ViViD Cuckoo Hash: Fast Cuckoo Table Building in SIMD

Flaviene Scheidt de Cristo, Eduardo Cunha de Almeida, Marco Antonio Zanata Alves

[1]Informatics Department – Federal Univeristy of Paraná (UFPR) – Curitiba – PR – Brazil

{fscristo,eduardo,mazalves}@inf.ufpr.br

Abstract. *Hash Tables play a lead role in modern databases systems during the execution of joins, grouping, indexing, removal of duplicates, and accelerating point queries. In this paper, we focus on Cuckoo Hash, a technique to deal with collisions guaranteeing that data is retrieved with at most two memory access in the worst case. However, building the Cuckoo Table with the current scalar methods is inefficient when treating the eviction of the colliding keys. We propose a Vertically Vectorized data-dependent method to build Cuckoo Tables - ViViD Cuckoo Hash. Our method exploits data parallelism with AVX-512 SIMD instructions and transforms control dependencies into data dependencies to make the build process faster with an overall reduction in response time by 90% compared to the scalar Cuckoo Hash.*

1. Introduction

The usage of hash tables in the execution of joins, grouping, indexing, and removal of duplicates is a widespread technique on modern database systems. In the particular case of joins, hash tables do not require nested loops and sorting, dismissing the need to execute multiple full scans over the same relation (table). However, a hash table is as good as its strategy to avoid or deal with collisions. Cuckoo Hash [Pagh and Rodler 2004] stands among the most efficient ways of dealing with collisions. Using open addressing, it does not use additional structures and pointers - as Chained Hash [Cormen et al. 2009] does - and assures only two memory access to retrieve a key on the worst case.

The Cuckoo hashing method is built with two tables, each one indexed by a different hash function. In a metaphor with the cuckoo bird behavior, when a collision occurs during insertion, the older key is evicted from the *nest*[1] it currently holds and rehashes to another one in a second table. The new key now takes the old one's position. This eviction process repeats until a key finds an empty nest or a given threshold is reached. In this second case, the table must be rebuilt using new hash functions.

All these evictions are costly when building a Cuckoo table, and even more, when we perform a bulk insert of all the keys from a given relation, which will happen during the Join execution. We hypothesize that the use of parallelism could reduce the bottleneck caused by the eviction process. We experimented different levels and dimensions of parallelism, and some other developments described in the literature, such as the transformation of control structures into logical operations. The result is the *ViViD Cuckoo Hash*, a

[1]Using the analogy of the cuckoo bird, a key is a bird, and the bucket it occupies is the nest. The female cuckoo bird ejects old eggs from nests to set her own.

This study was partially supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001 and by the Serrapilheira Institute (grant number Serra-1709-16621).

vertically vectorized data-dependent method to build Cuckoo Tables. Experimental evaluations showed that our ViViD Cuckoo Hash technique has 10× higher throughput on average than the scalar method, maintaining the same power consumption profile.

2. Hash Join Landscape and Literature Review

The join operation is presented by the relational algebra theory to combine data from different relations (i.e., two tables, for instance). Taking into account the formal definition from [Ramakrishan and Gehrke 2003], a natural join between two relations $R(A_1, A_2, ..., A_n)$ and $S(B_1, B_2, ..., B_n)$ is denoted $R \bowtie S$. The natural join, or the general definition of a relational join operation, is a binary operation to combine certain selections (or filters on attributes $\sigma_{R.A_n=S.B_n}$), a Cartesian product $R \times S$ into one operation and produces a new relation with headers (or columns) $R \cup S$, as follows:

$$R \bowtie S = \Pi_{R \cup S}(\sigma_{R.A_1=S.B_1 \wedge ... \wedge R.A_n=S.B_n}(R \times S))$$

In practice, to perform a join between two relations, we need to compare each key in the first relation with every key in the second relation. A naive method is to nest two loops being R and S the relations we are joining and T the resultant relation, also called Nested Loop Join.

In databases, join is the most expensive operation regarding time and resources. Query processing requires combining keys from the inner relation with those on the outer relation. The number of comparisons between keys is a trivial operation for the processor, while access to memory is an expensive operation, figuring as the bottleneck of most join methods. In this paper, we focus on Hash Join, that indexes keys using hash functions to build dictionaries, avoiding excessive table scans.

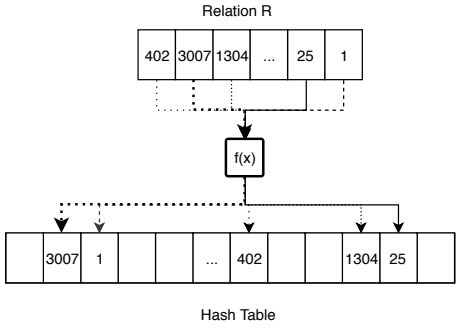


Figure 1. The construction of a hash table. Each key passes through a hash function f(x), the result indicates the index where the key must be stored.

Hashing methods are separated into two big groups according to the way they deal with collisions. A collision occurs when two different keys hash to the same position on the table. Note that when we try to insert a key that is already on the table, we have duplicity of keys and not a collision. The first method to deal with collisions is the traditional Chained Hash [Ramakrishan and Gehrke 2003], and the second one is the Open Addressing, which is the most interesting when it comes to probing speed and better memory usage.

Previous work already addressed a complete and widespread vision of the main hashing methods used nowadays [Richter et al. 2015]. The authors analyze the impact of

each hashing method in some databases operators: Chained Hash, Linear, and Quadratic Probing and two more sophisticated methods: Robin Hood Hash and Cuckoo Hash.

The Robin Hood Hash was introduced in 1985 coining the concept of *poor* and *rich* keys [Celis et al. 1985]. The closer a key is stored to its index in the hash table, the richer it is. When a collision occurs, a richer key is reallocated to give its spot to the poor one. When using open addressing to deal with hash collisions, the worst-case scenario occurs when the data gets clustered into certain regions, causing a chain of memory access far from its original hash bucket. Robin Hood Hash amortize the costs of the worst-case by switching keys to bring them close to the bucket where they should be. However, it does not offer much improvement in the build phase; we still have to deal with the costs of relocating keys.

Another attractive method - not disclosed on Richter's work - is the Hopscotching Hash [Herlihy et al. 2008]. Hopscotch is a hybrid technique between Linear Probing and Cuckoo Hashing. It defines a neighborhood of buckets where a key may be found. It is an advance on what concerns Linear Probing, Robin Hood, and even the Cuckoo Hash and works well when the load factor grows beyond 90%. However, when no empty bucket is found, the algorithm traverses the buckets sequentially, causing overhead on the building phase of the join.

The most straightforward and powerful method remains the Cuckoo Hashing [Pagh and Rodler 2004]. The most relevant development on Cuckoo Hashing related to the present paper is the conversion of *control dependencies* into *data dependencies* [Zukowski et al. 2006], [Ross 2007]. Regarding Cuckoo Hashing, previous work built fast concurrent Cuckoo tables by narrowing the critical section and decreasing interprocessor traffic [Li et al. 2014]. Others also suggested an optimistic approach of a lock-free Cuckoo table [Nguyen and Tsigas 2014].

We can observe that several papers treat the vectorization of Cuckoo Hashing, for instance, presenting an excellent view of a semi vertically vectorized probe [Zukowski et al. 2006]. Others suggest a vertical vectorization for the Cuckoo Hash, but focus on the probing, not on the building of the table [Ross 2007]. A robust approach was also proposed, describing the vertical vectorization and the removal of most control dependencies both on the build and the probe phase, implemented both in Intel Knights Landing, Haswell and Sandy Bridge microarchitectures [Polychroniou et al. 2015].

Our work also focuses on the vectorization of the Cuckoo Hash, bringing improvements over previous approaches. By maximizing the usage of logical expressions, we take advantage of the most recent AVX-512 (using 512 bits registers) capabilities - such as inline collision detection and operations with masks - and by applying fast hash functions that guarantee minimum collision rates. We also analyzed the impact of the usage of some AVX-512 capabilities - such as bitmask and scatter operations - versus an AVX-256 (using 256 bits registers) implementation that tries to emulate those capabilities.

3. Our Proposal: ViViD Cuckoo Hash Join

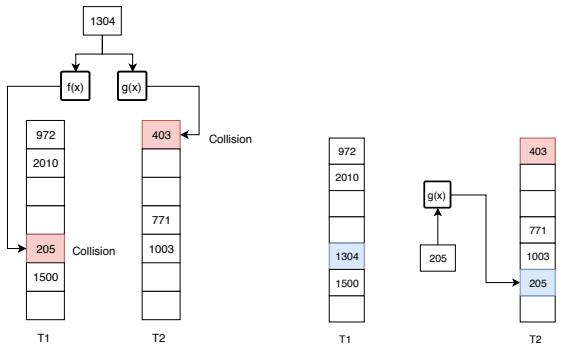
A join operation between two relations is the most expensive operation in the query processing [Silberschartz et al. 2006]. However, most of this cost can be mitigated by the

use of hash tables. Cuckoo Hash is the most efficient method to build and probe keys across these tables. In practice, we build a Cuckoo Table every time we perform a join to avoid consistency issues and unnecessary duplication of the data, and we call this join processing, as Cuckoo Join.

3.1. Concurrent and Vectorized Cuckoo Hashing

Out first attempt to decrease the time consumed by the build phase of the Cuckoo Join is to use multiple workers, each one of them simultaneously performing reads and writes. The synchronization of the workers is first maintained by locking the critical regions trying to keep these areas at a minimum size, similarly to previous work [Li et al. 2014].

However, shrinking the critical areas is not straightforward, as the *cuckoo path* of a key can collide with the path of another one; both inserted simultaneously. The *Cuckoo path* is the sequence of evicted keys [Nguyen and Tsigas 2014]. Every inserted key has its *cuckoo path*. Figure 2 shows the insertion of the key 1304, that causes the eviction of the key 205. In this case, the *cuckoo path* P of the key 1304 has only one member - 205 - so P_{1304}={205}, if the key 205 had to evict another key, let's say 42, so P_{1304}={205, 42}. Now, if we try to insert simultaneously a second key 506 with P_{506}={309, 42, 2000}, the paths collide and lead to inconsistencies in the final table.



(a) Collision on the insertion of the key 1304.

(b) Relocation of the key 205.

Figure 2. The key 205 is deallocated and must be rehashed to be be reinserted on the table 1. The set of the keys deallocated to successfully inserted a given key is called cuckoo path.

The immediate answer to deal with path collisions is to lock all the insertion as a critical region because we cannot support a read operation while reallocating keys. It turns the concurrent insert into a serial process, adding the overhead of dealing with context switches, cache invalidation and consistency between workers.

Using the *cuckoo path* to narrow the critical region [Li et al. 2014], and forecasting path collisions to serialize the insertion only when needed are efficient strategies for tables that suffer inserts and lookups at the same time. However, they add the overhead of calculating the path. Then, using multiple workers to build the table is not the ideal approach.

The next method to speed up the build phase of the Cuckoo Join is the vectorization of the operation. Our first attempt is to bucketize the table, each bucket holding eight

keys of 32-bits unsigned integers allowing the use of SIMD registers to perform lookups. Although this model is efficient for the probe phase, it does not satisfy the build process that has to remain scalar. Our solution came to be the vertical vectorization - against the horizontal one of the previous method. Instead of using buckets, we used vectors of 256 and 512 bits of length to insert multiple keys at the same time.

3.2. About Branches and Predictions

The insertion of a key in a Cuckoo Table involves a loop that goes on until an empty spot is found, or a certain threshold of evictions is reached. The execution flow of the conditional statements performed is unpredictable. By definition, each key has the same probability of hashing to any bucket independently of other keys [Cormen et al. 2009]. It causes overhead since, for each wrong branch taken preemptively, the processor has to undo all the instructions. Previous work already discussed this issue by transforming the *control dependency* into *data dependency* (i.e., transforming the control flow structures into logical operations) [Polychroniou et al. 2015], [Ross 2007], although not so extensively as our method does.

3.3. ViViD Cuckoo Hash Implementation

We created the *Vertically Vectorized Data Dependent Cuckoo Hash* (ViViD Cuckoo) by transforming most of the *control dependencies* into *data dependencies* and using SIMD registers to vectorize the build and the probe vertically. With ViViD Cuckoo we could address the two main issues that caused the build to become a bottleneck for the Cuckoo Join: the chained unpredictable control dependencies and the data dependencies between cuckoo paths. We extended the methods described by [Polychroniou et al. 2015] and [Polychroniou and Ross 2014] to use new capabilities of AVX-512 and to take advantage of logical operations and the dispersion provided by the hash functions.

The ViViD Cuckoo Hash can be implemented in two different ways, according to the AVX extension supported by the target microarchitecture. The first approach is a more general implementation of the method, using AVX-256 extensions. Intel introduced AVX-256 (or AVX-2) in 2013 in the Haswell family of processors extending the AVX architecture[2] to support 256-bit vectors operations and adding 30 new instructions to the set. AVX-256 does not support scattered loads from memory directly, so we emulated those operations by storing the keys and addresses in two C language vectors and then store the keys scalarly. The downside of this solution is the reminiscence of some branch structures.

The second implementation uses the AVX-512 extensions, that allow operations with 512-bit-vectors and also bring *scatter* operations, prefetching and opmasks. Intel MIC based co-processors, such as Xeon-Phi, have been already working with 512-bit vectors for a while, but they were not introduced on other Xeon families until the launching of the Skylake-X processor. AVX-512 extends and refines the 512-bit operations present on earlier Xeon Phi's architectures, but they are not compatible [Reinders 2017]. We could find one previous work that implemented vectorized Cuckoo Hashing on Xeon-Phi x-100 using 512-bit-vectors and operations [Polychroniou et al. 2015], but unfortunately, we could not compare the performance of their implementation using our metrics because of the incompatibility stated above.

[2]present on Intel processors since 2008, presented with the Sandy Bridge family

AVX-512 also has special registers to hold masks exclusively, that may be of 8, 16, 32 and 64-bit length. A whole set of instructions was added to deal solely with these mask registers, and almost all AVX-512 instructions operate with masks. AVX-256 has operations involving masks, but there are no special registers, so the masks are held on standard vector registers of 32-bit integers. We emulated the operation of masks instructions with logical operations.

The AVX-256 ViViD Cuckoo uses SIMD registers of 256-bits, holding eight keys of 32-bits unsigned integers each. As for the AVX-512, we implemented two versions, the first one with 256-bit registers and the second one with 512-bit registers. We used two tables stored sequentially with 262 K positions, each position holding a single key. It is possible to access the table using both scalar and vectorized methods.

Figures 3, 4 and 5 present three of the eight phases of the ViViD Cuckoo Hash. Each SIMD variable is a boolean mask or holds integer values that represent keys, hashed keys, and counters. We split the build process into 8 phases to better understand and present the insertion of three keys to exemplify a collision-free successful insertion, an insertion with collision and a duplicated key. The collision-free insertion is exemplified by the key 675. The second key is 543, that collides with the keys 954 and 786, respectively on the first and second table. The duplicated key is 9087, that is already present in the second table.

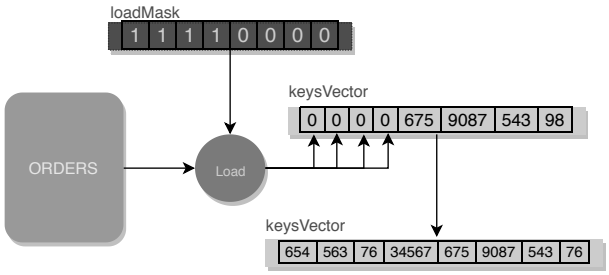


Figure 3. Load of keys from the relation Orders. News keys are loaded when the bit is set on the mask for that given position.

Phase 1 is the load of the keys; if it is the first iteration, eight or sixteen keys must be loaded, depending on the vector size. Otherwise, *loadMask* will indicate how many keys need to be loaded. Old keys occupying the positions indicated by the *loadMask* are not set, and the new keys are set. Figure 3 shows the load of four keys into a 256-bit vector that holds eight 32-bit keys. Only four keys are loaded because those are the positions set on *loadMask*. The remaining keys already on *keysVector* are the ones deallocated on the previous iteration. On the first run, since there are no keys in the table, all positions on *loadMask* are set and *keysVector* is empty and must be loaded entirely with new keys. The same occurs when all the keys find empty spots in the table on the previous iteration. The keys 675, 543 and 9087 are loaded on this iteration respectively on the fourth, sixth and fifth positions of *keysVector*.

Phase 2 is the hash, we used FNV1a [Fowler et al. 2011] for the first table and Murmur3 [Appleby 2016] for the second table. We choose Murmur3 and FNV1a hash functions based on the work of [Estébanez et al. 2014], that state Murmur3 and FNV1a as having the lowest collision rate between all the functions analyzed. These two hash

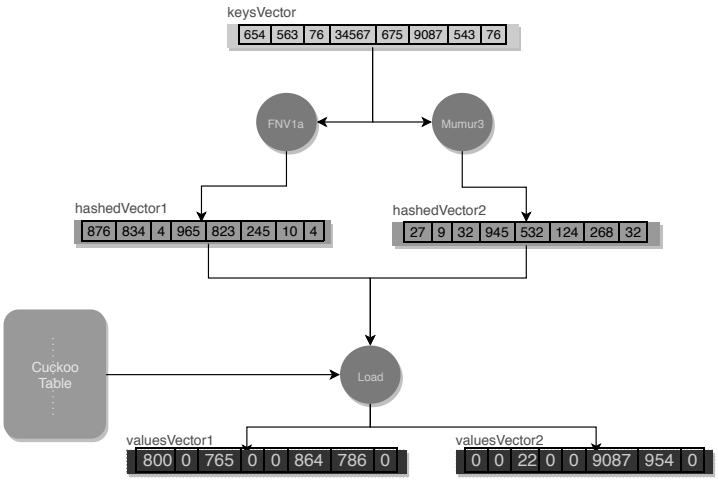


Figure 4. Phase 2: The keys to be inserted are hashed using two hash functions. Phase 3: We then gather the keys, already stored on the given positions, from both tables and store them in two vectors.

functions are also the most straightforward to vectorize and are fast to compute.

On **Phase 3** all the keys are hashed with both hash functions, and the values are retrieved from both tables to detect duplicated values on the next phase. Since the two sides of the Cuckoo table are stored as one unique table in memory, each side occupying half of the physical structure, we use logical operations to find the correct indexes.

Phase 4 involves the detection of duplicated keys, conflicts, and successful insertions. *Part 1* of Figure 5 presents the duplicates detection; we compare the values gathered with the keys being stored. If there is a value on *keysVector* already present in the Cuckoo table, we set the *remotionMask* on that position to identify that these values must not be stored. The key 9087 is already present in the second table, so we set the fifth position of the *remotionMask*, indicating that this key must not be stored. AVX-512 has a single instruction that detects conflicts inside the vector. We used this instruction on both AVX-512 implementations, even the in-vector collision being a relatively rare event; in fact, we did not detect any in-vector collisions for the workload used on our experimental evaluation. When a conflict occurs, the second occurrence of the key gets set on *remotionMask*, as shown in *Part 3* of Figure 5 on the example of the key 9087.

Part 2 of Figure 5 shows the detection of successful insertions. An insertion is considered successful if it finds an empty spot. In our implementation, we test only the value retrieved from the table assigned to the key on the current iteration, not looking to the other table. Most implementations of Cuckoo Hashing test both tables for empty spots to minimize unneeded deallocations. We opted for not implementing this way to avoid another calculation of the table assigned for the key; instead, we do an *or* operation between the values gathered from both tables using *table1Mask* to indicate the table from where the key will be deallocated. The key 675 is due to be stored on the first table on the position 823; since this position is empty, the insertion is successful without the need to reallocate any keys, so we set the fourth position of the *loadMask* to indicate that a new key will be loaded from the relation. The key 543 finds its position on the first table occupied by the key 786, that will be deallocated and will now occupy the sixth position

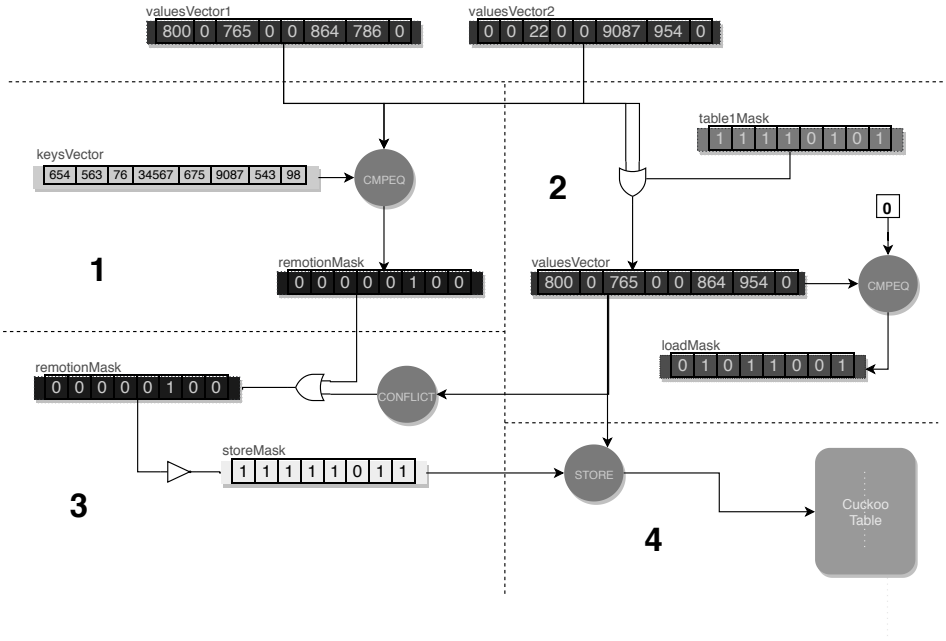


Figure 5. Phase 4 and a simplification of Phase 7. Duplicated values, conflicts and successful insertions detection and actual store of keys. Part 1 compares the keys gathered from the table with the keys we are trying to insert to detect duplicated keys. Part 2 selects the keys according to the table they must be stored and compares to zero, to find insertions that do not need reallocations. Part3 detects in-vector duplicated keys and Part 4 shows the insertion of keys on the Cuckoo table.

on *keysVector* to be inserted on the second table on the next iteration.

At the end of **Phase 4**, we have a *storeMask*, indicating all the values that will be stored, i.e., the values that are not duplicated. We use this mask to store the keys on the Cuckoo table. At this point, *loadMask* indicates all the position where the keys were successfully inserted or are duplicated. Those are the positions that receive new keys from the relation.

In **Phase 5**, we calculate the number of evictions already made to store each key. If the number of evictions (or ”hops” because we hop between table 1 and 2) is equal or greater than a given threshold, the table must be rebuilt. **Phase 6** calculates the table where each key must be stored in the next iteration. All the keys that hold an odd number of hops will be stored in table 1 in the next iteration, and the ones that have even number of hops goes to the second table. New keys loaded from the relation are always inserted in the first table. The sixth position of *keysVector* holds now the key 786, and the sixth position of *hopsVector* will indicate that a deallocation has already been made, as this vector indicates an odd number of hops, in the next iteration the on the sixth position will be stored on the second table.

Phase 7 is when we store the keys in the Cuckoo table. On the AVX-256 version, this store occurs sequentially with the use of two auxiliary C language vectors. On AVX-512 version we used the *scatter* operation. *Part 4* of Figure 5 shows this operation.

Phase 8 permutates the keys putting on the right side the keys that must remain on

keysVector for the next iteration, i.e., the keys gathered on **Phase 3** and are not duplicated or zero, in this case the key 9078. The left side receives all the keys that will be replaced for new ones from the relation, in the example the keys 675 and 76. This process is based on [Polychroniou and Ross 2014] approach using a permutation table kept in memory to speed the process.

4. Experimental Methodology

To verify the effectiveness of the vertical parallelism and the data dependency brought by the ViViD Cuckoo on the building of Cuckoo tables, we performed experiments analyzing the throughput of keys, power and energy consumption and memory behavior. All the experiments were compiled and executed using Linux Ubuntu 18.04.1 LTS Bionic Beaver on an Intel® Xeon® Silver 4114 at 2.20GHz based on Skylake-X architecture. All the code[3] was compiled with GCC 7.3.0 specifically for the Skylake-X micro-architecture using AVX-512 with -O2 optimization flag. The automatic vectorization provided by the compiler was also enabled for the scalar version. However, the logs showed almost no vectorization because of the many branches inside the loops.

The data was generated based on TPC-H Benchmark [Council 2008] (scale factor of 1) with some adaptations for varying the selectivity of the join. All the results presented are for the build phase of the Cuckoo Join, since previous works have already studied the probe phase [Ross 2007], [Zukowski et al. 2006].

The query used in the experiments (below) comprehends an *anti-join* operator that can be understood as an implicit *exclusive left join*. We implemented this anti-join based on the strategy taken by the PostgreSQL optimizer [4]. This anti-join creates the hash table based on the relation *Orders* instead of *Customer*, as it would be done if the optimizer chose a left join.

```
1 SELECT C_ACCTBAL
2 FROM CUSTOMER
3 WHERE NOT EXISTS (SELECT * FROM
4 ORDERS WHERE O_CUSTKEY = C_CUSTKEY);
```

Query used on the experiments

The metrics analyzed are throughput, power and energy consumption, and cache bandwidth between L2 and L1. All the metrics were collected from the Intel *model-specific registers* (MSR) and *running average power limits* (RAPL) registers using the Likwid wrapper [Treibig et al. 2010]. Those registers measure hardware events [Intel® 2011] that are then read by the Likwid wrapper and reported on a readable form, giving the maximum value, the minimum value and the average value of each metric for each execution. All the values plotted are the average values of each metric for 101 executions. The Likwid wrapper was also used to pin the process to a single core, avoiding issues caused by interprocessor traffic and cache coherence.

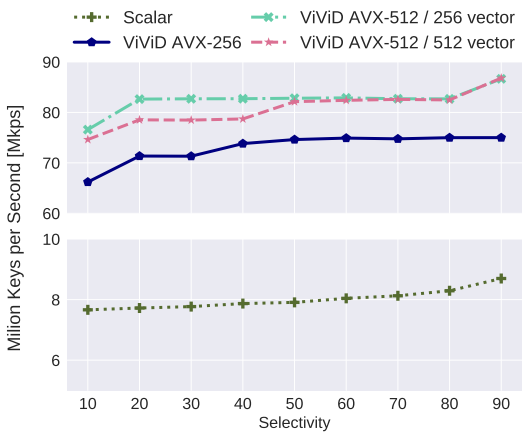
5. Experimental Results

For our results we considered three different implementations of the ViViD Cuckoo - AVX-256 using 256-bit vector and AVX-512 using 512-bit and 256-bit-vectors - and the

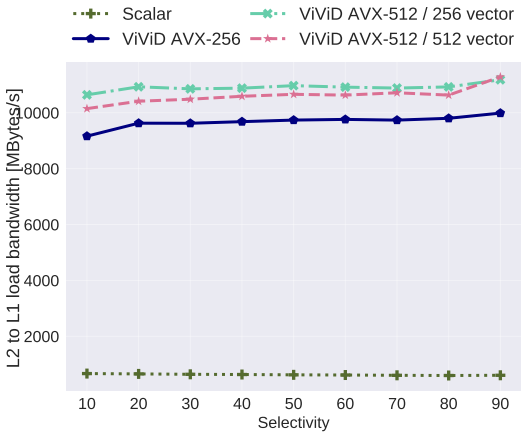
[3] Available at https://github.com/FlavScheidt/MHaJoL

[4] PostgreSQL v.11.2 Optimizer: https://www.postgresql.org/docs/current/planner-optimizer.html

single scalar implementation. Figure 6(a) shows the throughput in million keys per second when building a Cuckoo table using the scalar Cuckoo hashing and ViViD Cuckoo implemented with AVX-256 and AVX-512, varying the selectivity of the join operation. It is important to note that the number of duplicated keys contract according to the growth of the selectivity to maintain the same size of the relations for each selectivity. The ViViD Cuckoo in any version improved the throughput in almost 10× on average. The AVX-256 version had the poorest throughput, but still showed an average improvement of 8× over the scalar version. For selectivities bellow 50%, the 256-bit vector version of the AVX-512 implementation showed the same throughput as the 512-bit vector version when the selectivity is equal or greater than 50%. This happens because the experiments were performed on a Skylake-X based processor, which uses the AVX-512 ISA extension. This ISA extension implements 256-bit registers by using the lower 256 bits of the first sixteen 512-bit registers [Intel® 2019].



(a) Throughput of the build phase measured in a million keys per second (MKPS) using AVX-256, AVX-512, and the scalar method. The upper plot shows the throughput of the vectorized versions; the bottom plot shows the throughput of the scalar method.

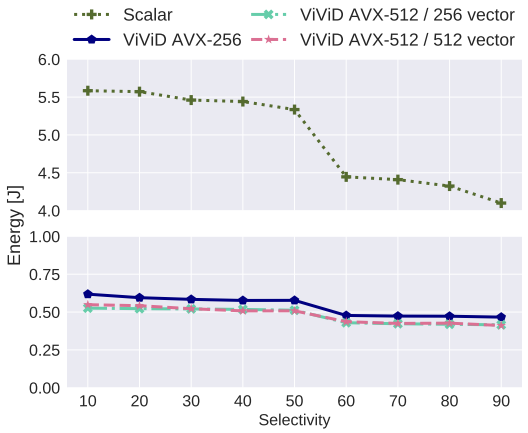


(b) Load bandwidth of data from L2 to L1 cache on the build of a Cuckoo Table.

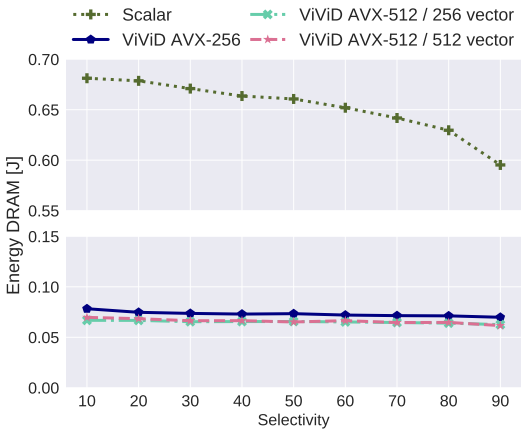
Figure 6. Throughput and Load Bandwidth.

This improvement of 10× on the average throughput of keys occurred because the 8 or 16 keys stored each time did not depend on their neighbors *path* during the insertion. There is just one step to verify collisions within the same iteration. Without the control dependencies, throughput improved without the overhead caused by the wrong predictions made by the branch predictor.

Vectorization improves energy efficiency by reducing the number of instructions occupying the pipeline, but increases the pressure on the whole memory hierarchy, since the request of data at each cycle is higher than on scalar executions [Cebrián et al. 2014]. This pressure increased power consumption for the memory system. Figure 7 shows that the absolute amount of energy spent by our three implementations of ViViD Cuckoo is 10× smaller than the scalar version when considering the chip package or only the DRAM memory. This follows the trend of the execution time, which also achieved great performance results.



(a) Energy consumed by the package in Joules on the build of a Cuckoo table.



(b) Energy consumed by the DRAM in Joules on the build of a Cuckoo table.

Figure 7. Energy consumed by the Package and the DRAM.

Figure 6(b) show the bandwidth between L2 and L1. The scalar Cuckoo Hash shows the lowest values, while the three ViViD variants present the highest. As stated in the previous subsection, vectorization shrinks the pressure over the pipeline but increases the pressure over the memory hierarchy. Wider registers mean more data required by each instruction, and a higher volume of data across the cache levels and the DRAM.

Between the three ViViD variants, the AVX-256 has the lowest bandwidth, as expected considering the size of the vector. However, the highest bandwidth belongs to the AVX-512 with 256 bits vectors, the same vector size as AVX-256. It should be expected the 512-bit vector variant to show the highest pressure, however, we could also notice that the functional unit bandwidth when using 512 bits vector increases compared to 256 bits, reducing thus the pressure on the memory controller.

6. Conclusions and Future Work

The ViViD Cuckoo Hash is a fast way to build Cuckoo Tables to perform hash joins. It uses SIMD vertical vectorization and transforms most of the control dependencies into data dependencies. Experiments conducted on the Skylake micro-architecture using AVX-256 and AVX-512 registers and operations show that our method improves the throughput of the build ten times on average compared with the scalar Cuckoo Hashing described in the literature. We were also able to maintain the power consumption of the package at the same as the scalar method while greatly reducing the total energy consumption. However, ViViD Cuckoo increased the pressure over the memory system, since wider SIMD registers mean that more data is needed to perform each instruction.

ViViD Cuckoo enhanced the performance of the build phase of Cuckoo tables, maintaining the same power consumption of the scalar method and 10× less energy on average. Future work is required on applying the ViViD method on different hash strategies - such as Robin Hood and Hopscotching - and compare the results with the Cuckoo Hash version. It would also be interesting to investigate the impact of using ViViD Cuckoo Hash on the build of Cuckoo Filters, comparing with the already existent vectorized versions of Bloom Filters.

References

Appleby, A. (2016). Smhasher. `https://github.com/aappleby/smhasher/`. Accessed: 2018-03-22.

Cebrián, J. M., Natvig, L., and Meyer, J. C. (2014). Performance and energy impact of parallelization and vectorization techniques in modern microprocessors. *Computing*, 96(12).

Celis, P., Larson, P.-A., and Munro, J. I. (1985). Robin hood hashing. In *SFCS*.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press.

Council, T. P. P. (2008). Tpc-h benchmark specification.

Estébanez, C., Saez, Y., Recio, G., and Isasi, P. (2014). Performance of the most common non-cryptographic hash functions. *Software: Practice and Experience*, 44(6).

Fowler, G., Noll, L. C., Vo, K.-P., Eastlake, D., and Hansen, T. (2011). The fnv non-cryptographic hash algorithm. *Ietf-draft*.

Herlihy, M., Shavit, N., and Tzafrir, M. (2008). Hopscotch hashing. In *DISC*.

Intel® (2011). *Intel® 64 and ia-32 architectures software developer's manual*. Intel®.

Intel® (2019). *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel®.

Li, X., Andersen, D. G., Kaminsky, M., and Freedman, M. J. (2014). Algorithmic improvements for fast concurrent cuckoo hashing. In *ACM EuroSys*, page 27.

Nguyen, N. and Tsigas, P. (2014). Lock-free cuckoo hashing. In *ICDCS*, pages 627–636.

Pagh, R. and Rodler, F. F. (2004). Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144.

Polychroniou, O., Raghavan, A., and Ross, K. A. (2015). Rethinking simd vectorization for in-memory databases. In *ACM SIGMOD*, pages 1493–1508.

Polychroniou, O. and Ross, K. A. (2014). Vectorized bloom filters for advanced simd processors. In *ACM Damon*, page 6.

Ramakrishan, R. and Gehrke, J. (2003). *Database Management Systems*. McGraw-Hill, 8th edition.

Reinders, J. (2017). Intel® avx-512 instructions.

Richter, S., Alvarez, V., and Dittrich, J. (2015). A seven-dimensional analysis of hashing methods and its implications on query processing. *PVLDB*, 9(3).

Ross, K. A. (2007). Efficient hash probes on modern processors. In *IEEE ICDE*, pages 1297–1301.

Silberschartz, A., Korth, H. F., and Surdashan, S. (2006). *Database System Concepts*. McGraw-Hill, 5th edition.

Treibig, J., Hager, G., and Wellein, G. (2010). Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *IEEE ICPPW*.

Zukowski, M., Héman, S., and Boncz, P. (2006). Architecture-conscious hashing. In *ACM Damon*, page 6.