2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

Future Vector Microprocessor Extensions for Data Aggregations

Timothy Hayes*†, Oscar Palomar*†, Osman Unsal*, Adrian Cristal*†‡ and Mateo Valero*†
* *Barcelona Supercomputing Center* † *Universitat Politècnica de Catalunya*
‡ *Consejo Superior de Investigaciones Científicas (IIIA-CSIC)*
{*first*}.{*last*}@*bsc.es*

***Abstract*—As the rate of annual data generation grows exponentially, there is a demand to aggregate and summarise vast amounts of information quickly. In the past, frequency scaling was relied upon to push application throughput. Today, Dennard scaling has ceased and further performance must come from exploiting parallelism. Single instruction-multiple data (SIMD) instruction sets offer a highly efficient and scalable way of exploiting data-level parallelism (DLP). While microprocessors originally offered very simple SIMD support targeted at multimedia applications, these extensions have been growing both in width and functionality. Observing this trend, we use a simulation framework to model future SIMD support and then propose and evaluate five different ways of vectorising data aggregation. We find that although data aggregation is abundant in DLP, it is often too irregular to be expressed efficiently using typical SIMD instructions. Based on this observation, we propose a set of novel algorithms and SIMD instructions to better capture this irregular DLP. Furthermore, we discover that the best algorithm is highly dependent on the characteristics of the input. Our proposed solution can dynamically choose the optimal algorithm in the majority of cases and achieves speedups between $2.7\times$ and $7.6\times$ over a scalar baseline.**

I. Introduction

The rate of data generation is growing exponentially each year [1]. Since this has led to enormous volumes of data to manage and query, there is pressure on both software and hardware developers to create solutions that can cope with the increasing requirements. Aggregation is a very useful operation when summarising considerable amounts of data and is a cornerstone of important technologies such as SQL, MapReduce, OLAP cubes, pivot tables and statistical languages. In the TPC-H decision support benchmark, aggregations can dominate eight of the twenty-two queries [2]. A simple aggregation is shown in Figure 1; earnings per persons are grouped together and averaged by age. A summary like this may help the user uncover trends not immediately apparent from the raw data, e.g. if there is a correlation between earnings and age.

For many years, frequency scaling was relied upon to achieve better performance and higher throughput in applications. This technique was generally transparent to the programmer and algorithms were expected to execute faster with every new generation of microprocessor. Due to thermal and power issues, frequency scaling came to an end and

name	age	earnings
Hendry	46	€24,000
O'Sullivan	39	€11,000
Davis	58	€24,000
Higgins	40	€10,000
White	53	€15,000
Williams	40	€8,000
Parrott	51	€9,000
Doherty	45	€6,000

age	earnings (avg)
30-39	€11,000
40-49	€12,000
50-59	€16,000

Figure 1. Example of an aggregation operation. The input table on the left is summarised on the right. Earnings are grouped by age range and averaged.

application performance became something more explicit to the programmer [3]. The clearest way to tackle this is to exploit various forms of parallelism to gain further speedups. Recent developments in microprocessor architectures have pushed a focus on multi-core acceleration. While this is an effective technique to exploit thread-level parallelism (TLP), single instruction-multiple data (SIMD) instruction sets offer a way to accelerate data-level parallelism (DLP), a more efficient form of parallelism [4], [5].

There has typically been some level of support for SIMD instructions in general-purpose microprocessors—commonly dubbed multimedia extensions, e.g. MAX-1, Altivec and SSE. Although these ISA extensions started out relatively simple, successive generations have become more sophisticated and offer wider SIMD registers to process more elements per instruction as well as more intricate instructions to operate on them. For example, Intel's AVX-512 [6] increases the width of the registers to 512 bits and includes mask registers, full gather/scatter support and many non-trivial SIMD instructions. This trend is anticipated to continue in the future, and the SIMD register width and instruction sets are expected to grow further. We predict that the SIMD support found in commodity microprocessors will eventually resemble the instruction sets of classic vector architectures traditionally found in supercomputers [7]. As current SIMD support is still quite restrictive, and the transformation from multimedia extensions to true vector support is still incomplete, the exact potential of exploiting the DLP found in data aggregations is hitherto unknown.

This work makes three principal contributions. (1) We propose and implement several vectorised algorithms for

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.44

data aggregation using common vector SIMD instructions and evaluate them using a cycle-accurate simulation framework with true vector support. (2) In order to determine the sensitivity of the input to performance, we assess the behaviour of these algorithms for a range of data distributions and cardinalities. (3) Leveraging a recent proposal for irregular DLP [8], we augment the simulation framework with new vector instructions and hardware then assess their contribution to data aggregation. We then extend this hardware with minimal additions to create new instructions useful for data aggregations.

There are several notable outcomes of this work. Firstly, we find that the performance of vectorised data aggregation is immensely dependent on the distribution and cardinality of the input. As a consequence, there is not a single vectorised algorithm that provides the best performance in every case. Secondly, we discover that vectorising the algorithms is not trivial due the irregularity of the DLP. We propose two distinctly different types of solutions—the first, *evasion*, attempts to avoid this irregularity through transformation whereas the second, *confrontation*, tackles it head on. The evasion techniques—relying on typical vector SIMD instructions—yield speedups only in a subset of the cardinalities/distributions with significant slowdowns over the scalar baseline in other cases. On the other hand, the confrontation techniques—embracing the irregularity with new SIMD instructions—achieve speedup for all cardinalities/distributions, even when in some cases results are surpassed by an evasion technique. Finally, since cardinality can be determined at runtime, we introduce an adaptive near-optimal implementation that selects the most appropriate algorithm. Our proposed vector implementations exhibit speedups between $2.7\times$ and $7.6\times$ over a scalar baseline for a maximum vector length of 64 and four lockstepped lanes.

Section II introduces our custom simulation framework. Section III outlines our experimental setup, the scalar baseline and discusses the obstacles related to vectorising data aggregations. We propose and evaluate vectorised evasion techniques in Section IV and vectorised confrontation techniques in Section V. Related work is discussed in Section VI and Section VII concludes the article.

II. Simulation Framework

Our goals are to look at the performance characteristics of vectorised aggregation algorithms running on microarchitectures with true vector support as well as proposing new instructions and hardware to facilitate innovative algorithms. Achieving these goals would be impossible if using only existing architectures. Accordingly, we have created a simulation environment to conduct the necessary experiments.

At its heart we use PTLsim [9]—a cycle-accurate x86-64 simulator. PTLsim models many features of modern out-of-order superscalar processors including µop translation, multistage pipelines, speculation and recovery, and a multi-tiered cache hierarchy. We have configured the simulator to behave as close as possible to Intel's Westmere microarchitecture [10]. Table I contains various microarchitectural parameters used in our setup. There are six execution unit clusters in total—a load address generation cluster; a store address generation cluster; a store data cluster; and three arithmetic (non-memory) clusters.

Table I
Microarchitecture Parameters

superscalar and out of order			
parameter	value	parameter	value
fetch width	4	fetch queue	28
frontend width	4	frontend stages	17
dispatch width	4	writeback width	4
commit width	4	reorder buffer	128
issue width per cluster	1	total issue width	6
issue queue per cluster	8	total issue queue	48
load queue	48	store queue	32
L1-d misses	10	L2 misses	16

cache hierarchy					
level	size	latency	line size	ways	sets
L1-i	32 KB	1	64	4	128
L1-d	32 KB	4	64	8	64
L2	256 KB	10	64	8	512

By default, PTLsim uses a fixed latency memory system that does not model bandwidth and contention issues. Recent work on vector processors [11] has shown that they have the ability to saturate a system's available bandwidth, thus making it crucial to model the memory system accurately when executing vectorised algorithms, otherwise the results may be inaccurate and misleading. For this reason, we have integrated DRAMSim2 [12]—a cycle-accurate memory system simulator—into PTLsim and replaced the default memory model. Having an accurate memory model allows the vectorised algorithms to work within a realistic bandwidth envelope and thus enforces a fairer comparison to non-vectorised algorithms. Table II shows various memory system parameters used in our setup. The simulated processor has a frequency of 2.67 GHz so—as a result—the memory controller is clocked every four processor cycles. Additionally, the following address layout scheme is used as it was found to work well with all of our experiments: **row:rank:bank:column:burst**.

Table II
Memory System Parameters

parameter	value	parameter	value
type	DDR3-1333	transaction queue	64
clock	1.5 ns	command queue	256
policy	open page	row accesses	8
queue	per rank per bank	banks	8
scheduling	rank then bank	ranks	4
rows	32,768	columns	2,048
burst length	64 bytes	device width	4

A. Vector SIMD Support

We have modified the simulation framework substantially to give it extensive vector SIMD support. For brevity, we provide a high-level overview that captures the most important features of these additions.

We have extended the x86-64 ISA with sixteen logical vector registers and four logical mask registers. The width of these registers is a configurable parameter of the simulator in order to experiment with different maximum vector lengths (MVL). As the baseline microarchitecture uses register renaming, we also apply this technique to the new vector registers. It has been shown that renaming vector registers is beneficial when the number of physical registers is double the number of logical registers [13] hence we provide thirty-two physical vector registers and eight physical mask registers. There is also an additional vector length register that controls the number of elements operated on by any given vector instruction. We manage this register explicitly using get/set instructions.

The vector capabilities are tightly integrated into the microarchitecture. We have added two new clusters—one to perform the address generation of vector memory instructions and another to execute non-memory vector instructions. The latter contains two functional units which can execute independent non-memory vector instructions in parallel.

We have defined and implemented three classes of vector memory instructions. Each class corresponds to an access pattern and supports *load*, *store* and *prefetch* instructions. **(1) unit-stride**: memory is accessed contiguously. This is the most efficient access pattern due to the spatial locality of the elements accessed. **(2) strided**: memory instructions use a base address and a parameter that refers to the increment in memory between elements. **(3) indexed**: also known as gather/scatter, these instructions use a base address and an additional vector register of offsets.

Unit-stride and strided instructions calculate their addresses formulaically. The number of cycles spent performing the address generation depends on the number of cache lines needed to fulfil the request, e.g. four cache lines accessed would require four cycles in the functional unit. Indexed memory instructions require adding an offset to a base address and need $\frac{VL}{lanes}$ cycles to perform address generation where VL is the vector length of the instruction. The number of cycles needed to complete a memory instruction depends on how many individual cache lines are requested and whether or not these are already resident in the cache hierarchy.

Bulding upon an existing technique [14], [15], we configure the vector register file to bypass the L1-d cache and go directly to the L2 cache. This way more bandwidth can be provided at the expense of higher latency. It has been shown in the cited work that this extra latency is easily amortised due the high number of elements operated on per individual instruction. We interleave the L2 cache sets using a simple mapping scheme based on irreducible polynomials suggested in [16], [17]. This scheme eliminates pathological behaviour where a particular strided memory access uses the same cache set for all its requests.

To operate on the vector registers, we have added a suite of non-memory vector instructions which is summarised in Table III. Mask instructions require just one cycle to complete. Most vector instructions require $\frac{VL}{lanes}$ cycles to pass through a functional unit. Reduction instructions are calculated slightly differently; there is a partial reduction local to each lane requiring $\frac{VL}{lanes}-1$ cycles and then $log_2 lanes$ additional cycles needed for interlane reduction. Comparison instructions produce a result to a mask register. The permutative instructions—which rearrange the order of the input vector's elements—require a mask operand, whereas most other instructions can use masks optionally. `iota` is an instruction found in the CRAY-1.

Table III
NON-MEMORY VECTOR INSTRUCTIONS

class	instructions
initialisation	set all, clear all, iota
arithmetic	maximum, add, subtract, multiply
bitwise logical	and, shift left, shift right
comparison	not equal, not equal to zero
mask	popcount
permutative	compress, expand
reduction	maximum, minimum, sum
other	get/set element, get/set vlen

III. EXPERIMENTAL SETUP

Here we describe the experimental setup. Our goal is to define a representative data-intensive aggregation query, implement it in a variety of ways and evaluate the implementations with a diverse range of parameters. This will help expose the strengths and weaknesses of different algorithm designs. Additionally, we present a scalar aggregation algorithm which we use as a common baseline in subsequent experiments. Finally, we discuss the obstacles to vectorising data aggregation and propose two possible solution paths.

A. Query and Input Data

In our experiments, we evaluate the SQL query in Figure 2. This type of query has been successfully used in prior work to evaluate data aggregations [18]–[20]; its performance depends highly both on the underlying implementation as well as the characteristics of the input data. `r` is a two-column table with `n` rows consisting of a 32-bit integer group key `g` and a 32-bit integer value `v`. The result is a three-column output table where each row contains a group, the frequency of that group `count` and the sum of all values corresponding to that group `sum`. We emulate the behaviour of a column-oriented database management system (DBMS) in which columns are stored contiguously

as arrays in memory. These types of DBMS are becoming prevalent in large datacentres used for online analytical processing [21].

```
1: SELECT g, COUNT(*), SUM(v)
2: FROM r GROUP BY g
```

Figure 2. SQL code used in experiments

In all the experiments, we fix the number of input rows n at 10,000,000. This value is sufficient to represent behaviour indicative of non-cache resident datasets while also being small enough to simulate to completion in a reasonable time frame. The value column v is a uniform distribution in the interval $[0, 9]$; since this column does not directly affect the performance of the different algorithms, it remains constant in all experiments. We generate 110 variations of the group column g by varying the distribution and cardinality c of the data.

We use five unique data distributions similar to the ones used by Cieslewicz et al. [18]. (1) uniform: a pseudo-random selection in the interval $[0, c)$ with equal probability. (2) sorted: a presorted uniform distribution. (3) sequential: a repeating sequence $\{0, 1, 2, ..., c-1\}$. (4) hhitter: similar to uniform however 50% of the data is a single heavy hitting value. (5) zipf: a pseudo-random selection in the interval $[0, c)$ with a Zipfian probability.

There are 22 possible cardinalities $c \in \{10,000,000,$ $5,000,000,$ $2,500,000,$ ..., 38, 19, 9, $4\}$. Due to the nature of each distribution, c represents a maximum possible cardinality rather than a guaranteed cardinality. For example, it is not always possible to generate a Zipfian distribution where $|g| = c$, therefore—for zipf—c represents the upper bound of the domain in which we sample from rather than a strict cardinality. sequential is the only distribution where c guarantees both a maximum and an actual cardinality in every case. Unless otherwise stated, cardinality refers to this upper bound.

For the sake of discussion, we group the cardinalities into four divisions. (1) *low* cardinalities $[4,$ ..., $152]$, e.g. gender of a person. (2) *low-normal* cardinalities $[305,$ ..., $9,765]$, e.g. date of birth of a client. (3) *high-normal* cardinalities $[19,531,$ ..., $312,500]$, e.g. a zip or postal code. (4) *high* cardinalities $[625,000,$ ..., $10,000,000]$, e.g. a passport number.

We assume that the application has a priori knowledge that the sorted datasets are already ordered and thus avoids the overhead of resorting. This is normal in DBMSs in which similar metadata is used to choose between alternative algorithms and make optimisations. This assumption also helps identify performance trends independent of a sorting phase.

In some aggregation techniques, it is useful to detect the maximum group key and use it to improve the algorithm's runtime behaviour. In algorithms with a sorting phase—or if the input is presorted—the maximum group key is simply the last value in the array. In algorithms without a sorting phase—excluding presorted input—we locate an exact maximum group key by scanning the entire array g. We find that this adds little overhead compared to the aggregation itself, however, it could be replaced with sampling and some additional checks.

Since we are already looking at many variables, we fix the vector parameters at $MVL = 64$ and $lanes = 4$. These parameters were shown to be reasonable in recent vector work [8], [11]. They also represent a configuration that we anticipate could eventually appear on the market given current trends. We report all our results using cycles per tuple (CPT)—the total number of cycles needed to execute the algorithm divided by the total number of input tuples n.

B. Scalar Baseline

Here we introduce the baseline algorithm ***scalar***, designed without any vector SIMD instructions. We divide its implementation into four steps. (1) Find the highest value, maxg, stored in the array g. (2) Clear $maxg + 1$ cells of the output tables count and sum. (3) Aggregate the input arrays g and v to output tables count and sum. Pseudocode for this step is shown in Figure 3. (4) Compress the tuples to remove absent groups with NULL results.

```
1: for each i in n do
2:     count[g[i]]++;
3:     sum[g[i]] += v[i];
4: end for
```

Figure 3. Pseudocode for step 3 of ***scalar***

The results are shown in Figure 4. For all datasets, the performance is similar in *low* and *low-normal* but then changes drastically entering *high-normal*. When $c = 9,765$, the L1-d cache capacity of 32 KB is exceeded. At this point hhitter, uniform and zipf increase their CPT intensely; uniform alone exhibits a dramatic $8\times$ increase in CPT. This behaviour is not surprising as a uniform distribution exhibits poor locality when the bookkeeping structures exceed the cache size. In contrast, sorted does not take any significant hit in performance in *high-normal* as having the tuples presorted introduces a lot more locality. This effect wears off in *high* and sorted experiences a steep slope in its CPT as well.

sequential follows a similar pattern to sorted although slightly increases its CPT in *high-normal*. After processing the first 9,765 tuples out of n, the L1-d cache will be filled and processing subsequent tuples causes dirty line evictions thus reducing the memory system's performance. These evictions can occur with sorted as well, but unlike sequential, there will be repeated values stored adjacently causing more locality. This behaviour would suggest that sorting all the datasets will lead to better performance, however, the cost of doing this with a scalar ISA would be very high—especially for a large n.

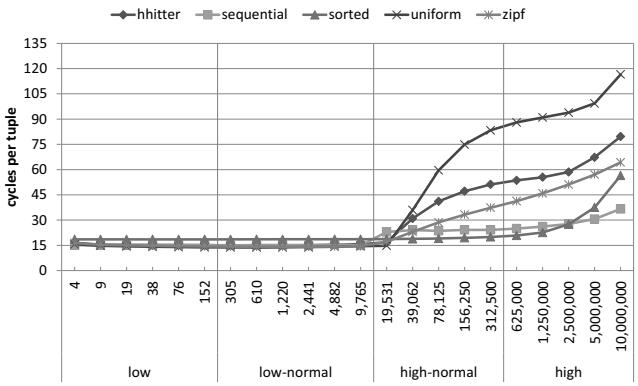


Figure 4. Results for ***scalar*** baseline

C. DLP and Vectorisation

Data-level parallelism (DLP) is accomplished when the same operations are applied to multiple elements of homogeneous data, i.e. a vector of data. DLP can be achieved by leveraging a vector SIMD instruction set such as the one described in Section II-A. We further categorise DLP as either regular or irregular.

Regular DLP is a form of DLP in which result `i` of a vector procedure depends only on element `i` of its input vectors' operands, i.e. every element is independent. A typical vector SIMD instruction set is generally geared towards regular DLP.

Irregular DLP can be defined as DLP where result `i` of a vector procedure depends on element `i` of its input vectors' operands and may additionally depend on other results of the vector procedure. It is still DLP as the same operations are applied uniformly on all data, however, the result of one action may depend on the outcome of another action within the same unit of work, e.g. SIMD instruction.

Our reference ***scalar*** baseline is a relatively straightforward algorithm that makes use of tables. Nevertheless—due to the irregularity of the DLP—there are numerous obstacles when vectorising the code. Updating a table is accomplished by—(a) an indexed load to the table (b) modifying the value (c) an indexed store to the table. In a SIMD model of computation, this translates to—(a) gathering multiple table entries to a vector register (b) modifying the vector of loaded values (c) scattering the modified values back to the table. If the indices used in the gather/scatter operations are not unique, i.e. conflicting, the behaviour is undefined and updates can be lost causing erroneous output. We refer to this as a gather-modify-scatter (GMS) conflict.

There are two possible ways to tackle this. One is to *evade* the irregularity by transforming the problem into something more regular and then vectorising it. The other is to *confront* the irregularity directly through the use of novel instructions. In Section IV we evaluate our evasion solutions and in Section V we evaluate our confrontation solutions.

IV. Evasion Techniques

In this section, we propose and evaluate two alternative vectorisable solutions using typical vector SIMD instructions.

A. Standard Sorted Reduce

Historically, vector architectures have offered some support for aggregating vectors to scalars in the form of reduction instructions [22]. A reduction instruction takes a single vector register as input, applies an associative/commutative operation to all its elements, and outputs a single reduced scalar value. Figure 5 shows an example of a sum reduction operation performed on a vector register of eight elements. There are two parallel lockstepped lanes that each processes four elements in three cycles followed by one extra ($log_2 lanes$) cycle of interlane reduction. We classify reductions as semi-regular DLP instructions. They are not completely regular because the output element depends on more than input element `i`, yet, they are not irregular either as there is a single output value and, therefore, output element `i` does not depend on any other output element.

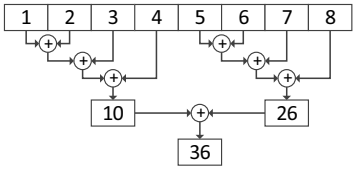


Figure 5. Sum reduction when $VL = 8$ and $lanes = 2$

We evaluate the benefit of using these types of instructions in data aggregation. If the input is sorted, vector reduction instructions can be used directly. If not, the input must be sorted first. Our algorithm is as follows. (1) If not already sorted, `g` is sorted using `v` as the associated payload. (2) The sorted `g` is scanned for runs of repeated keys. Runs can be found by first comparing `g[i]` with `g[i+1]` to generate vector masks. The distance between set bits in these vector masks corresponds to the length of a run. These lengths also correspond to the elements of the output column `count`. (3) The run lengths are used to load and reduce segments of `v`. Run lengths that exceed the MVL are stripmined.

To sort the input arrays in step 1 we choose radix sort [23]. It is a good match for this algorithm for several reasons. Firstly, it is vectorisable using typical vector SIMD instructions. Secondly, recent work [8] demonstrated that it outperforms quicksort and bitonic mergesort when $MVL = 64$ and $lanes = 4$—the same configuration used in this work. Thirdly, it has an equal CPT for any input size `n`, hence making it scalable for larger datasets. Finally, it can be optimised for a particular maximum group key thereby reducing the cost of sorting any particular cardinality.

The results of ***standard sorted reduce*** evaluated with all data distributions and cardinalities are shown in Figure 6. To

make comparisons easier, we keep the scale of the y-axis the same as the scalar baseline for all vector experiments. In Table IV, a summary is given of the overall performance by taking the average speedup (and standard deviation) over ***scalar*** for each cardinality division. Highlighted cells indicate that this is the best average performance so far for that particular combination of dataset and cardinality division.

sorted is the only dataset that does not cause additional sorting overhead, as such, we see the cost of the aggregation step itself. Its performance is consistent for *low*, *low-normal* and *high-normal* but then diminishes in *high*. The increasing cardinality causes the average run length to decrease and serialises the algorithm thereby underutilising the vector unit. In most cases, it can be seen that hhitter, sequential, uniform and zipf show slowdowns over ***scalar***; only uniform exhibits a $1.1\times$ average speedup for *high*. These slowdowns are due to the overhead of sorting the input which often exceeds the total cost of ***scalar***.

Although being the most efficient SIMD sorting algorithm, radix sort must undergo significant transformations to be vectorised. The vectorised algorithm suffers from two major bottlenecks. (1) In order to avoid GMS conflicts, its internal bookkeeping structures need to be replicated by the number of elements in a vector register. (2) To ensure sorting stability, each element of a vector register must process a contiguous portion of the input. To achieve this effect, the input must be loaded into a vector register using a strided memory access pattern in lieu of a unit-stride one.

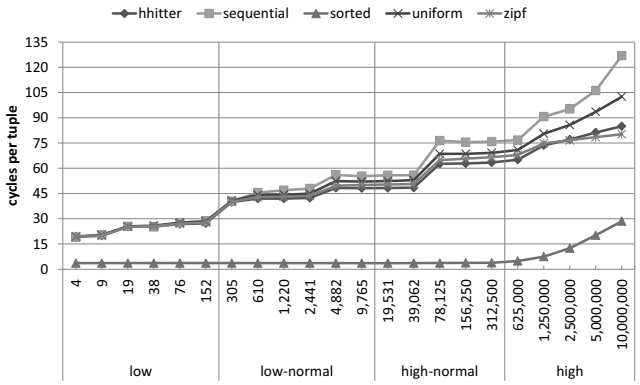


Figure 6. Results for ***standard sorted reduce***

Table IV
AVERAGE SPEEDUPS (STDEV) OF ***standard sorted reduce*** OVER BASELINE. HIGHLIGHTED CELLS MARK BEST RESULT SO FAR.

	low	*low-normal*	*high-normal*	*high*
hhitter	$0.7\times$ (0.1)	$0.3\times$ (0)	$0.6\times$ (0.2)	$0.8\times$ (0.1)
sequential	$0.6\times$ (0.1)	$0.3\times$ (0)	$0.4\times$ (0.1)	$0.3\times$ (0)
sorted	$5.1\times$ (0)	$5.1\times$ (0)	$5.2\times$ (0.1)	$2.7\times$ (1)
uniform	$0.6\times$ (0.1)	$0.3\times$ (0)	$0.8\times$ (0.4)	$1.1\times$ (0.1)
zipf	$0.6\times$ (0.1)	$0.3\times$ (0)	$0.5\times$ (0.1)	$0.7\times$ (0.1)

B. Polytable

It is also possible to make a vectorised translation of ***scalar*** using vector SIMD instructions. Steps 1, 2 and 4 can be vectorised directly using typical SIMD instructions, however, in a similar vein to radix sort, the third step requires transformation.

To circumvent GMS conflicts, we must replicate the output tables `count` and `sum` for every element of a vector register, i.e. there are MVL independent versions of each table. Figure 7 shows the process of incrementing the `count` table when $MVL = 4$. In the figure, input array `g` is arranged in blocks of consecutive MVL elements. The elements with dotted patterns have already been processed. The highlighted values are currently being used to update the table. In this case it can be seen that there are multiple instances of the value 3 in the vector register (`vreg`). This duplication would cause a GMS conflict if a single table were used, however, since each vector element accesses a local copy, we avoid conflicts entirely.

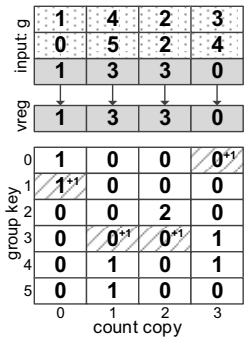


Figure 7. Table replication used to avoid GMS conflicts

After the input has been processed, the local copies of `count` and `sum` must be reduced to singular global tables. MVL consecutive elements—which form a single group—are loaded into the vector register (`vreg`) that is then summed together using a reduction instruction. This local to global reduction is illustrated in Figure 8.

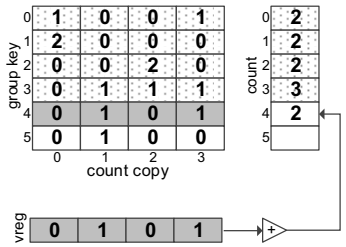


Figure 8. Local tables are reduced to a single global table

The results of ***polytable*** are shown in Figure 9 and Table V. For *low*, all datasets exhibit a positive speedup. Due to the arrangement of the table structures, sorted shows

the biggest improvement and sequential exhibits the least improvement. This is due to the layout of the MVL table copies. Replications are stored contiguously in memory, i.e. the cell for group `k`'s local copy `i` is adjacent in memory to copy `i+1`. Since sorted contains long runs of the same group, the number of cache lines accessed is minimal. In contrast, sequential has the opposite behaviour. The datasets have runs of ascending groups which causes a strided memory access pattern where the stride is $MVL+1$ elements, i.e. a diagonal access through the structure. Since the MVL is larger than the number of elements in a cache line, MVL cache lines will be accessed with every memory instruction. All other datasets exhibit performance between these two extremes.

After *low*, the performance begins to decrease. Similar to the ***scalar*** baseline, the tables grow larger than what the cache can accommodate and performance drops. In this case, replicating the tables causes the deterioration to happen sooner. In the scalar baseline, this transition occurs when $c = 9,765$ whereas here it happens when $c = 152$ which is sixty-four—the MVL—times smaller than the former. For hhitter, sequential, uniform and zipf the results are always worse than ***scalar***. sorted continues to outperform ***scalar*** in *low-normal* and *high-normal* due to the spatial locality of its accesses, however, in *high*, it deteriorates and becomes worse than ***scalar***. A slightly surprising result here is that for sorted, *low* and *low-normal* outperform their counterparts in ***standard sorted reduce***. This due to an unignorable overhead incurred when scanning the input to build the array of run lengths.

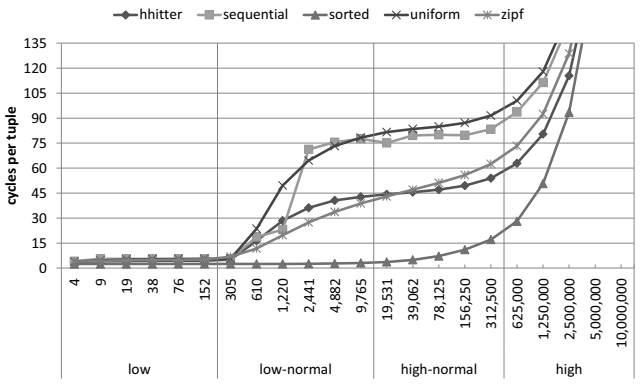


Figure 9. Results for ***polytable***

Table V
AVERAGE SPEEDUPS (STDEV) OF ***polytable*** OVER BASELINE. HIGHLIGHTED CELLS MARK BEST RESULT SO FAR.

	low	*low-normal*	*high-normal*	*high*
hhitter	3.7× (0.4)	0.9× (1)	0.8× (0.2)	0.5× (0.2)
sequential	2.9× (0.4)	0.8× (1)	0.3× (0)	0.2× (0.1)
sorted	7.6× (0)	7× (0.6)	2.9× (1.6)	0.4× (0.2)
uniform	3× (0.6)	0.7× (0.9)	0.6× (0.3)	0.6× (0.2)
zipf	3.3× (0.6)	0.9× (0.7)	0.5× (0.1)	0.4× (0.2)

C. Summary

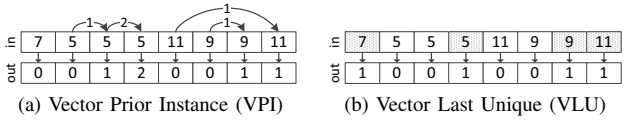
We have evaluated two distinct techniques that vectorise data aggregations through algorithm transformation. If the input is already sorted, there are positive speedups to be gained using ***polytable*** for lower cardinalities and ***standard sorted reduce*** for higher cardinalities. For non-sorted data distributions, it is beneficial to use ***polytable*** if the cardinality is very low. For other combinations of distribution and cardinality, neither of these techniques suffice. These limitations arise due to the transformations necessary to vectorise data aggregation using a typical vector SIMD ISA. These findings motivate us to explore other techniques using novel vector SIMD instructions which will allow us to vectorise the algorithms without these detrimental transformations.

V. CONFRONTATION TECHNIQUES

In this section we look at alternative solutions that attempt to confront the irregular DLP head on rather than evade it.

A. Advanced Sorted Reduce

The vectorised radix sort used in Section IV-A suffers from performance bottlenecks caused by algorithm transformation. Recent work on vectorised sorting algorithms proposed VSR sort [8]. VSR sort is a novel vectorised implementation of radix sort that avoids replicating its internal table structures and processes the input arrays sequentially. Contiguous portions of the input are read into vector registers using an efficient unit-stride memory access pattern; the algorithm then searches for elements that may cause GMS conflicts and corrects them accordingly before accessing the bookkeeping structures. To enable this new algorithm in a vector SIMD architecture, two new instructions are required—`VPI` and `VLU`. A detailed explanation of the VSR sort algorithm is beyond the scope of this article, however, we provide an overview of the new instruction and their hardware implementation as these pertain to later sections.



(a) Vector Prior Instance (VPI) (b) Vector Last Unique (VLU)

Figure 10. New instructions for VSR Sort

`Vector Prior Instances (VPI)` uses a single vector register as input, processes it serially, and outputs another vector register as a result. Each element of the output asserts exactly how many instances of a value in the corresponding element of the input register have been seen before in the register. An example is given in Figure 10a (elements are processed from left to right). In each pass of VSR sort, a histogram of the input is first created. Using the values of the histogram as offsets, the input is scattered to an auxiliary array where its order becomes partially or fully sorted depending on the pass. Since it is possible

that multiple input values within a SIMD operation may correspond to the same histogram bin, scattering these values would go to the same location in the auxiliary array. To circumvent this, `VPI` is used to correct the offsets going to conflicting locations by transforming them to adjacent locations instead.

`Vector Last Unique (VLU)` also uses a single vector register as input but produces a vector mask as a result. The idea is to mark the last instance of any particular value found. An example is given in Figure 10b (elements are processed from left to right). A bit in the output mask register is set if the corresponding value in the input vector is not seen afterwards. In VSR sort, `VLU` is used to select a non-conflicting subset of indices to the histogram and increment them based on the number of corrections made by `VPI`. Thus, `VPI` and `VLU` together can be used to increment a histogram structure without GMS conflicts.

`VPI` and `VLU` are implemented using a CAM structure with MVL entries. Figure 11 illustrates such a setup where $MVL = 8$. An input vector register is processed from the least significant element ($idx = 0$) to the most significant element ($idx = 7$). The diagram shows that six of the eight elements have already been processed with the seventh in progress. Processing each input element requires two cycles; activity in the first cycle is shown with solid lines and activity in the second is shown with broken lines. In the first cycle, the input value 9 is used as a key and a valid entry in the CAM is found. The `count` field of the CAM entry is copied to the corresponding element of the output vector and also routed to an increment unit. In the second cycle, the result of the increment is written back to the `count` field. Simultaneously, the `last idx` field of the CAM entry is updated with the value 6—the index of the input/output entry being processed at that moment. When all input elements have been processed, the output vector will contain the results of `VPI` whereas `VLU` can be generated by converting the `last idx` field of all valid CAM entries to a bitmask. To reduce instruction latency, the CAM is given p ports. The CAM structure can be updated in parallel provided that a slice of p adjacent elements of the input vector has no conflicts. For more details the reader is referred to [8].

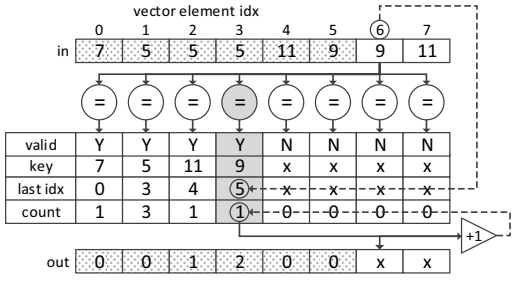


Figure 11. Hardware implementation of `VPI` and `VLU`

We now evaluate the same algorithm used in ***standard sorted reduce*** but replace radix sort with VSR sort while keeping all other steps equal. The results are shown in Figure 12 and Table VI. Since the sorted dataset can skip the sorting step, its behaviour and performance remain equal to ***standard sorted reduce***; these cases are marked with a Ξ symbol.

For hhitter, sequential, uniform and zipf the results are always better than ***standard sorted reduce***. There are still some slowdowns over ***scalar*** for *low* and *low-normal*. Despite the performance of VSR sort being better than radix sort, the overhead is still too high to surpass the CPT of ***scalar*** for lower cardinalities. For *high-normal*, this overhead becomes less significant and we achieve speedups in all cases.

For *high*, hhitter, uniform and zipf continue to exhibit speedups whereas sequential shows a slowdown. The reason for this is twofold: (1) sequential exhibits good locality in *high* for ***scalar*** thereby having better performance relative to the other three datasets. (2) The average vector length is reduced to values below the MVL in *high*. For example, when $c = 10,000,000$ the vector length of every reduction is 1 and this reduces performance considerably. This second point also affects hhitter, uniform and zipf for *high*, but to a lesser extreme than sequential.

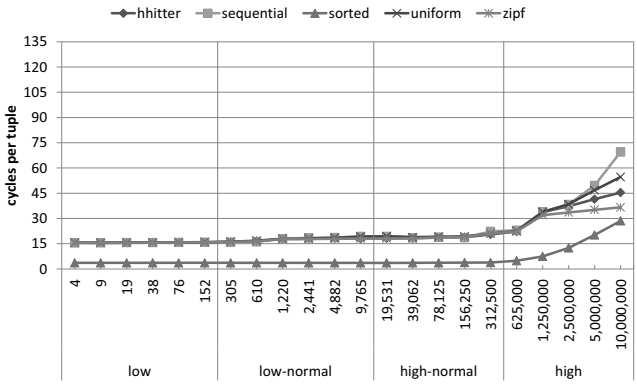


Figure 12. Results for ***advanced sorted reduce***

Table VI
AVERAGE SPEEDUPS (STDEV) OF ***advanced sorted reduce*** OVER BASELINE. HIGHLIGHTED CELLS MARK BEST RESULT SO FAR.

	low	*low-normal*	*high-normal*	*high*
hhitter	1× (0)	0.9× (0)	2× (0.7)	1.8× (0.4)
sequential	1× (0)	0.9× (0.1)	1.2× (0.1)	0.7× (0.2)
sorted	5.1× (0) Ξ	5.1× (0) Ξ	5.2× (0.1) Ξ	2.7× (1) Ξ
uniform	0.9× (0.1)	0.8× (0)	2.7× (1.4)	2.7× (0.7)
zipf	1× (0.1)	0.8× (0)	1.5× (0.4)	1.6× (0.2)

B. Monotable

One problem with the ***polytable*** approach of Section IV-B is that the table replication destroys any locality that may otherwise be present in the ***scalar*** baseline. Here we propose

an alternative implementation called ***monotable*** that draws from the novel instructions used in ***advanced sorted reduce***.

`VPI` and `VLU` use a hardware implementation based on a CAM and adder. We propose reusing this hardware structure and building new functionality on top. We define a new set of instructions called `Vector Group Aggregate (VGAx)` that can aid us further when vectorising data aggregation. There are three operations supported which form new instructions: sum (`VGAsum`), minimum (`VGAmin`) and maximum (`VGAmax`). Each `VGAx` instruction uses two registers as input—a vector of groups `ing` and a vector of values `inv`. The instructions produce a vector `out` of running partial aggregates among values of the same group.

We can implement these instructions with relatively minor additions to the hardware already in place for `VPI` and `VLU`. As an example, we describe `VGAsum`. The semantics are illustrated in Figure 13 and the implementation is shown in Figure 14. For each input element, instead of incrementing its CAM entry by one as would be done with `VPI`, the entry is summed with the corresponding value in `inv`. The semantics resemble `VPI` where its values would be a vector of 1s, however, an important difference is that the output of `VPI` comes from the CAM entry's value before the increment whereas the output of `VGAsum` is taken after the increment.

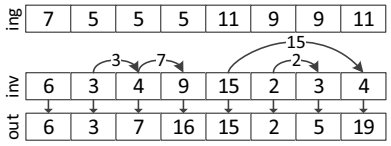


Figure 13. Semantics of the `VGAsum` instruction

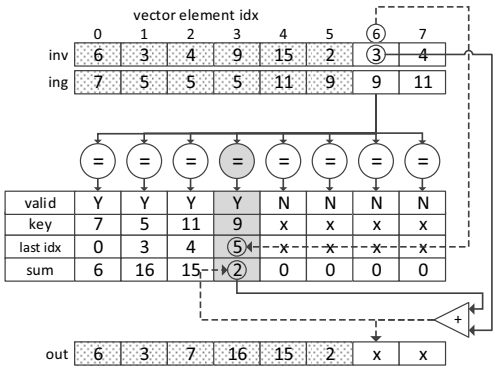


Figure 14. Hardware implementation of `VGAsum`

We use `VGAsum` to build a vectorised version of ***scalar*** using non-replicated tables with no GMS conflicts. Combining `VGAsum` with `VLU` allows us to update a single table in parallel. Figure 15 shows the pseudocode of this step. The masked `scatter` instruction could optionally be replaced with a `compress` followed by a non-masked `scatter`.

```
1: v2 ← vgasum(v0, v1)                    ▷ groups in v0 & values in v1
2: m0 ← vlu(v0)
3: v3 ← gather(base=table, idx=v0, mask=m0)
4: v4 ← vadd(v2, v3)
5: scatter(base=table, idx=v0, vals=v4, mask=m0)
```

Figure 15. Pseudocode for updating a table using `VGAsum`

Figure 16 and Table VII show the results of ***monotable***. The graph resembles the trends found in ***scalar*** (see Figure 4) but with lower CPTs. For *low*, ***monotable*** exhibits good performance for hhitter, sequential, uniform and zipf and outperforms ***polytable***—the only evasion method that was useful for this cardinality division. sorted is not as fast as ***polytable*** for *low* and *low normal*, which is understandable since the majority of the `VGAsum` instruction's input will cause CAM port conflicts and, therefore, pay the maximum latency. In contrast, ***monotable*** outperforms ***polytable*** in all cases for sorted in *high-normal* and *high*.

It can be seen that ***monotable*** has consistent performance for lower cardinalities, but for higher cardinalities hhitter, sequential and uniform become worse whereas sequential and sorted remain relatively stable. This behaviour is related to the locality of memory accesses. When $c \leq 9,765$, the data structures can reside fully in the L2 cache. When this cardinality is exceeded—depending on the distribution of the data—it may destroy the locality. Despite this behaviour, all the datasets in the higher cardinalities exhibit a positive speedup and beat the ***polytable*** method in every case. Compared with ***advanced sorted reduce***, sometimes the performance is better and sometimes worse.

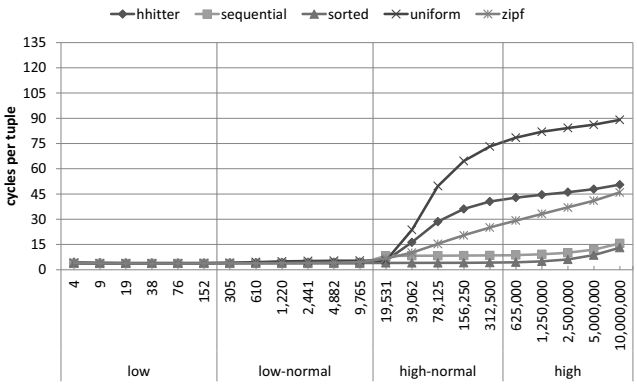


Figure 16. Results for ***monotable***

C. Partially Sorted Monotable

We observe that ***monotable*** works particularly well for the lower cardinalities. For higher cardinalities, some of the datasets lose their cache locality and exhibit rapid increases in CPT. sorted and sequential—the datasets that do not lose their locality—maintain more consistent behaviour. We estimate that to achieve the optimal behaviour of ***monotable***,

Table VII
AVERAGE SPEEDUPS (STDEV) OF ***monotable*** OVER BASELINE. HIGHLIGHTED CELLS MARK BEST RESULT SO FAR.

	low	*low-normal*	*high-normal*	*high*
hhitter	3.9× (0.1)	3.5x (0.1)	1.8× (0.8)	1.3× (0.1)
sequential	4.1× (0)	4.1× (0.1)	2.9× (0)	2.7× (0.2)
sorted	4.6× (0)	4.6× (0)	4.7× (0)	4.5× (0.2)
uniform	3.8× (0.1)	2.9× (0.3)	1.5× (0.7)	1.2× (0.1)
zipf	4× (0.1)	3.5× (0.2)	2× (0.4)	1.4× (0)

the input does not necessarily have to be fully sorted but instead be partitioned in such a way that maximises temporal locality.

In ***advanced sorted reduce***, we use VSR sort to fully sort the input before reducing it. Each pass of VSR sort orders the input according to a subset of bits of each value, building from the order already found by previous passes. By default, VSR sort finishes after the last pass processes the most significant bits of the values resulting in a completely sorted input. Each pass contributes to the algorithm's overhead. If only sorting on a subset of each value's bits is necessary, the number of passes could be significantly reduced. In ***monotable***, it is not paramount that all group keys be stored together contiguously like in the ***sorted reduce*** methods. Instead, it should be sufficient to position repeated groups keys just close enough to one another that nothing in between will evict that group key's line from the cache. Accordingly, we propose partially sorting the inputs with higher cardinalities before executing ***monotable***.

We modify VSR sort to perform a single pass of the algorithm on a subset of bits between the most significant bit of the maximum group key and a user-specified offset. As the L2 cache in our experiments is 256 KB and each group key requires 4 bytes, up to 16 bits of each value could be ignored when sorting. Using a configuration that considers the remaining 16 bits would divide the input into partitions with a maximum of 65,536 unique groups. In practice, we need only sort the most significant 8 bits of *high-normal* cardinalities and increase this gradually all the way to 11 bits for the largest cardinality in *high*. We do not need to partially sort any datasets in *low* and *low-normal* as these exhibit good temporal locality already.

Figure 17 and Table VIII show the results. Since we need not partially sort the lower cardinalities or the sorted dataset, their behaviour and performance remain equal to ***monotable***; these cases are marked with a Ξ symbol. For *high-normal* and *high* there is a significant increase in performance for hhitter, uniform and zipf. These results are considerably better than ***polytable***, ***monotable*** and either ***sorted reduce*** methods. sequential is the only dataset that takes a hit in performance over ***monotable*** for the higher cardinalities. This degradation is because sequential already exhibits enough spatial locality to compensate for a lack of temporal locality and partially sorting the input only adds to its CPT.

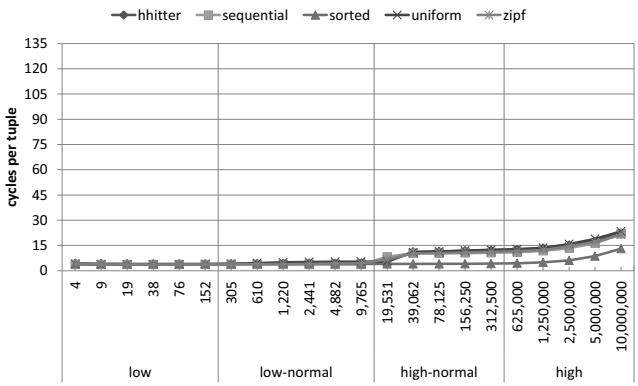


Figure 17. Results for ***partially sorted monotable***

Table VIII
AVERAGE SPEEDUPS (STDEV) OF ***partially sorted monotable*** OVER BASELINE. HIGHLIGHTED CELLS MARK BEST RESULT SO FAR.

	low	*low-normal*	*high-normal*	*high*
hhitter	3.9× (0.1) Ξ	3.5× (0.1) Ξ	3.5× (0.6)	3.9× (0.3)
sequential	4.1× (0) Ξ	4.1× (0.1) Ξ	2.4× (0.2)	2× (0.2)
sorted	4.6× (0) Ξ	4.6× (0) Ξ	4.7× (0) Ξ	4.5× (0.2) Ξ
uniform	3.8× (0.1) Ξ	2.9× (0.3) Ξ	4.8× (1.8)	5.9× (0.8)
zipf	4× (0.1) Ξ	3.5× (0.2) Ξ	2.8× (0.4)	3.4× (0.3)

D. Summary

We have evaluated a broad range of datasets using five vectorised data aggregation techniques—each with varying success—that either evade or confront the irregular DLP inherent to aggregation. In the majority of cases, using an evasion technique adds too much overhead to be useful whereas our proposed confrontation techniques show more promising results. Table IX summarises the best results for all data distributions and cardinality divisions. Each cell provides an average speedup over ***scalar***. The ‡ symbol indicates it may not be practical to detect this configuration at runtime in order to apply the most suitable algorithm.

Table IX
BEST AVERAGE SPEEDUP (ALGORITHM) OVER BASELINE

	low	*low-normal*	*high-normal*	*high*
hhitter	3.9× (mono)	3.5× (mono)	3.5× (psm)	3.9× (psm)
sequential	4.1× (mono)	4.1× (mono)	2.9× (mono) ‡	2.7× (mono) ‡
sorted	7.6× (poly)	7.0× (poly)	5.2× (sr)	4.5× (mono)
uniform	3.9× (mono)	2.9× (mono)	4.8× (psm)	6.0× (psm)
zipf	4.0× (mono)	3.5× (mono)	2.8× (psm)	3.4× (psm)

In all cases the results are positive although there is no single algorithm that matches all of the configurations. The best speedup is achieved using a variety of different techniques. For *low* and *low-normal*, the non-sorted datasets fare best using ***monotable*** whereas sorted achieves the highest speedup using ***polytable***. For hhitter, uniform and zipf the best method for *high-normal* and *high* is ***partially sorted monotable*** and for sequential the best choice is ***monotable***. sorted

performs best using either of the ***sorted reduce*** methods for *high-normal* and ***monotable*** for *high*.

In most situations, we have enough information to choose the best method for that particular combination of dataset and cardinality dynamically. In general, the rule is to apply ***monotable*** to non-sorted datasets for lower cardinalities and ***partially sorted monotable*** for higher cardinalities; for sorted datasets, ***polytable*** can be used for lower cardinalities and ***sorted reduce*** and ***monotable*** for higher cardinalities. Only detecting the case of sequential with higher cardinalities would prove difficult, however, the difference between ***partially sorted monotable*** and ***monotable*** for these two cases is not overly significant. Using the ideal algorithm selection yields a $4.21\times$ total average speedup whereas a realistic algorithm selection—where sequential with higher cardinalities is evaluated using ***partially sorted monotable***—yields $4.15\times$. This slowdown is a mere 1.3%.

VI. Related Work

Here we discuss the related work. We separate this section into two subsections. The first looks at work related to parallel data aggregation acceleration. The second looks at alternative hardware proposals that attempt to tackle irregular DLP vectorisation.

A. Parallel Aggregation Acceleration

Zhou and Ross [24] explore the implementation of DBMS algorithms with basic SIMD multimedia instruction extensions. They only mention GROUP BY aggregation in passing and do not find a way to implement it with SIMD instructions unless first sorting the input. We have seen that fully sorting the input has major overhead for radix sort—an evasion algorithm—and even a significant overhead for VSR sort—a fast confrontation algorithm.

Ye et al. [19] evaluate how various aggregation methods scale with multiple threads. Although multithreading and SIMD are not completely comparable, they do observe some similar behaviour found in our experiments. We implement table replication to avoid GMS conflicts between vector register elements; Ye et al. do the same for read-modify-write conflicts that can occur between threads. Similar to our observations, they observe a massive loss in performance when cardinalities exceed the L1-d cache size. In general, our motivation is such that vector SIMD acceleration is more efficient than multithreading. We achieve $7.6\times$ speedup in some cases using a single vector unit whereas to achieve this result using multithreading would require—at minimum—eight cores. That said, vector SIMD and multithreading are not mutually exclusive and can complement each other nicely with the right algorithm design.

Polychroniou and Ross [20] propose SIMD optimisations when aggregating datasets similar to the zipf and hhitter datasets. Their approach uses multimedia SIMD extensions, although in a very different way to our vector SIMD instructions. Where we use a struct-of-arrays model and completely vectorise our algorithms, they use an array-of-structs model and partially vectorise their algorithm in an orthogonal direction. In all of our vector implementations, we vectorise with `n`, i.e. along the output arrays `count` and `sum`. Polychroniou and Ross instead pack each `count[i]` and `sum[i]` adjacently in SIMD registers to process both together. Although this approach offers some benefits, it has limited applicability and little scalability since the amount of parallelism depends on the number of aggregation operations in the query. In many cases this will be one and, therefore, offers no advantage over a scalar algorithm. Additionally, the number of aggregations and their datatypes in a query may not be available until runtime time thereby adding an obstacle when defining the appropriate memory structures. This scrutiny is not a criticism of their work, but instead an observation on the limitations when using simple multimedia extensions to vectorise complex algorithms like data aggregation. Our work uses a true vector SIMD ISA and we vectorise our algorithms in the direction of the arrays. This type of vectorisation is very beneficial for column-store databases—typically used in analytical processing—which favour a struct-of-arrays model over an array-of-struct model [21].

Power et al. [25] utilise GPGPUs to perform data aggregation. They argue that when using discrete off-chip GPUs, there is a high overhead associated with data movement as well as the coordination between the CPU/OS and the GPU. They motivate using integrated GPUs, i.e. GPUs on the same die as the CPU. They present two techniques. The first approach uses a replicated table for each GPU thread. The second approach uses a single lock-free table with the GPU threads repeatedly trying to perform atomic read-modify-write updates. The first technique shows benefits for very low cardinalities whereas the second approach proves to be good for very high cardinalities. For middle/normal cardinalities, neither approach works very well. While GPU and vector hardware organisations are not directly comparable, we do see some commonalities between the techniques used to parallelise aggregation. Their first technique is very similar to our ***polytable*** method, and, congruous to our observations, they also find table replication causes rapid performance deterioration as cardinality increases. Their second technique is similar to our ***monotable*** approach in that we both try to update a single table with potential GMS conflicts. Power et al. propose using atomic memory instructions, however, they find that contention is too frequent to achieve good performance if the cardinality is not very high. Our ***monotable*** method does not use such instructions; instead, we rectify any conflicting operations that would cause GMS conflicts in the registers before even making the memory access. We show experimentally that this is useful for a variety of data distributions and cardinalities.

B. Hardware Support for Irregular DLP

Scatter-add [26] is a proposal for streaming architectures that allows a conflict-free gather-modify-scatter operation on an array using one instruction. There are several significant differences with our proposal. The first and foremost is that scatter-add cannot be used to implement VSR sort. There are two reasons for this—(1) It lacks a return path for original values in the array before the modifications, and (2) it lacks the deterministic ordering semantics found in `VPI` and `VLU`. Scatter-add, therefore, has limited applicability to our proposed algorithms in which partially sorting is a major component. Secondly, scatter-add is an extension to the processor's memory hierarchy. Adding a major feature to memory can be less modular, highly intrusive and more difficult to verify. In contrast, the `VGAx` instructions use only vector registers as input and output. Thirdly, although scatter-add is beneficial in the sense that a single instruction expresses a lot of work, the same behaviour can be emulated by `VGAsum`. Since the `VGAx` instructions generate a running cumulative for each group in a vector register, this could have uses beyond aggregation, e.g. a customised prefix sum operation. Finally, we are building upon hardware that is already in place for the instructions `VPI` and `VLU`. The addition required to implement the `VGAx` instructions are minor.

Atomic vectors operations [27], and more recently, AVX512-CDI [6], are both solutions from Intel that attempt to solve the problem of GMS conflicts. Both of these proposals operate with a best-effort mechanism as follows. A mask register with all its bits set is coupled with the vector GMS procedure. The processor attempts to execute as many non-conflicting elements of the procedure as it can and clears the associated mask bits of successful outcomes. The programmer is responsible for placing the GMS procedure inside a loop that is dependent on the state of the mask register. This means in the worst case scenario the operation will be completely serialised inside a loop with a difficult to predict exit condition. Since each retry requires loading, modifying and storing the data again, it could even lead to more operations than its scalar counterpart. We anticipate that for datasets with low cardinalities and skewed distributions, the number of retries will be high and thus impede performance. This problem will be exacerbated further as vector SIMD register widths increase. `VPI`, `VLU` and the `VGAx` instructions are different because they exist as self-contained non-memory instructions. This difference means GMS conflicts are resolved completely and deterministically before committing to the memory hierarchy. We have shown experimentally that datasets with low cardinalities and skewed distributions perform well with a large MVL. Furthermore, it is not obvious how VSR sort could be constructed from either the atomic vector operations or AVX512-CDI. We have demonstrated that partially sorting the input using VSR sort for high cardinalities provides major performance improvements and, consequently, remains an important part of this work.

VII. Conclusions

As the amount of data increases exponentially each year it is important that data aggregation algorithms can scale accordingly. In this work, we have looked at vector SIMD instructions as a means to accelerate GROUP BY data aggregations. We have found that this is not a trivial target due to the irregularity of the DLP.

We have made experiments with a sophisticated vector SIMD ISA that we anticipate appearing in future microprocessor generations. We have found that this ISA has limitations since it only permits us to evade the irregular DLP through performance-degrading algorithm transformations. Based on this realisation, we have proposed the use of novel vector instructions which directly confront this irregularity and allow us to vectorise the algorithms directly without alteration. We have made detailed evaluations using multiple algorithms taken from both *evasion* and *confrontation* techniques.

We have observed that the evasion techniques have limited applicability unless the input is presorted, otherwise, the confrontation techniques prove to be more advantageous. The latter draws heavily from recent work on vectorised sorting algorithms that tackles irregular DLP through novel vector instructions. We have discovered that these novel instructions—and their associated sorting algorithm—can aid data aggregation, especially with the realisation that the input need not be fully sorted. With minimal modifications, we have extended the base hardware used in this proposal to accommodate data aggregation further by defining a suite of new instructions called `VGAx`.

We have found that the best algorithm depends highly on both the distribution and cardinality of the input. In most cases, this can be detected at runtime to make a choice dynamically. Using a combination of these techniques, we have achieved speedups over a scalar baseline between $2.7\times$ and $7.6\times$ for a maximum vector length of 64 and four lockstepped lanes.

VIII. Acknowledgements

The research leading to these results has received funding from the RoMoL ERC Advanced Grant GA nº 321253 and is supported in part by the European Union (FEDER funds) under contract TTIN2015-65316-P. Timothy Hayes is supported by a FPU research grant from the Spanish MECD.

REFERENCES

[1] M. Chen, S. Mao, and Y. Liu, "Big Data: A Survey," *Mobile Networks and Applications*, vol. 19, no. 2, pp. 171–209, 2014.

[2] P. Boncz, T. Neumann, and O. Erling, "TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark," in *Performance Characterization and Benchmarking*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8391, pp. 61–76.

[3] M. Bohr, "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper," *Solid-State Circuits Society Newsletter, IEEE*, vol. 12, no. 1, pp. 11–13, 2007.

[4] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the Tradeoffs Between Programmability and Efficiency in Data-parallel Accelerators," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA, 2011, pp. 129–140.

[5] J. Gebis and D. Patterson, "Embracing and Extending 20th-Century Instruction Set Architectures," *Computer*, vol. 40, no. 4, pp. 68–75, 2007.

[6] *Intel®Architecture Instruction Set Extensions Programming Reference*, Intel, March 2014.

[7] R. Espasa, M. Valero, and J. E. Smith, "Vector Architectures: Past, Present and Future," in *Proceedings of the 12th International Conference on Supercomputing*, ser. ICS, 1998, pp. 425–432.

[8] T. Hayes, O. Palomar, O. Unsal, A. Cristal, and M. Valero, "VSR Sort: A Novel Vectorised Sorting Algorithm & Architecture Extensions for Future Microprocessors," in *21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 26–38.

[9] M. Yourst, "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator," in *IEEE International Symposium on Performance Analysis of Systems Software*, ser. ISPASS, 2007, pp. 23–34.

[10] *Intel®64 and IA-32 Architectures Optimization Reference Manual*, Intel, March 2014.

[11] T. Hayes, O. Palomar, O. Unsal, A. Cristal, and M. Valero, "Vector Extensions for Decision Support DBMS Acceleration," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO, 2012, pp. 166–176.

[12] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE Computer Architure Letters*, vol. 10, no. 1, pp. 16–19, Jan. 2011.

[13] R. Espasa, M. Valero, and J. E. Smith, "Out-of-Order Vector Architectures," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO, 1997, pp. 160–170.

[14] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec, "Tarantula: A Vector Extension to the Alpha Architecture," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ser. ISCA, 2002, pp. 281–292.

[15] F. Quintana, J. Corbal, R. Espasa, and M. Valero, "Adding a Vector Unit to a Superscalar Processor," in *Proceedings of the 13th International Conference on Supercomputing*, ser. ICS, 1999, pp. 1–10.

[16] B. R. Rau, "Pseudo-randomly Interleaved Memory," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ser. ISCA. ACM, 1991, pp. 74–83.

[17] A. González, M. Valero, N. Topham, and J. M. Parcerisa, "Eliminating Cache Conflict Misses Through XOR-based Placement Functions," in *Proceedings of the 11th International Conference on Supercomputing*, ser. ICS '97. ACM, 1997, pp. 76–83.

[18] J. Cieslewicz and K. A. Ross, "Adaptive Aggregation on Chip Multiprocessors," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 339–350.

[19] Y. Ye, K. A. Ross, and N. Vesdapunt, "Scalable Aggregation on Multicore Processors," in *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, ser. DaMoN '11. ACM, 2011, pp. 1–9.

[20] O. Polychroniou and K. A. Ross, "High Throughput Heavy Hitter Aggregation for Modern SIMD Processors," in *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, ser. DaMoN '13. ACM, 2013, pp. 6:1–6:6.

[21] D. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden, "The Design and Implementation of Modern Column-Oriented Database Systems," *Foundations and Trends in Databases*, vol. 5, no. 3, pp. 197–280, 2013.

[22] L. M. Ni and K. Hwang, "Vector Reduction Methods for Arithmetic Pipelines," in *IEEE 6th Symposium on Computer Arithmetic (ARITH)*, June 1983, pp. 144–150.

[23] M. Zagha and G. E. Blelloch, "Radix Sort for Vector Multiprocessors," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. SC, 1991, pp. 712–721.

[24] J. Zhou and K. A. Ross, "Implementing Database Operations Using SIMD Instructions," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '02. ACM, 2002, pp. 145–156.

[25] J. Power, Y. Li, M. D. Hill, J. M. Patel, and D. A. Wood, "Toward GPUs Being Mainstream in Analytic Processing: An Initial Argument Using Simple Scan-aggregate Queries," in *Proceedings of the 11th International Workshop on Data Management on New Hardware*, ser. DaMoN'15. ACM, 2015, pp. 11:1–11:8.

[26] J. H. Ahn, M. Erez, and W. J. Dally, "Scatter-Add in Data Parallel Architectures," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, ser. HPCA, 2005, pp. 132–142.

[27] S. Kumar, D. Kim, M. Smelyanskiy, Y.-K. Chen, J. Chhugani, C. J. Hughes, C. Kim, V. W. Lee, and A. D. Nguyen, "Atomic Vector Operations on Chip Multiprocessors," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA, 2008, pp. 441–452.