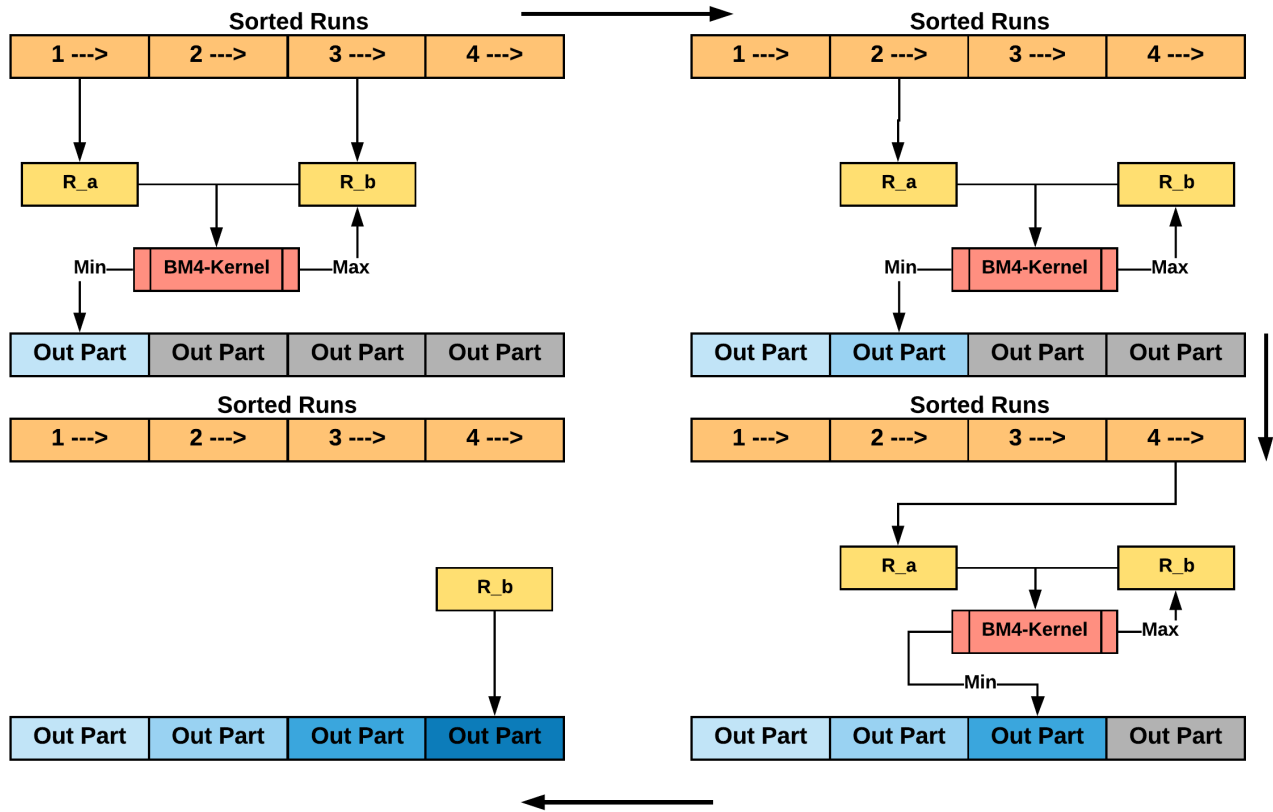
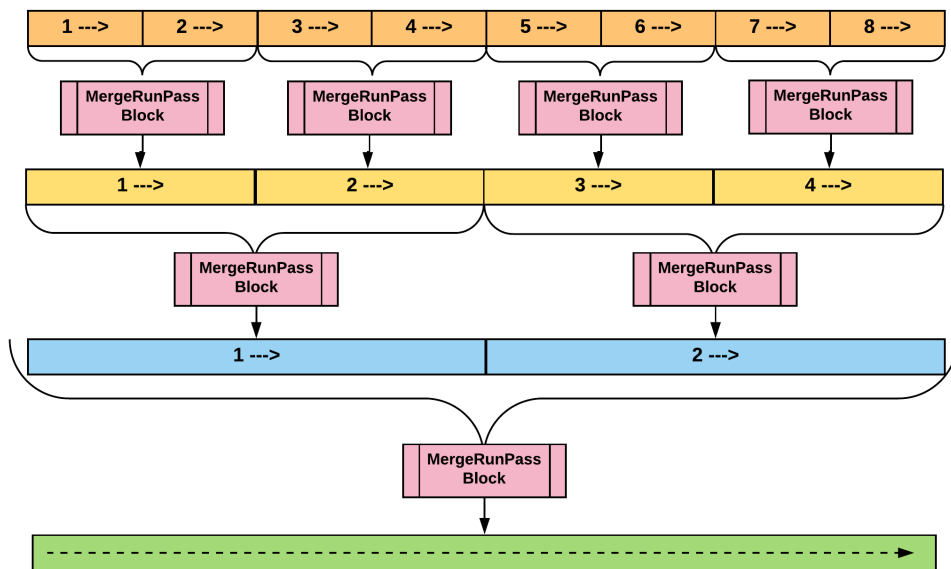


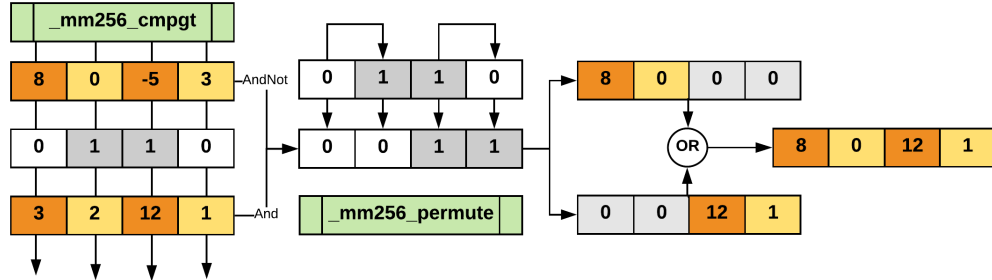
Figure 9: End-To-End Merging



pair sorting. However he recommends keeping the key-value pairs in separate registers - working on the keys and mimicing the same movements on the values. Such a design allows for little code reuse and would need complete rewriting of all the components for KV sorting. We instead propose a KV sorting design in which KV pairs are packed into the same SIMD register contiguously. Once this is done it can be noted that majority of the original code(especially the utility operations described in Section III) and design can be reused with carefully masking any comparison operations. It's important to note that the effect of doing a movement operation should be the same on registers holding different types within them since movements happen across lanes of 32, 64 or more bits without caring about values. It is at the same time important to note however that performing a movement operation with the intrinsic of correct type has much lower latency in some cases compared to intrinsics of a different type.

2. **Masked Min-Max:** The procedure to perform masked comparison operations to perform the original MinMax operations while copying the movements of the value with the key are summarized in Figure 10. Once we have a `MaskedMinMax` we can begin to construct masked versions of all other operations by simply replacing the Min-Max with the Masked Min-Max variants. Additionally this is only required in steps where there is any sort of comparison.
3. **Adjust Movement operations:** We also need to replace the original movement operations on units of same run size with operations working on double the lane size and half the data. eg. Transpose 4x4 on 64-bit values to Transpose 2x2 with 128-bit values when sorting in Phase 1.

Figure 10: Masked Min-Max



4.3 "ORDER BY" IN DBMS

The operation of sorting key-value pairs can be used as a lower-level API to perform a Database order by(albeit limited to 1 column - We weren't able to come up with an efficient generalized algorithm for multicolumn order by.) This can be done by the following steps:

1. **Convert to KV pairs:** The specific column on which we want to `Order By` goes into the Key field and we assign the Record-Id or 0-offset in the value.
2. **KV Pair Sort:** Now apply the algorithm described in previous sections for KV sorting.
3. **Rearrange Records:** Once we have our sorted KV pairs we can use the 0-offsets to rearrange records to their correct positions. Its important to note that this step will have a huge penalty as it will involve a large number of random accesses throughout main-memory. H. Inoue (2015) has proposed a variation of this step where we perform such rearrangements incrementally while running the algorithm. This is a possible future work for us.

4.4 PARALLELIZING SIMD SORTING

We used OpenMP to try and parallelize our code. C. Kim & Dubey (2009) have done an excellent job in this aspect by allowing for parallelized merge pass, multiway merges and carefully distributed

data placement, we attempted to parallelize Phase 1 of the algorithm and the merge pass for lack of time. Following is our methodology and inferences:

1. **Parallelizing creation of Sorted Runs:** The sorted runs are simple to parallelize since they involve independent application of the Bitonic Sorting network on chunks of fixed size. Unfortunately this step is highly bandwidth bound. Thus instead of seeing a speedup with more threads we saw a drop in speed because of the additional overhead of fork-join and task assignment with no gains in terms of latency due to being bandwidth bound.
2. **Parallelizing Merge Pass:** As can be seen in Figure 9, the merge pass involves application of the `MergeRunPass` kernel on chunks of equal size. This can be parallelized as long as we ensure that all relevant variables and the buffer offset is correctly setup. When we applied this, we did find a speedup, but this was sub-linear. This again is because there is a lot of load-store operations from arrays into registers and vice versa during operations.

4.5 DSL SETUP AND GENERALIZING SIMD SORTING

As we've seen in the above subsections, there is a strong motivation to build a DSL for SIMD sorting:

1. Between register types the nature of supported functions widely varies. Thus what is most optimal strongly depends on the sorting problem and hardware in question.
2. Different CPUs support different types of registers: SSE4, AVX, AVX2, AVX512. We always want to use the most optimal register type available on the hardware in question.
3. Different functions within a register for different data types: Thus most available implementations only support integer data types. However this is not true of real world workloads which have varying types.

Thus, we propose **UltraSort** as a DSL for SIMD sorting which:

- Work across different hardware architectures(supporting and optimizing differently for both AVX2 and AVX512.)
- Support most optimized variants for every numeric data type. Based on the data type used the most optimal sorting configuration possible is picked.
- Support key-value pair sorting for any numeric data type.
- Supports record-level sorting based on a single "key".
- Parallel execution(currently in a multicore non-distributed manner).

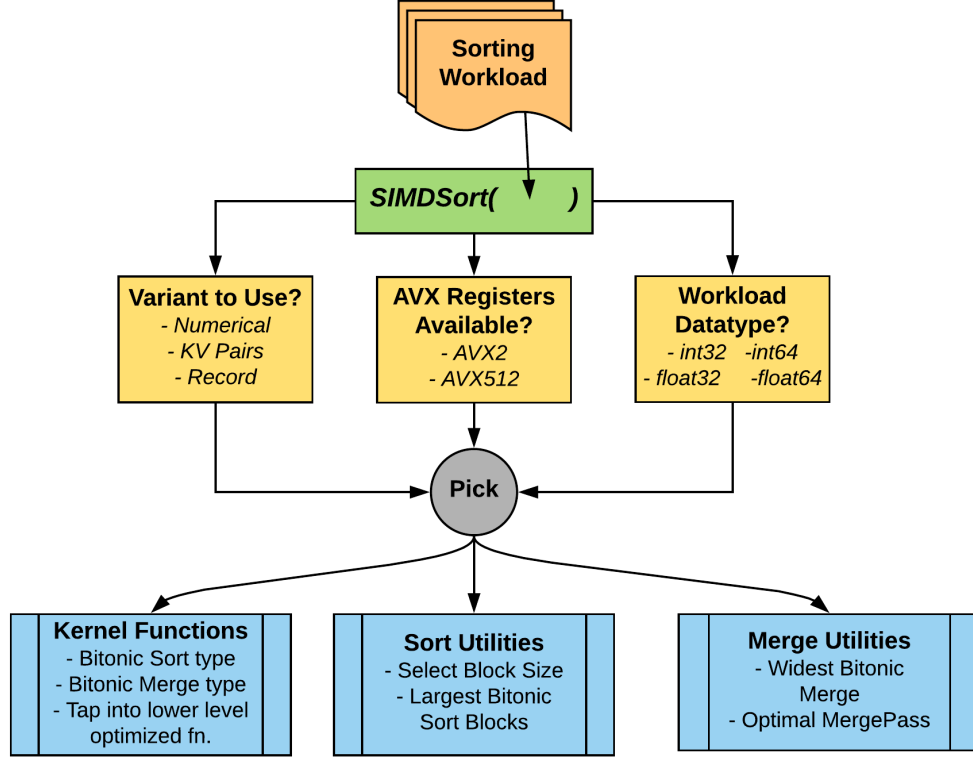
As can be seen in Figure 11 our DSL framework chooses the optimal SIMD sorting variant based on different characteristics of the workload and hardware. At a higher level is the selection of the optimal kernels and then at a lower level the correct set of functions(by data type) for the implementation of the kernel. For example in the case of Bitonic sorting for int32 on AVX512 we can have a 16x16 bitonic sorting network possible but only 8x8 on AVX2. Further the implementation varies since there are different function signatures and behaviors on both architectures.

Wherever possible we've tested for and templated functions and kernels, performing register type-casting only where there were no penalties(`_mm_cast` has zero latency but certain movement operations on a different data type than expected incur additional overheads). This also allowed us to have extensive code reuse.

4.6 TESTING AND CORRECTNESS

We've tested every component of the entire framework from lower level functions to higher level kernels to the entire sort correctness within the **GTest testing framework**. This allowed us to confidently play around with changes without worrying about breaking anything. This was extremely crucial given that our complete project stands at **4000** lines of code.(Just FYI our tests also cover **4000** lines of code).

Figure 11: DSL Framework for SIMD Sorting



5 BENCHMARKING & RESULTS

5.1 EXPERIMENTS MACHINE STATISTICS

The machine we used to obtain all our experiments results are of the following configurations(information listed by executing `lscpu`):

1. Hardware: Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz. 40 CPUs(2 Threads per core). L3 Cache: 14080K.
2. Software: Compile with flag `-O3 -flto`.

5.2 BENCHMARKING

We show exact benchmarks for all settings listed in our proposal, with sequential SIMD. We have also obtained results for multicore settings, but for sake of space have omitted the entire table of results. Instead in the Analysis section we have analyzed the general trend followed compared to the sequential algorithm in Figure 22. We can confirm that this trend generalizes to other settings(KV pairs, Order By).

Due to large experiments results, in this section, representative results would be presented. The full timing and speedup results can be found at Section 8. Table 4, 1, 2 showed the time in seconds, ipc and speedup of those different algorithms took to sort $1M(2^{20})$ elements of different settings.

Table 1: 1M Elements Sort IPC(Instructions Per Cycle)

Settings/Algorithms	std::stable_sort	std::sort	ips4o::sort	pdqsort	avx256::sort	avx512::sort
32BitInteger	1.506790	0.681694	2.076354	2.336017	2.152515	1.630018
32BitFloat	1.197460	0.613799	1.823862	2.090052	1.478475	1.060270
64BitInteger	1.457014	0.670313	1.999275	2.285352	1.416596	1.157502
64BitFloat	1.169164	0.623366	1.794311	2.050244	1.201110	1.065701
32BitKeyValueInt	0.961099	0.789943	2.395018	0.612214	1.292443	1.060222
64BitKeyValueInt	0.900830	0.767878	2.345641	0.602405	1.295691	1.335133
32BitKeyValueFloat	0.861952	0.697731	2.085952	0.581825	1.066630	1.504925
64BitKeyValueFloat	0.797310	0.683473	2.090357	0.581601	1.194682	1.348604

Table 2: 1M Elements Sort Speedup(Relative to std::sort)

Settings/Algorithms	std::stable_sort	std::sort	ips4o::sort	pdqsort	avx256::sort	avx512::sort
32BitInteger	1.254256	1.000000	2.263849	2.223261	4.200139	7.741558
32BitFloat	1.116557	1.000000	2.375618	2.187715	3.594582	4.471524
64BitInteger	1.258417	1.000000	2.220918	2.266862	1.484623	2.931080
64BitFloat	1.102980	1.000000	2.284896	2.201074	2.043218	2.543850
32BitKeyValueInt	0.832961	1.000000	2.246350	0.993639	1.230044	2.458543
64BitKeyValueInt	0.739063	1.000000	1.927807	1.020069	0.773838	0.794888
32BitKeyValueFloat	0.855136	1.000000	2.273620	0.995006	1.302896	1.567567
64BitKeyValueFloat	0.775070	1.000000	2.004722	1.003840	0.856764	0.926604

Table 5, 6, 7 showed the time in seconds, ipc and speedup of those different algorithms took to sort 1024 elements of different settings.

Table 3: AVX OrderBy Performance Comparison with 32KV

# Elements	Vector Width	32-Bit Key-Value	Order By
2^{10} (1024)	256	0.00011110	0.00010232
	512	0.00002439	0.00001828
2^{20} (1M)	256	0.02007687	0.02899678
	512	0.03365534	0.04621883
2^{24} (10M)	256	1.33803573	1.40597913
	512	0.73420464	0.91561371

We also tested using more than 1 threads in AVX512 to experiment. Figure 12 shows the trend of time taken for `avx512::sort` under different number of threads setting, N is 10 million `int` data. The parallelism we took is for the phase 2, using **OpenMP** to parallelize the merge pass process.

6 ANALYSIS

6.1 SEQUENTIAL PERFORMANCE

To better view the statistics we collected in Section 5, we plotted them out. Since AVX256 version is for reference so only AVX512 statistics are plotted.

We can clearly see from Figure 13 that for sorting 1024 elements, `avx512::sort` beats all other sorting algorithms across all different settings. Due to native instructional support for 32 bits `int` and `float` value, it achieved the best timing. Compared with sorting 1M elements, `avx512::sort` suffered a lot from Key-Value pair sorting, the reason behind this is directly from the lack of natively supported instructions. For `int64_t` Key-Value pair(128 bits in total), a mask is necessary for the Min/Max operations and other operations that involve a lot of data movement(shuffling, permutation, blending, etc). Instead using a single instructions with latency 1, to ensure correct implementation, we have to use multiple instructions to achieve the desired functions. For example, to implement Min/Max of 64 bit Integer values, we have to convert those

Figure 12: Number of threads and Time in seconds of `avx512::sort`

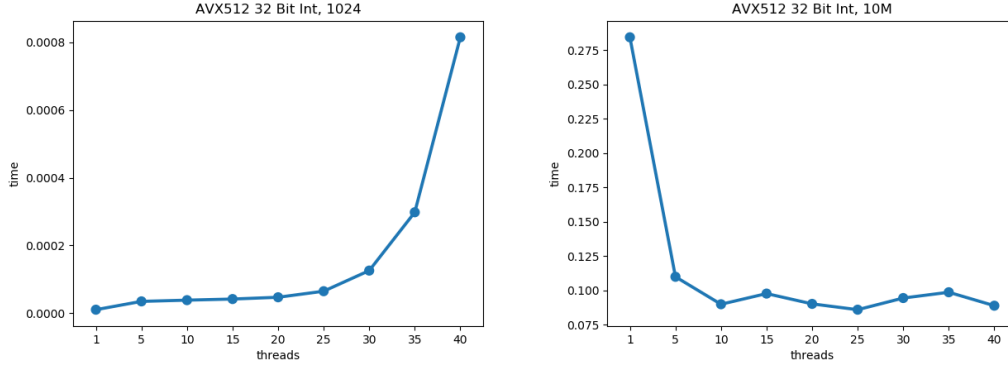
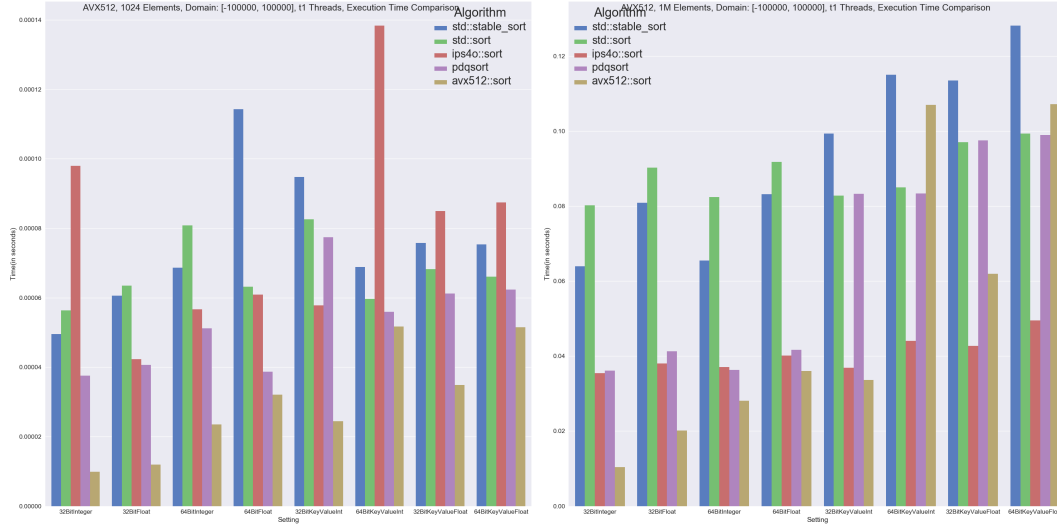


Figure 13: Algorithms Time(in seconds) Comparison on Different Settings, 1024 Elements on the Left, 1M on the Right



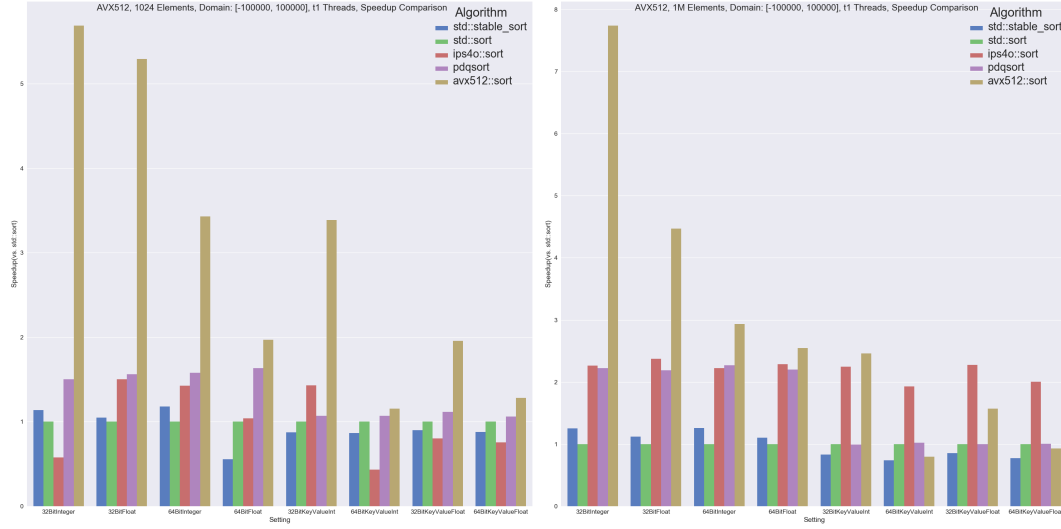
`int64_t` values to double values then use supported double precision instructions.

The timing results for 64 bits integer values and 32 bits Key-Value pairs are around the same, this is expected since we're using 64 bits integer value to sort a packed 32-bit integer key, 32-bit integer value. The same logic can't be applied to packed 32-bit float key, 32-bit float value considering the much more complex representations double precision float number takes in computer, otherwise the actual value would be lost and Min/Max operations would fail the correctness check. In this case, masked Min/Max come into play.

Speedup plot against `std::sort` from Figure 14 gives a clearer view about how the performance is for each algorithm. `avx512::sort` achieved over 5x speedup against `std::sort` for sorting 1024 elements, but achieved near 8x speedup when sorting 1M elements which is near the expected speedup compared with literatures that using AVX2 to achieve near 4x speedup against `std::sort` (AVX512 is double width of AVX2). The reason behind sorting 1024 only achieved 5x could be that `std::sort` does particular optimizations to sort sequence of smaller number of elements to get a better performance, we can get this conclusion from the timing plot.

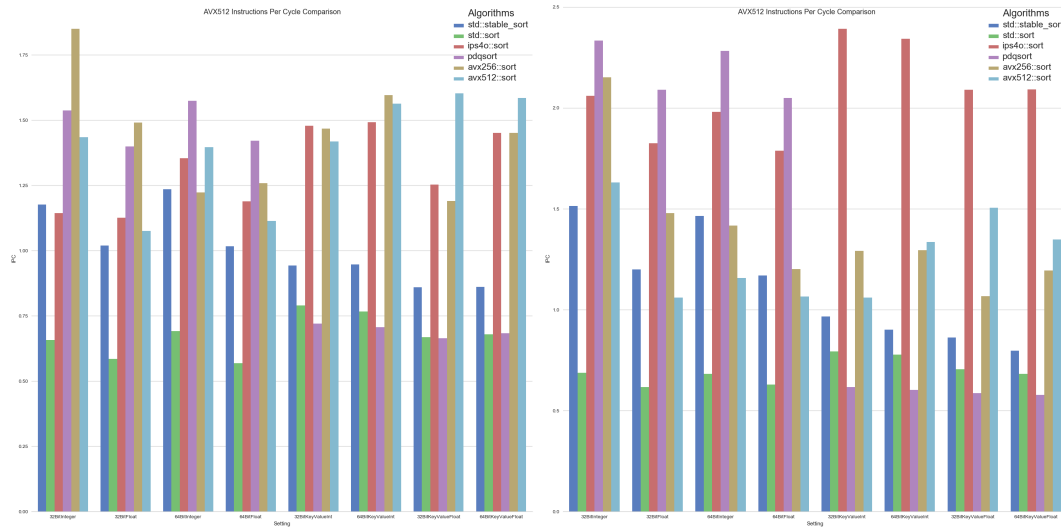
We also produced IPC results(Figure 15) corresponding to Figure 13 to get a better sense of how well our SIMD sorting system performed from a different perspective. We can see that correspond-

Figure 14: Algorithms Speedup Comparison on Different Settings, 1024 Elements on the Left, 1M on the Right



ingly from the timing statistics, `avx256::sort` and `avx512::sort` achieved very high IPC against other algorithms when the number of elements to sort is fairly small(1024), they are consistently getting around 1.5 IPC across different settings, this is also expected because of the whole working set fits into cache and everything happens without interacting with main memory. But this is not the scenario when 1 million numbers need to be sorted, the IPC plot showed that with larger size value being involved(from 32 bits value to 128 bits pair), the IPC of `avx256::sort` and `avx512::sort` keeps decreasing. The machine we ran experiments on has an L3 cache size of 14080k bytes, which can hold around 3 million integers, but can only hold 880000 128-bit values which is a little more than half of 2^{20} elements. So a lot of memory latency is exposed when doing large value SIMD operations, especially when the operations are not directly supported.

Figure 15: Algorithms IPC Comparison on Different Settings, 1024 Elements on the Left, 1M on the Right



In conclusion, `avx512::sort` has a lot of speed advantages over other algorithms even when the number of elements is large, as long as the sequence to be sorted is a primitive numeric type instead of a pair of key and values. The key and value pair sorting system performed rather poor

against `ips40::sort` due to our carefully crafted masked implementation because of the lack of originally supported instructions. In comparison, though `ips40::sort` didn't beat our sorting system when the number of elements to sort is small, it performed fairly stable across different settings in terms of Speedup and IPC, even when the number of elements varies. So under different application scenarios on different platforms, different algorithms should be applied to achieve the best performance.

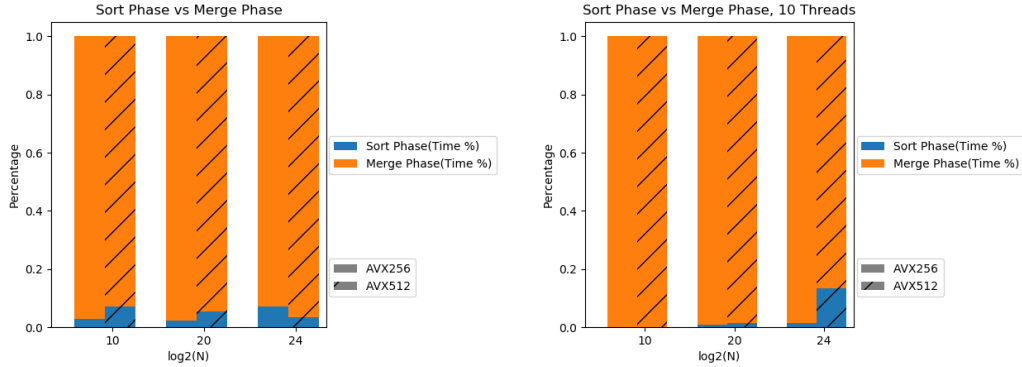
6.2 OPENMP PARALLELISM

As explained in Section 4.4, we applied parallelism on our merge phase. Figure 12 showed the trends of `avx512::sort` time taken for sorting 1024 and 1M elements changing with the number of threads used. We can see from those two plots that the overhead of spawning threads becomes obvious when the number of elements to sort is small. Arithmetic intensity in this case is too small compared to sorting 10M elements. With 10M elements we can observe a sharp speedup of `avx512::sort` by benefiting from multi-threads execution. This is because the merge phase is compute bound. With the number of threads continues to increase, the speedup is not optimal anymore as the merging process becomes bandwidth bound (and the amount of parallelism is limited in the last few merge passes). The speedup reached approximately the maximum when the number of threads is around 10.

6.3 ALGORITHM TIME BREAKDOWN ANALYSIS

As explained in Section 4.1 that our main algorithm contains 2 phases: sort phase and merge phase. To better understand how each phase performed obtained the timing for each phase running using 1 thread. Figure 16 showed the percentage breakdown for sorting 32 bits integer values at 3 different scales. We can clearly see that most of the time went to the merging phase, it remains the same for both `avx256::sort` and `avx512::sort` with a small difference.

Figure 16: Sort vs. Merge Phase Time Percentage, 1 Thread on the Left, 10 Threads on the Right



Comparing the results running with 1 thread versus running with 10 threads, we can see that `avx512::sort` merge phase time percentage dropped when sorting 2^{24} elements. The effect of threads to this percentage results is not clear of sorting smaller number of elements as explained in Section 6.2. Merge phase simply contributed all the time to the whole sorting process due to the large overhead of OMP threads applied in the merge phase.

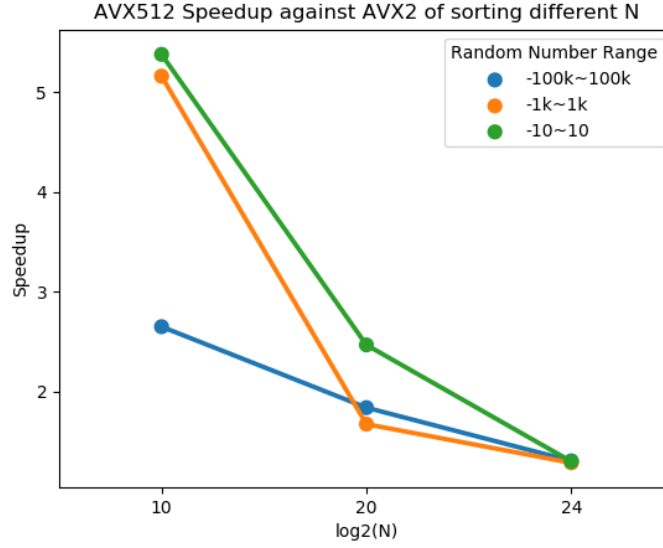
6.4 AVX2 & AVX512

Figure 17 showed the speedup of `avx512::sort` against `avx256::sort`. As the number of elements to sort increase, the speedup from `avx512::sort` showed a linear decrease. When sorting smaller number of elements, AVX512 can achieve between 2.6x-5.2x over AVX2 depending on the domain of numbers. As the number of elements increases however the speedup uniformly decreases to around 1.5x AVX512 over AVX2.

6.5 ELEMENT DOMAIN ANALYSIS

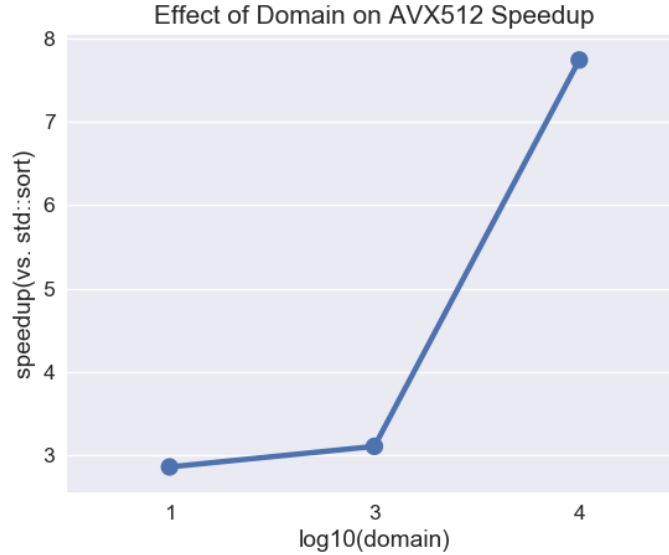
In this section we analyze the effect of using different domains with the same number of elements. The speedup against `std::sort` with 1M elements on a domain between 10 to 10000 have been vi-

Figure 17: AVX512 Speedup against AVX2 of sorting different N



sualized in Figure 18. It is clearly visible that as the size of our domain increases, the speedup of AVX512 over `std::sort` scales up. The reason for this is that `std::sort` begins slowing down considerably as the domain becomes larger. We can thus conclude that AVX512 is more viable and stable when working with numbers in a larger domain.

Figure 18: Effect of the Domain of elements on AVX sorting speedup(vs. `std::sort`)

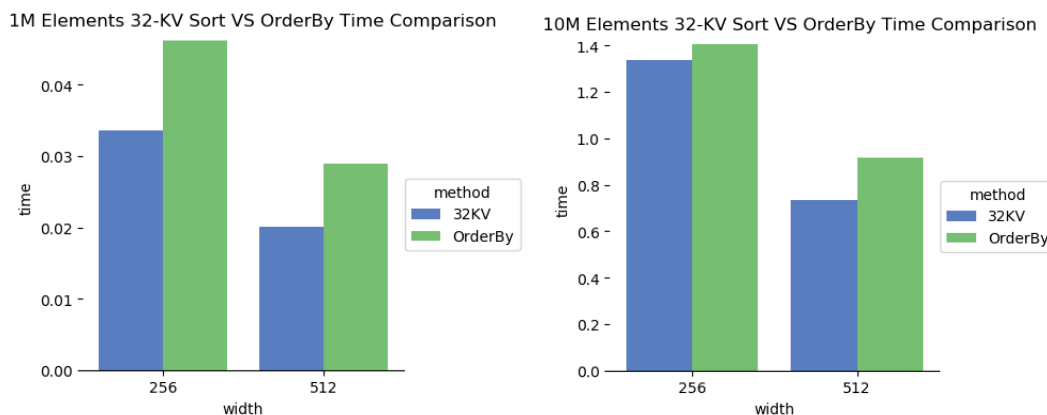


6.6 SIMD ORDER BY(RECORD-LEVEL SORTING)

In this section we analyze the results of SIMD record level sorting and deduce the overhead of random accesses it incurs over KV-pair sorting. Its important to keep in mind that record-level sorting simply packs the sort key and an `offset/index` as a value into a KV pair. It then applies KV sorting and rearranges records by the order of `offset/index`.

We have visualized our numbers in Figure 19. The overhead between the two is that of setting up the `offset/index` and rearranging records by random access. We have only used Order-By on 1 key/column. The overhead is quite significant and was also observed by H. Inoue (2015) who suggests a solution involving record rearrangement at each step of the merge pass.

Figure 19: KV Sort vs. Order-By with AVX512



6.7 NOTES ON SIMD

In this subsection we provide some suggestions for ourselves and other readers regarding SIMD in general which we found during experimentation and development:

1. **Missing Operations:** One should not expect standard SIMD intrinsics to be available for all types. For example `min/max` on AVX2 is available for all types except `int64`. On the other hand `min/max` is available for all types on AVX512. That said most missing operations can be recreated albeit with the penalty of more latency.
2. **Movement operations are type oblivious:** One can perform shuffles, blends, permutes or similar movement operations meant for float registers on an `int32` register or vice versa. Same is true for `int64` and double. The reason for this is that movement operations don't involve comparison.
3. **Cross-Type Blend Movements have penalties, Shuffles and Permutes don't:** Following from the above point, while it might be possible to do movement operations correctly between cross-types, blend operations end up having significant penalties when operating between types. [Additionally performing float operations on int types has little to no penalty than the other way around.](#) This is why all templated functions involve typecasting register types to float types to operate on.

6.8 STARTER CODE AVAILABLE

We referred to the following code on the internet for help on understanding how to implement correct/optimized versions of certain utility functions:

1. [C. Balkesen \(2013\)](#)
2. [Github AVX2 32-bit int sorting](#)

7 CONCLUSION

As part of our course project we've primarily achieved two goals:

1. Creating a first of its kind DSL-style SIMD sorting architecture supporting both AVX2 and AVX512, multiple types, KV and record-level sorting.
2. Analysis and visualization of different aspects of the algorithm and workload to find the sweet spot where SIMD makes sense.

REFERENCES

- J. Teubner M. Tamer Ozsu C. Balkesen, G. Alonso. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*(1), 7, 2013.
- J. Chhugani T. Kaldewey A. D. Nguyen A. D. Blas V. W. Lee N. Satish C. Kim, E. Sedlar and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *PVLDB*2(2), pp. 1378–1389, 2009.
- Timothy Furtak, José Nelson Amaral, and Robert Niewiadomski. Using simd registers and instructions to enable instruction-level parallelism in sorting algorithms. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pp. 348–357. ACM, 2007.
- K. Taura H. Inoue. Simd- and cache-friendly algorithm for sorting an array of structures. *PVLDB*, 8, 2015.
- Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

8 APPENDIX

8.1 DETAILED RESULTS

Table 4: 1M Elements Sort Time(in seconds)

Settings/Algorithms	std::stable_sort	std::sort	ips4o::sort	pdqsort	avx256::sort	avx512::sort
32BitInteger	0.063970	0.080234	0.035442	0.036089	0.019070	0.010364
32BitFloat	0.080819	0.090239	0.037985	0.041248	0.024958	0.020181
64BitInteger	0.065461	0.082378	0.037092	0.036340	0.028105	0.028105
64BitFloat	0.083170	0.091735	0.040148	0.041677	0.045282	0.036061
32BitKeyValueInt	0.099336	0.082743	0.036834	0.083273	0.068085	0.033655
64BitKeyValueInt	0.115010	0.085000	0.044091	0.083327	0.109607	0.106933
32BitKeyValueFloat	0.113443	0.097009	0.042667	0.097496	0.075137	0.061885
64BitKeyValueFloat	0.128100	0.099286	0.049526	0.098907	0.116892	0.107151

Table 5: 1024 Elements Sort Time(in seconds)

Settings/Algorithms	std::stable_sort	std::sort	ips4o::sort	pdqsort	avx256::sort	avx512::sort
32BitInteger	0.000050	0.000056	0.000098	0.000038	0.000026	0.000010
32BitFloat	0.000061	0.000063	0.000042	0.000041	0.000029	0.000012
64BitInteger	0.000069	0.000081	0.000057	0.000051	0.000047	0.000024
64BitFloat	0.000114	0.000063	0.000061	0.000039	0.000045	0.000032
32BitKeyValueInt	0.000095	0.000083	0.000058	0.000077	0.000056	0.000024
64BitKeyValueInt	0.000069	0.000060	0.000138	0.000056	0.000056	0.000052
32BitKeyValueFloat	0.000076	0.000068	0.000085	0.000061	0.000044	0.000035
64BitKeyValueFloat	0.000075	0.000066	0.000087	0.000062	0.000062	0.000052

Table 6: 1024 Elements Sort IPC(Instructions Per Cycle)

Settings/Algorithms	std::stable_sort	std::sort	ips4o::sort	pdqsort	avx256::sort	avx512::sort
32BitInteger	1.506790	0.681694	2.076354	2.336017	2.152515	1.630018
32BitFloat	1.197460	0.613799	1.823862	2.090052	1.478475	1.060270
64BitInteger	1.457014	0.670313	1.999275	2.285352	1.416596	1.157502
64BitFloat	1.169164	0.623366	1.794311	2.050244	1.201110	1.065701
32BitKeyValueInt	0.961099	0.789943	2.395018	0.612214	1.292443	1.060222
64BitKeyValueInt	0.900830	0.767878	2.345641	0.602405	1.295691	1.335133
32BitKeyValueFloat	0.861952	0.697731	2.085952	0.581825	1.066630	1.504925
64BitKeyValueFloat	0.797310	0.683473	2.090357	0.581601	1.194682	1.348604

Table 7: 1024 Elements Sort Speedup(Relative to std::sort)

Settings/Algorithms	std::stable_sort	std::sort	ips4o::sort	pdqsort	avx256::sort	avx512::sort
32BitInteger	1.137291	1.000000	0.575207	1.500533	3.004192	5.689899
32BitFloat	1.047894	1.000000	1.500710	1.561654	2.191511	5.296327
64BitInteger	1.178717	1.000000	1.426102	1.579914	1.693648	3.432088
64BitFloat	0.553212	1.000000	1.037420	1.633755	1.464924	1.967933
32BitKeyValueInt	0.872149	1.000000	1.428250	1.067589	1.470357	3.387044
64BitKeyValueInt	0.866976	1.000000	0.431514	1.067979	1.055197	1.154962
32BitKeyValueFloat	0.900092	1.000000	0.802920	1.113469	1.478458	1.955836
64BitKeyValueFloat	0.876327	1.000000	0.755606	1.059182	1.061596	1.281335

Table 8: AVX256 10M(2^{24}) Elements Sort

# Elements	Setting	Algorithm	Time(in seconds)	Speedup(vs. std::sort)
10M	32BitInteger	std::stable_sort	1.222261	0.458637
		std::sort	0.560575	1.000000
		ips4o::sort	0.075775	7.397907
		pdqsort	0.128691	4.355985
		avx256::sort	0.371563	1.508691
	32BitFloat	std::stable_sort	1.570623	1.114942
		std::sort	1.751154	1.000000
		ips4o::sort	0.666538	2.627237
		pdqsort	0.749901	2.335179
		avx256::sort	0.506288	3.458809
	64BitInteger	std::stable_sort	1.301385	0.414187
		std::sort	0.539017	1.000000
		ips4o::sort	0.086897	6.202964
		pdqsort	0.135643	3.973785
		avx256::sort	1.107240	0.486812
	64BitFloat	std::stable_sort	1.654480	1.073045
		std::sort	1.775331	1.000000
		ips4o::sort	0.707242	2.510218
		pdqsort	0.778227	2.281250
		avx256::sort	0.911587	1.947517
	32BitKeyValueInt	std::stable_sort	1.095975	0.544410
		std::sort	0.596660	1.000000
		ips4o::sort	0.097868	6.096571
		pdqsort	0.333106	1.791203
		avx256::sort	1.338036	0.445922
	64BitKeyValueInt	std::stable_sort	1.579650	0.413890
		std::sort	0.653802	1.000000
		ips4o::sort	0.134097	4.875572
		pdqsort	0.374753	1.744619
		avx256::sort	1.981982	0.329873
	32BitKeyValueFloat	std::stable_sort	2.310919	0.819373
		std::sort	1.893506	1.000000
		ips4o::sort	0.759013	2.494696
		pdqsort	1.903081	0.994969
		avx256::sort	1.577517	1.200308
	64BitKeyValueFloat	std::stable_sort	2.516338	0.769152
		std::sort	1.935447	1.000000
		ips4o::sort	0.895927	2.160273
		pdqsort	1.911992	1.012267
		avx256::sort	2.231243	0.867430

Table 9: AVX512 10M(2^{24}) Elements Sort

# Elements	Setting	Algorithm	Time(in seconds)	Speedup(vs. std::sort)
10M	32BitInteger	std::stable_sort	1.209813	0.450422
		std::sort	0.544927	1.000000
		ips4o::sort	0.075546	7.213202
		pdqsort	0.121541	4.483477
		avx512::sort	0.284380	1.916190
	32BitFloat	std::stable_sort	1.570905	1.123221
		std::sort	1.764474	1.000000
		ips4o::sort	0.664115	2.656880
		pdqsort	0.756700	2.331801
		avx512::sort	0.436469	4.042612
	64BitInteger	std::stable_sort	1.298014	0.414553
		std::sort	0.538096	1.000000
		ips4o::sort	0.086578	6.215152
		pdqsort	0.143790	3.742223
		avx512::sort	0.654172	0.822560
	64BitFloat	std::stable_sort	1.641503	1.081034
		std::sort	1.774520	1.000000
		ips4o::sort	0.705579	2.514986
		pdqsort	0.772459	2.297235
		avx512::sort	0.795529	2.230616
	32BitKeyValueInt	std::stable_sort	1.092674	0.553509
		std::sort	0.604805	1.000000
		ips4o::sort	0.097853	6.180758
		pdqsort	0.332797	1.817340
		avx512::sort	0.734205	0.823755
	64BitKeyValueInt	std::stable_sort	1.586500	0.407991
		std::sort	0.647278	1.000000
		ips4o::sort	0.135096	4.791253
		pdqsort	0.360557	1.795215
		avx512::sort	2.032096	0.318527
	32BitKeyValueFloat	std::stable_sort	2.307302	0.818679
		std::sort	1.888940	1.000000
		ips4o::sort	0.762947	2.475849
		pdqsort	1.901250	0.993525
		avx512::sort	1.324724	1.425912
	64BitKeyValueFloat	std::stable_sort	2.518255	0.764663
		std::sort	1.925616	1.000000
		ips4o::sort	0.895620	2.150036
		pdqsort	1.918413	1.003755
		avx512::sort	2.047556	0.940446

Figure 20: AVX256 Time Comparisons with Random Number from [-100k, 100k]

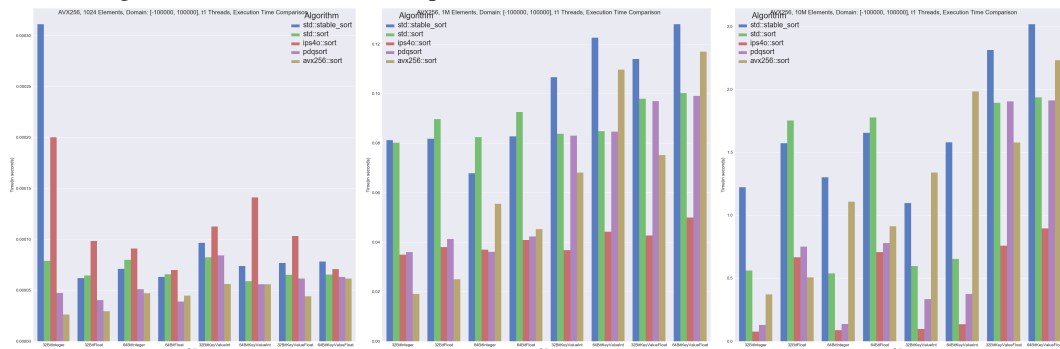


Figure 21: AVX512 Time & Speedup Comparisons with 10M Random Number from [-100k, 100k]

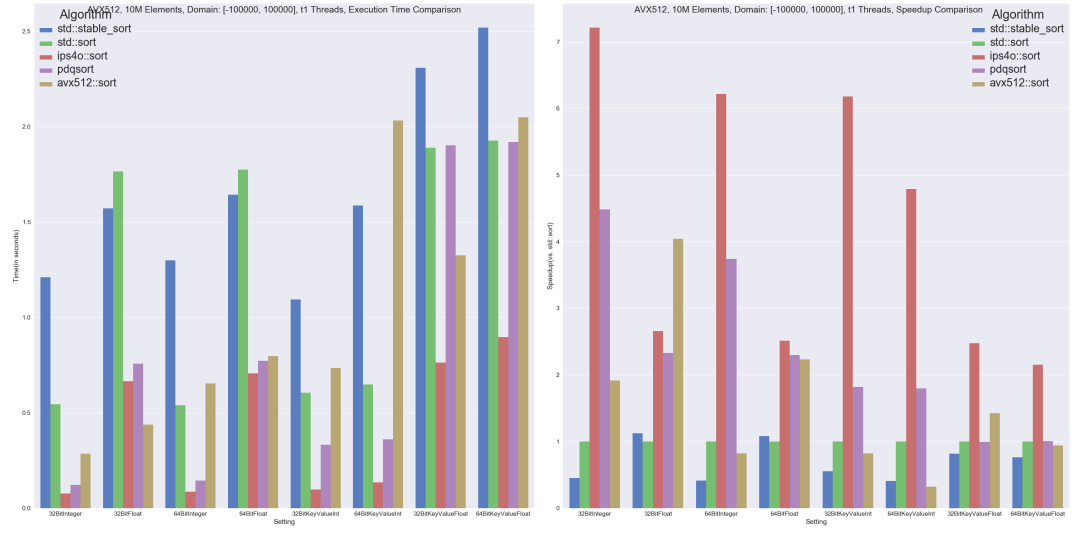


Figure 22: AVX512 Time Comparisons with Random Number from [-100k, 100k], 20 Threads

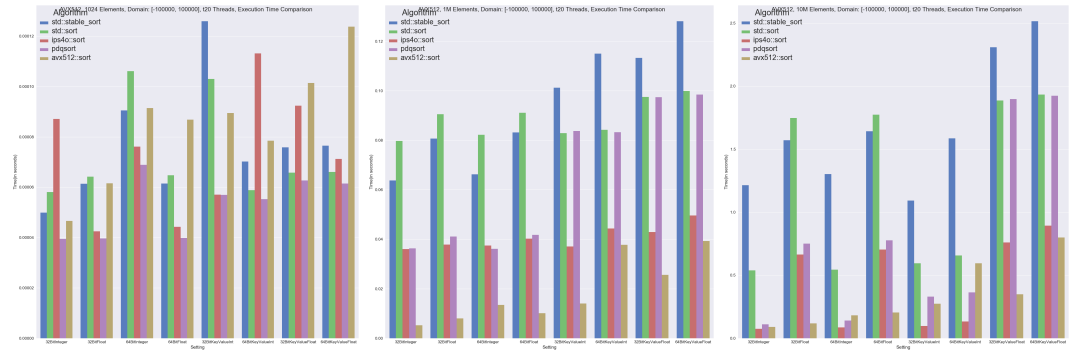
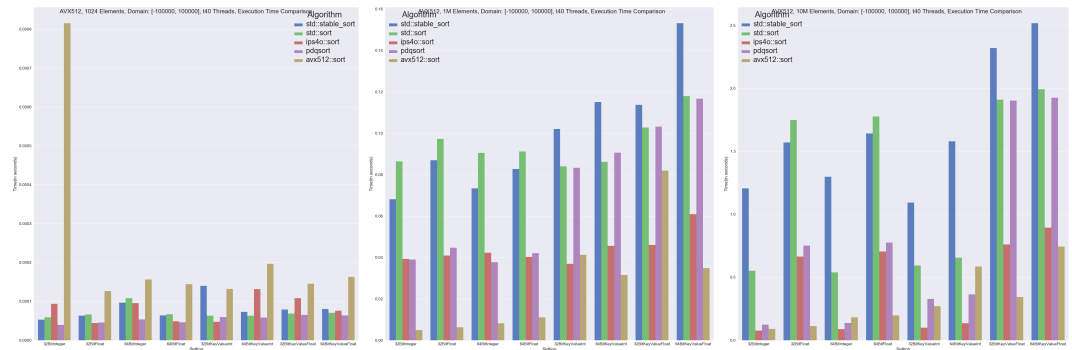


Figure 23: AVX512 Time Comparisons with Random Number from [-100k, 100k], 40 Threads



8.2 LARGE DIAGRAMS

Figure 24: Intra-Register Sort for 2 Registers of 16 Values Each

