

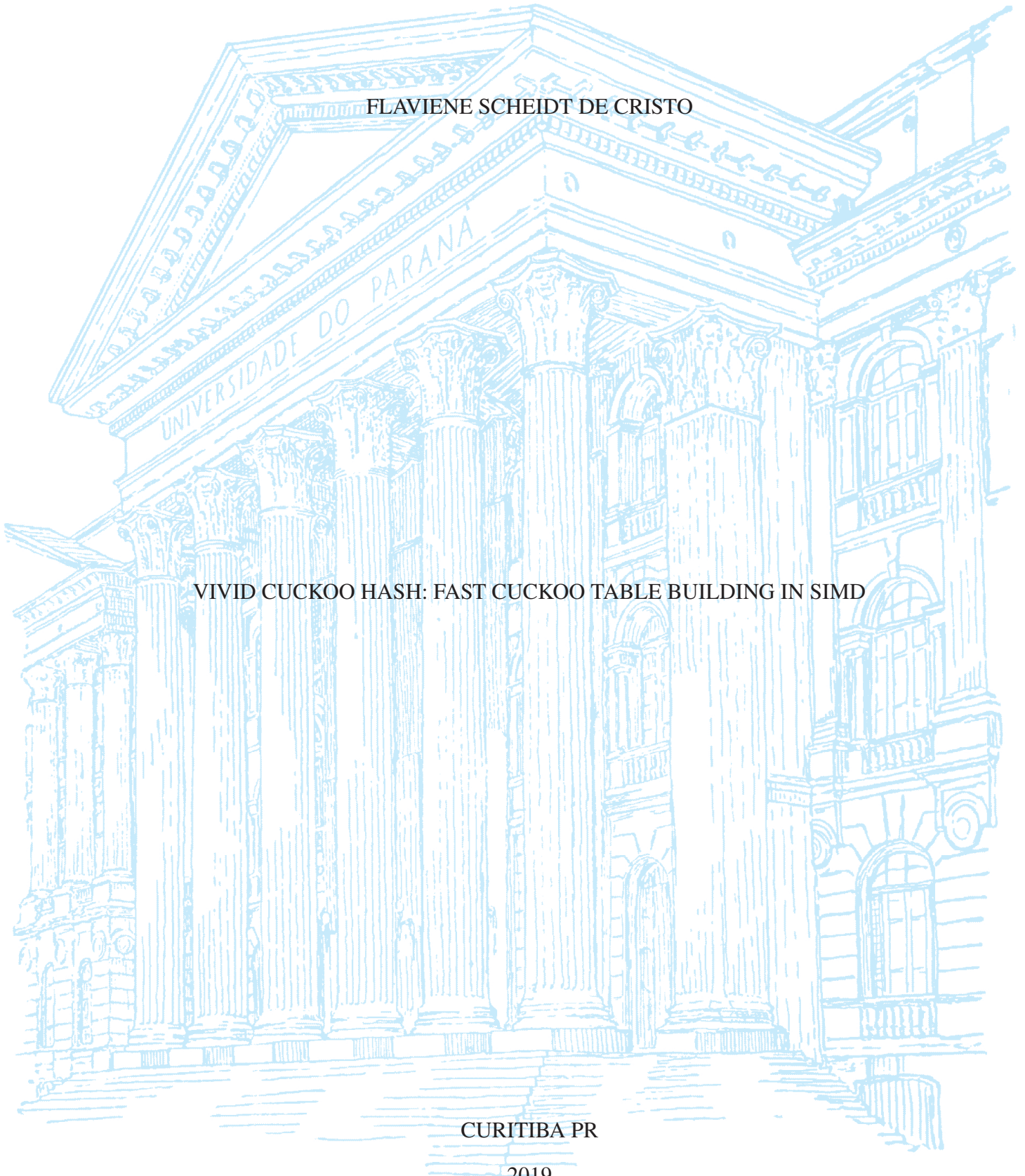
UNIVERSIDADE FEDERAL DO PARANÁ

FLAVIENE SCHEIDT DE CRISTO

VIVID CUCKOO HASH: FAST CUCKOO TABLE BUILDING IN SIMD

CURITIBA PR

2019



FLAVIENE SCHEIDT DE CRISTO

VIVID CUCKOO HASH: FAST CUCKOO TABLE BUILDING IN SIMD

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Eduardo Cunha de Almeida.

Coorientador: Marco Antonio Zanata Alvez.

CURITIBA PR

2019

Catálogo na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

C933v

Cristo, Flaviene Scheidt de

Vivid cuckoo hash: fast cuckoo table building in SIMD / Flaviene Scheidt de Cristo. – Curitiba, 2019.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2019.

Orientador: Eduardo Cunha de Almeida. Coorientador: Marco Antonio Zanata Alvez.

1. Banco de dados. 2. Cuckoo Hash. 3. SIMD (arquitetura de computadores). I. Universidade Federal do Paraná. II. Almeida, Eduardo Cunha de. III. Alvez, Marco Antonio Zanata. IV. Título.

CDD: 005.7

Bibliotecária: Vanusa Maciel CRB- 9/1928



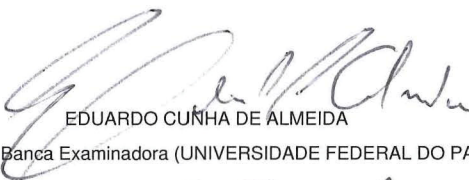
MINISTÉRIO DA EDUCAÇÃO
SETOR DE CIÊNCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA -
40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da dissertação de Mestrado de **FLAVIENE SCHEIDT DE CRISTO** intitulada: **VIVID CUCKOO HASH: FAST CUCKOO TABLE BUILDING IN SIMD**, sob orientação do Prof. Dr. EDUARDO CUNHA DE ALMEIDA, que após terem inquirido a aluna e realizado a avaliação do trabalho, são de parecer pela sua **APROVAÇÃO** no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 09 de Julho de 2019.



EDUARDO CUNHA DE ALMEIDA

Presidente da Banca Examinadora (UNIVERSIDADE FEDERAL DO PARANÁ)



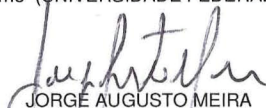
MARCO ANTONIO ZANATA ALVES

Coordenador - Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)



WAGNER MACHADO NUNAN ZOLA

Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)



JORGÉ AUGUSTO MEIRA

Avaliador Externo (UNIVERSITY OF LUXEMBOURG)



"Não, vocês não podem me dizer onde é! Quando sabemos onde estamos, o mundo fica delimitado como um mapa. Quando não sabemos, o mundo parece infinito!"
Cixin Liu, A Floresta Sombria (2008)

AGRADECIMENTOS

Meu principal agradecimento é à minha família, por todo o suporte emocional e financeiro. Meus pais, Elisabeth e Marcos, e minha irmã, Juliane e meu cachorro, Nibble. Sem eles nada disso seria possível.

Aos meus amigos, pela paciência frente às lágrimas e às mudanças de humor. Por cada sorriso e risada que me arrancaram quando os prazos apertavam. Evelini, Karina, Rogério, Nicole, Erik, Stephanie, Juliana e tantos outros que me ajudaram nessa caminhada. Jaque, Nicolly, Tati, Geórgia, Vanessa, Dani e tantas outras mulheres porque juntas nós aprendemos que não há dificuldade que não se vença e que é possível caminhar por esses caminhos sem perder a si mesma.

Aos meus professores e orientadores, por todo o conhecimento adquirido ao longo de todos esses anos. Em especial ao meu orientador Eduardo Cunha de Almeida e a meu coorientador Marco Antonio Zanata Alves, por acreditarem e por todo o suporte oferecido.

RESUMO

Tabelas Hash possuem um lugar de destaque em Bancos de Dados modernos, encontrando aplicações na execução de junções, agrupamentos, indexação, remoção de duplicidades e acelerando consultas pontuais. Essa dissertação tem como foco estudar o efeito do paralelismo em Tabelas Cuckoo. Cuckoo Hashing (Pagh and Rodler (2004)) é uma técnica que lida com colisões garantindo que o dado seja recuperado em, no máximo, dois acessos à memória no pior caso. No entanto, a construção de tabelas Cuckoo com os métodos sequenciais atualmente utilizados é ineficiente ao lidar com o expurgo de chaves que colidem na estrutura de dados. Nós propomos um método vetorizado verticalmente e com técnica de dependência de dados para construir tabelas Cuckoo - ViViD Cuckoo Hash. Nosso método explora paralelismo de dados com instruções SIMD AVX512 e transforma dependências de controle em dependências de dados para reduzir o tempo de resposta médio para o processo de construção em cerca de 90% comparado ao método de construção sequencial.

Palavras-chave: Cuckoo Hash. SIMD. Hash Join.

ABSTRACT

Hash Tables play a lead role in modern databases systems, finding applications in the execution of joins, grouping, indexing, removal of duplicates, and accelerating point queries. In this dissertation, we focus on Cuckoo Hash(Pagh and Rodler (2004)), 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 to treat the eviction of the colliding keys within the data structure. We propose a vertically vectorized data-dependent method to build Cuckoo Tables - ViViD Cuckoo Hash. Our method explores 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.

Keywords: Cuckoo Hash. SIMD. Hash Join.

LISTA DE FIGURAS

2.1	The construction of a hash table. Each key passes through a hash function $f(x)$, the result of the function points to the index where the key must be stored.	15
2.2	How Hash Join works. First, we generate a hash table using the relation R and a hash function $f(x)$. Then we perform lookups on this table for every key of the relation S , if the key is on the hash table, then it will be added to the resultant relation.	17
2.3	Chained Hash mechanism.	18
2.4	Cuckoo Hash mechanism	19
2.5	Runtime in miliseconds of the 5 hash functions for the hash of 150k ordered keys	20
2.6	Colision rates of the 5 hash functions for the hash of 150k ordered keys	20
3.1	Abstraction of the levels of parallelism. The tasks are programs running on the same machine. The threads are parallel if they do not share resources. The same machine can have many cores and each superscalar core has many logical an arithmetic units	22
3.2	Variants of the SIMD architecture. The first variant comprehends the old vectorial supercomputers. The third one, the modern GPUs. The one that interests us is the second, the SIMD ISA extensions. There are three variants of this ISA extensions: MMX, SSE and AVX	23
4.1	Insertion in a Cuckoo Table	25
4.2	Vertical versus horizontal vectorization	27
4.3	Flow chart of the insertion of a key on a Cuckoo Table. First, we check if the key is already present in one of the two tables, if it is not, we check if we reached the threshold. In the affirmative case, the table is rebuilt. We then check for empty spots on the two tables, if none is found, we deallocate a key and restart the process	28
4.4	Load of keys from the relation Orders. News keys are loaded when the bit is set on the mask for that given position	29
4.5	Phases 2 and 3 Hash of the keys to be inserted and gather of the keys already stored on the given positions. All the keys are hashed using the two hash functions. We then gather the keys from both tables and store them in two vectors	29
4.6	Phases 4,5 and 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. Part 4 shows the insertion of keys on the Cuckoo table	30

5.1	Throughput of the build measured in a million keys per second using AVX-2 and AVX-512 with 256 and 512 vectors and the scalar method. The upper plot shows the throughput of the vectorized versions; the bottom plot shows the throughput of the scalar method.	33
5.2	Average clock speed in MHZ for scalar Cuckoo table build, ViViD AVX2, ViViD AVX-512 256-bit vector and ViViD AVX-512 512-bit vector	34
5.3	Power consumed by the package on the build of a hash table using scalar Cuckoo Hash, ViViD Cuckoo Hash with AVX-2 and ViViD Cuckoo Hash with AVX-512 (256-bits and 512-bits versions).	34
5.4	Energy consumed by the package in Joules on the build of a Cuckoo table	35
5.5	Power consumed by the DRAM in Watts on the build of a Cuckoo table	35
5.6	Energy consumed by the DRAM in Joules on the build of a Cuckoo table	36
5.7	Load bandwidth of data from L2 to L1 cache on the build of a Cuckoo Table. . .	36
5.8	Load bandwidth of data from L3 to L2 cache on the build of a Cuckoo Table. . .	37

LISTA DE TABELAS

2.1	Works on improving Cuckoo Hash. We evaluated 4 characteristics, concurrency, the use of SIMD, vertical SIMD vectorization and if the work discusses on the build of the Cuckoo table	16
4.1	Vector variables and masks used to implement the ViViD Cuckoo	28

LISTA DE ACRÔNIMOS

AVX	Advanced Vector Extensions
CPU	Central Processing Unit
DLP	Data-level parallelism
DRAM	Dynamic Random Access Memory
FNV	Fowler–Noll–Vo
GB	Gigabytes
GHz	Gigahertz
GPU	Graphic Processing Unit
I/O	Input/Output
ISA	Instruction Set Architecture
MHz	Megahertz
MIC	Many Integrated Core
MKPS	Million keys per second
OAT	One-at-a-time
OS	Operational System
SIMD	Single Instruction-Multiple Data
TLP	Task-Level Parallelism
TPC-H	Transaction Processing Performance Council - Benchmark H

LISTA DE SÍMBOLOS

\bowtie	join
Π	projection
\cup	union
σ	selection
\wedge	logical and
\times	cartesian product

SUMÁRIO

1	INTRODUCTION	13
2	HASH JOIN LANDSCAPE.	14
2.1	JOINS	14
2.2	LITERATURE REVIEW IN HASHING.	14
2.3	HASH JOINS	16
2.3.1	Chained Hash	17
2.3.2	Cuckoo Hash	17
2.4	HASH FUNCTIONS	18
2.4.1	Murmur	19
2.4.2	Jenkin’s Functions.	20
2.4.3	FNV	20
3	NOTES ABOUT PARALLEL PROCESSING	22
3.1	THE SIMD ARCHITECTURE.	23
3.2	PROGRAMMING WITH ISA EXTENSIONS.	24
4	VIVID CUCKOO HASH	25
4.1	CONCURRENT AND VECTORIZED CUCKOO HASHING	25
4.2	ABOUT BRANCHES AND PREDICTIONS	26
4.3	VIVID CUCKOO.	26
5	EXPERIMENTAL EVALUATION.	32
5.1	METHODOLOGY	32
5.2	RESULTS	33
5.2.1	Throughput	33
5.2.2	Power and Energy	33
5.2.3	Memory	36
5.2.4	Conclusion	37
6	CONCLUSION AND FUTURE WORK.	38
	REFERÊNCIAS	39

1 INTRODUCTION

The use of hash tables in the execution of joins, grouping, indexing, and removal of duplicates is a widespread technique on modern database systems. Take indexing as an example; hash tables are used to point to blocks of secondary storage when data cannot fit into main memory (Garcia-Molina (2008)). In another example, hash tables are used to perform joins in order to avoid dealing with nested loops and sorting, dismissing the need to execute multiple full scans over the same relation. 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 will now take the old one's position. This eviction process repeats until a key finds an empty nest or a given threshold is reached; In the second case, the table must be rebuilt using new hash functions.

All these evictions are costly when we build a Cuckoo table, and even more when we perform a bulk insert of all the keys from a given relation, such as on a Hash Join. We formulated the hypothesis that the use of parallelism could reduce the bottleneck caused by the eviction process. To corroborate this hypothesis, 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 vertically vectorized data-dependent method to build Cuckoo Tables. Experimental evaluation showed that the ViViD Cuckoo Hash technique has ten times higher throughput on average than the scalar method, maintaining the same power consumption profile.

This dissertation aims to contribute to the following:

- a discussion of different methods to enhance the performance of the Cuckoo Join;
- the ViViD Cuckoo Hash, a robust and detailed described method that uses SIMD to make Cuckoo tables build faster;
- a brief discussion about the pros and cons of parallel SIMD instructions (AVX-2 and AVX-512) when building Cuckoo tables regarding throughput, memory, and power;
- an analysis of most suitable hash functions for ViViD Cuckoo;

This document is organized in five chapters; Chapter 2 describes the fundamental concepts on which this work is built upon and discuss some previous works on improving Cuckoo Hash, 3 discuss briefly parallel processing and SIMD. After this, Chapter 4 shows how we answered the research question and describes the implementation of the ViViD Cuckoo Hash. Chapter 5 brings the experimental evaluation. Finally, Chapter 6 concludes this dissertation and suggests future work to be done on the top of what we presented.

¹Using 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 its own

2 HASH JOIN LANDSCAPE

2.1 JOINS

The join operation is presented by the relational algebra theory to combine data from different relations (or tables). Taking into account the formal definition from Silberschartz et al. (2006), a natural join between two relations $R(A_1, A_2, \dots, A_n)$ and $S(B_1, B_2, \dots, B_n)$ 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:

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

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, as shown in the pseudo-code below, being R and S the relations we are joining and T the resultant relations, also called Nested Loop Join.

Nested Loop Join Pseudo-Code

```

1 for r in R:
2     for s in S:
3         if r->key == s->key:
4             T.insert(s)
5     end
6 end

```

This join method is costly regarding time and resources since the number of table scans executed would be the number of keys on the outer relation of the loop. The number of comparisons between keys, in this case, cannot be seen as a concern, just because it 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 dissertation, we focus on Hash Join, that indexes keys using hash functions to build dictionaries, avoiding excessive and table scans.

2.2 LITERATURE REVIEW IN HASHING

Join is one of the most expensive operations in query processing (Silberschartz et al. (2006)), and the use of Hash Tables is a widely spread technique to speed up the execution of joins. Hashing methods are separated into two big groups by 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 Open Addressing, the most interesting when it comes to probing speed and better memory usage. We discuss both methods in-depth on Section 2.3.

Richter et al. (2015) delivers a complete and widespread vision of the main hashing methods used nowadays. 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.

Celis et al. (1985) introduced Robin Hood Hash in 1985 coining the concept of *poor* and *rich* keys. 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 deallocated to give its spot to the poor one. When using open addressing

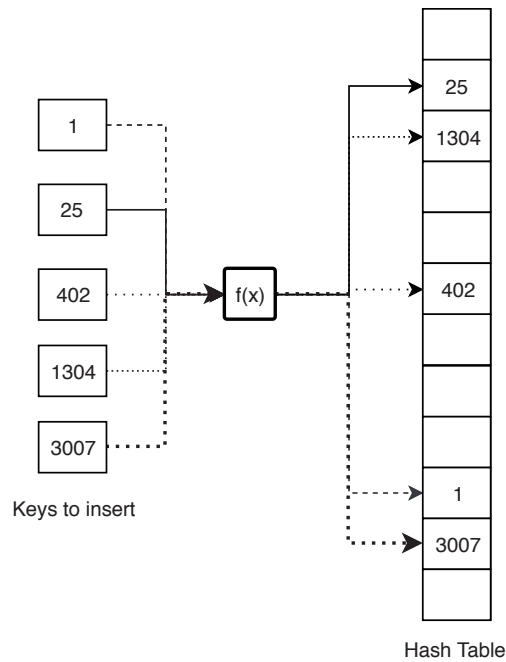


Figura 2.1: The construction of a hash table. Each key passes through a hash function $f(x)$, the result of the function points to the index where the key must be stored.

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, proposed by 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 proposed by 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*, proposed by Zukowski et al. (2006) and Ross (2007).

Although this work focusses on the vectorization of the Cuckoo Hash, some papers on concurrent Cuckoo Hash may be mentioned, since the first attempt in this dissertation to enhance the build performance was to use multiple readers and writers. Li et al. (2014) make an effort to build fast concurrent Cuckoo tables, narrowing the critical section and decreasing interprocessor traffic. Nguyen and Tsigas (2014) suggest an optimistic approach of a lock-free Cuckoo table.

Several papers treat the vectorization of Cuckoo Hashing, such as Zukowski et al. (2006), that presents an excellent view of a semi vertically vectorized probe. Ross (2007) suggest a vertical vectorization for the Cuckoo Hash, but focus on the probing, not on the building of the table. Polychroniou et al. (2015) propose a robust approach, describing the vertical vectorization and the removal of control dependencies both on the build and the probe phase, implemented both in Intel Knights Landing, Haswell and Sandy Bridge microarchitectures. Our work brings improvements to their methods. By maximizing the usage of logical expressions, we take

Title	Authors	Concurrency	SIMD	Vertical SIMD	Build
Cuckoo Hashing	Pagh and Rodler	N	N	N	N
Algorithmic improvements for fast concurrent cuckoo hashing	Li et al	Y	N	N	N
Lock-free cuckoo hashing	Nguyen et al	Y	N	N	N
Architecture-conscious hashing	Zukowski et al	N	Y	SEMI	N
Efficient hash probes on modern processors	Ross et al	N	Y	Y	N
Rethinking simd vectorization for in-memory databases	Polychroniou et al	N	Y	Y	Y

Tabela 2.1: Works on improving Cuckoo Hash. We evaluated 4 characteristics, concurrency, the use of SIMD, vertical SIMD vectorization and if the work discusses on the build of the Cuckoo table

advantage of more recent AVX capabilities - such as inline collision detection and operations with masks - and by applying fast hash functions that guarantee minimum collision rates.

We consolidated the more relevant works discussed above on Table 2.2, evaluating four main characteristics: concurrency, SIMD vectorization, SIMD vertical vectorization and focus on the build phase. The work of Pagh and Rodler (2004) introduced Cuckoo Hash without further discussion about concurrency and parallelism. Li et al. (2014) and Nguyen and Tsigas (2014) discussed concurrency on their works, bringing concepts such as the cuckoo path. Zukowski et al. (2006), Ross (2007) and Polychroniou et al. (2015) are the ones who discussed the SIMD vectorization for Cuckoo Hashing. Zukowski et al. (2006) and Ross (2007) did not focus on the build phase of Cuckoo Join. The work that fulfills the four characteristics is Polychroniou et al. (2015), describing the implementation of Cuckoo Join using SIMD vertically both for the build and the probe phase.

2.3 HASH JOINS

The use of hash tables is a widely spread method to enhance the performance of joins: Figure 2.1 shows the generation of a hash table. It guarantees the execution of only two full scans, one on the first relation to generate the hash table, and one on the second relation, to verify the presence of the key on the table. The same hash function is used on both relations, guaranteeing that two corresponding keys will hash to the same address. Figure 2.2 shows how a Hash Join works. The first step is the build of a hash table based on a relation R . The second is a loop over the relation S that perform lookups on the hash table to verify if the key also belongs to the relation R . If it does, we add this key to the resultant relation.

Hash tables also guarantee that most of the time, only one read will be made to retrieve a value. So, the complexity of the lookup on hash tables is $O(1)$. However, all the data structure must stay in memory to guarantee advantages, because I/O access to disk brings no improvement. Besides the significant amount of time spent to access the disk, it also has the penalty to go down all through the memory hierarchy.

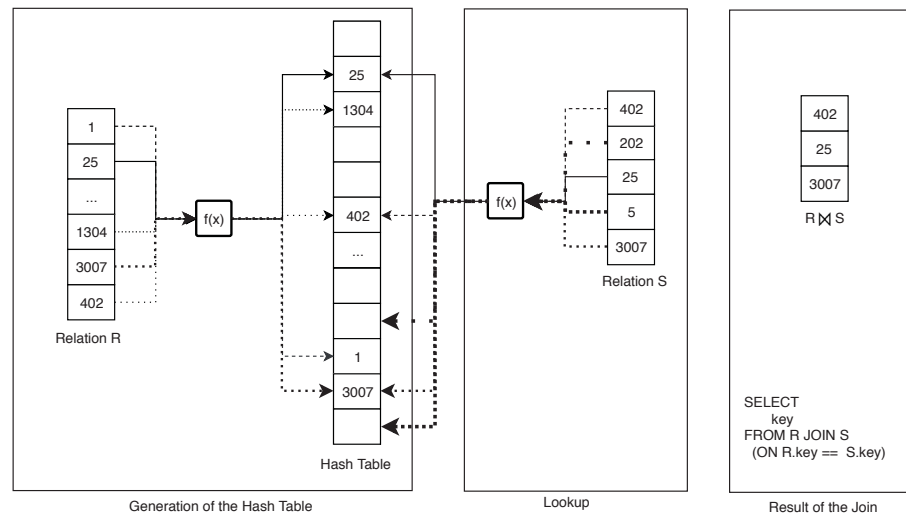


Figura 2.2: How Hash Join works. First, we generate a hash table using the relation R and a hash function $f(x)$. Then we perform lookups on this table for every key of the relation S , if the key is on the hash table, then it will be added to the resultant relation.

The project of a hash table has two significant points: the number of buckets - which must be enough to store all the keys of the table and avoid too many collisions - and the hash function - that has to assure a balanced distribution of keys along the table.

2.3.1 Chained Hash

The primary concern when designing hash tables are the collisions between keys that map to the same address on the table. E.g., being T a hash table, R a relation, k and l two distinct keys of R and h hash functions. If $h(k) = h(l)$, we have a collision between k and l , because both of them maps to the same address on T . There are some methods to avoid and deal with collisions.

To avoid collisions a hash function that ensures perfect load balancing and a table long enough to accommodate all the keys as the ideal scenario. However, since most of the hash tables are not ideal, some techniques are required to deal with collisions.

The most straightforward and perhaps most used schema is Chained Hash. Chained Hash Tables are vectors of pointers to linked lists. Every time a collision occurs, the key gets inserted into the linked list pointed by the index. Figures 2.3(a) and 2.3(b) presents an example of how collisions are treated on Chained Hash - the vector of pointers was suppressed for simplicity, showing, in fact, a vector where each bucket is the head of a linked list, which is also a way to implement Chained Hash.

To perform a lookup on such table, we only need to hash the key to be searched, then use the hashed value as an address to access the bucket that points to the linked list where the key must be. To find the key, we need to go through the linked list. This behavior implies that if the table is unbalanced, some linked lists will be too long and the lookup complexity will no longer be constant. In the worst case - being n the number of keys - all the keys map to the same bucket, and the complexity will be $O(\lg n)$ or n - if the key we seek maps to the last position of the list or is not in the table.

2.3.2 Cuckoo Hash

Another way to deal with collisions is to use open addressing. In general terms, when a collision occurs, the key must either find the nearest empty location or be hashed again with

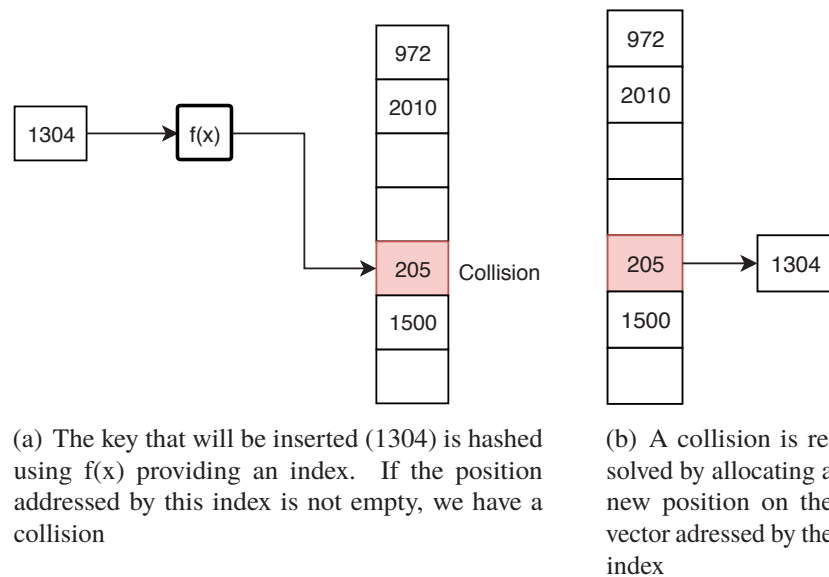


Figura 2.3: Chained Hash mechanism

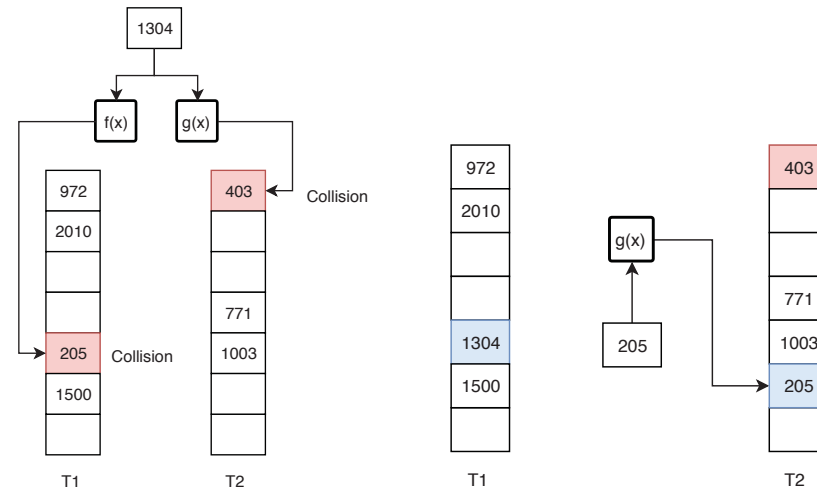
another function. It means that no key has a unique associated address on the table. There are a variety of open addressing methods, such as Linear Probing, Robin Hood and Cuckoo Hashing (Pagh and Rodler (2004)). In this dissertation, we will only discuss Cuckoo Hashing. The Cuckoo Hash algorithm involves two hash functions, and very often two tables, but it can also have only one table with two hash functions.

Since we have two hash functions, we assigned each one to a table. A key can be on any of them, but never on both at the same time. In every table, the key will occupy a different position - here called bucket -, since we use different functions to address each of them. When there is no place for a key on none of the hash tables, as shown in Figure 2.4(a), we choose one of the buckets and the key currently stored is removed and replaced by the new key. The old key is not deleted, only rehashed with the function assigned to the other table and try to insert, Figure 2.4(b) illustrates this process. If a collision occurs again, the process repeats within a limit; That is, we must set how many times the reallocation of keys will happen before the reconstruction of the entire table. In the rebuilding, we use two new functions - distinct from the previous - or new seeds for the same functions.

Cuckoo Hash maximizes the occupation of the table, while Chained Hash cannot assure load balancing, leaving most of its buckets empty and some with long linked lists. It means that on average, Cuckoo tables occupy less space in memory than chained tables. Although, the insertion could cost more than on the Chained method - especially when the table needs to be rebuilt -, the lookup is faster in average, it takes at most two access because the key can only be in one of two places.

2.4 HASH FUNCTIONS

Key collisions are a decisive matter on hash tables; we can either deal with them or try to avoid the said collisions by choosing the right hash function for the domain. The choice of a hash function must consider primarily two factors: the distribution of the outputs, and the speed. The first factor tells us about how the keys will be distributed on the table. If the distribution shows concentrated areas, then we have a clusterization of keys, and so the keys are more likely to collide. Clusterization gives us a better locality for keys, but if it is not essential for the problem -



(a) The key that will be inserted (1304) passes through the two hash functions providing two indexes, one for each table. If the two positions pointed by the indexes are not empty, we have a collision. If one of the positions is empty, we insert the key in this position

(b) When a collision occurs, one of the keys already stored must be removed to make a place for the new one, we choose the key 205 from T1 randomly. The old key is then rehashed with the function that addresses the other table, in this case, $g(x)$, which addresses the T2

Figura 2.4: Cuckoo Hash mechanism

although could be a good strategy for aggregations -, so we look for hash functions that do not clusterize its outputs.

Non-cryptographic hash functions are usually constructed following the Merkle-Dangard scheme, which divides the input into pieces and processes one by one using a *mixing function*, combining each piece with its internal state. Hash functions built with this mechanism usually have four fundamental blocks: Initialization, Mixing, Finalization, and Compression. Initialization and Finalization are optional, and Compression is the phase where the output is translated from a $[2, 2^n)$ interval to a $[0, M)$ interval, being n the output size in bits and M the output interval wanted (Estébanez et al. (2014)) - in our case, the number of buckets of the hash table.

In this dissertation, we compare the functions Murmur2, Murmur3, Lookup3, One-at-a-time, and FNV1a, mostly because each one of them showed decent speed while maintaining a good dispersion when compared with other functions on previous works - such as Estébanez et al. (2014) - and were straightforward to implement. Figure 2.5 shows the time to hash 150k ordered keys in milliseconds, and Figure 2.6 presents the collision rates for each function on an Intel® Core™ i7-6500U CPU @ 2.50GHz with 8GB of RAM, each function implemented scalarly.

2.4.1 Murmur

Murmur2 (Appleby (2016)) was released in 2008 as a more robust and fast version of Austin Appleby's original MurmurHash. It was reported to have a flaw in the mix that does not affect most applications. Its mixing function uses three multiplications, three *or* operations, and one *shift* operation. The Apache Hadoop data processing system, Maatkit and libmemcached - among others - are reported to use Murmur2 (Estébanez et al. (2014)). Algorithm 1 presents the pseudo-code of the Murmur3 Hash.

Murmur3 succeeded Murmur2 and was released in three variants. The first one is a 32-bit implementation that targets low latency. The second and third focuses on generating

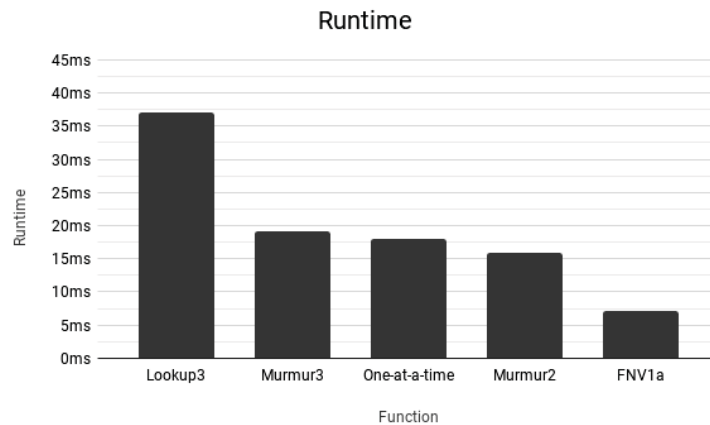


Figura 2.5: Runtime in miliseconds of the 5 hash functions for the hash of 150k ordered keys

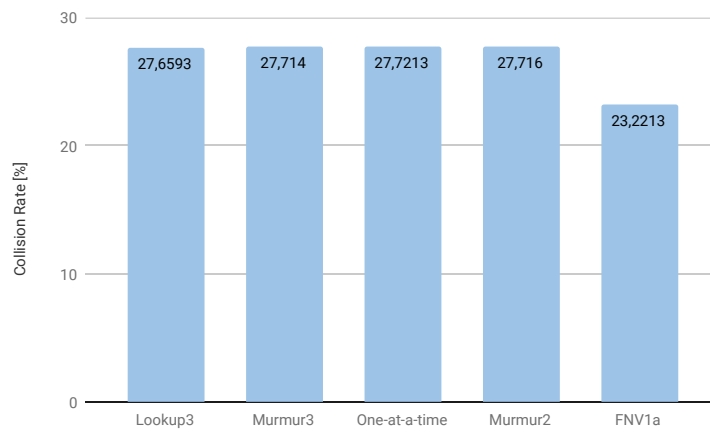


Figura 2.6: Collision rates of the 5 hash functions for the hash of 150k ordered keys

unique identifiers for large blocks of data, the two of them are 128-bit implementations and differs on the platform - x64 and x86 - they are optimized to. We used the 128-bit x64 version. The systems Apache Cassandra (Apache (2018)), RaptorDB (Gholam (2012)) and Elasticsearch (elastic (2018)) among other OpenSource projects uses Murmur3. The 128-bit x64 version of Murmur3 was slower than the Murmur2 on the Skylake processor.

2.4.2 Jenkin's Functions

OAT and lookup3 are both functions designed by Bob Jenkins. The first one is a simple function similar to rotating hash, except it mixes its internal state, taking $9n + 9$ instructions to produce a 4-byte result (Jenkins (2011)). Lookup3 is the default hash function used on Postgres to perform hash joins (PostgreSQL (2018)). It is reported by Jenkins to do a more robust mix than OAT, but our tests showed that both have similar collision rates. OAT also showed to be faster than Lookup3.

2.4.3 FNV

The last function is FNV1a, created by Glenn Fowler, Landon Curt Noll, and Kiem-Phong Vo and presented on Algorithm 2. FNV1a mixes each byte of the key using a *xor* operation

Algorithm 1 Murmur3 32-bits Generic Algorithm

```

1: function MURMUR3(key, len, seed)
2:   hash ← seed
3:
4:   for all fourBytes ∈ key do
5:     k ← fourBytes                                     ▶ Mixing
6:     k ← k * 0xcc9e2d51
7:     k ← rotateLeft(k, 15)
8:     k ← k * 0x1b873593
9:
10:    hash ← hash xor k
11:    hash ← rotateLeft(hash, 15)
12:    hash ← hash * 5 + 0xe6546b64
13:  end for
14:
15:  hash ← hash xor (hash >> 16)                       ▶ Finalizing
16:  hash ← hash * 0x85ebca6b
17:  hash ← hash xor (hash >> 13)
18:  hash ← hash * 0xc2b2ae35
19:  hash ← hash xor (hash >> 16)
20:
21:  return hash
22: end function

```

and one multiplication. A prime number and a seed must be chosen for its mixing state; both are dependent on the hash size, here called n . The prime is calculated using the formula $2^{n-8} + 2^8$, and the ideal seeds are fixed integers defined by the authors (Noll (2017)).

Algorithm 2 FNV1a 32-bits Generic Algorithm

```

1: function FNV1A(key, prime, seed)
2:   hash ← seed
3:
4:   for all byte ∈ key do
5:     hash ← hash xor byte
6:     hash ← hash * prime
7:   end for
8:
9:   return hash
10: end function

```

Jenkins (2011) reports FNV1a to be faster than Lookup3 on Intel platforms, and our tests corroborate this, as seen on Figure 2.5. FNV1a also had the lowest collision rate among the five functions. For those two reasons, we choose this function to index the first table of the ViViD Cuckoo, while Murmur3 was chosen for the second table for its robustness - i.e., had the flaws presented by Murmur2 corrected-, even not being faster or having lower collision rates.

3 NOTES ABOUT PARALLEL PROCESSING

Single-Instruction Multiple-Data (SIMD) is a concept that has been around for a while (Hennessy and Patterson (2012)). It brought enhancements to resolve a broad set of problems that are highly parallelizable on what concerns the data. However, the concept of parallelism goes beyond the *data-level parallelism*, finding applications on most of computer science problems.

Parallelism works at multiple levels in two primary classes: *Data-level Parallelism* (DLP) and *Task-Level Parallelism* (TLP) (Hennessy and Patterson (2012)). Figure 3.1 shows an abstraction of those levels. On the top, we have tasks and threads; each task and thread is either specified by the programmer or by the OS; these tasks can be seen as programs running on a machine. It is possible to run multiple tasks for multiple users on modern machines, and each of those tasks operates by its own; however since they compete for system resources, they are more likely to be concurrent than parallel processes. We call this kind of TLP MIMD (*Multiple-Instruction Multiple-Data*) since we have multiple independent tasks operating over distinct data sets. This kind of parallelism is exploited on *clusters* and *warehouse-scale computers*, that deal with independent requests in parallel.

Threads operate inside the tasks, and they often need to communicate among each other and synchronize their executions. Again, they are more likely to be concurrent than parallel processes, but they are also considered MIMD processes.

The second layer of Figure 3.1 represents that parallelism of the *cores* inside the processor. Typically, each core can handle one thread, but technologies as hyperthreading allow the use of virtual cores. We will abstract the scheduling of processes and consider that each core executes a thread without any interruption. When there is no communication or synchronization between the cores, we can also say that we have a MIMD processing (Flynn (1972)).

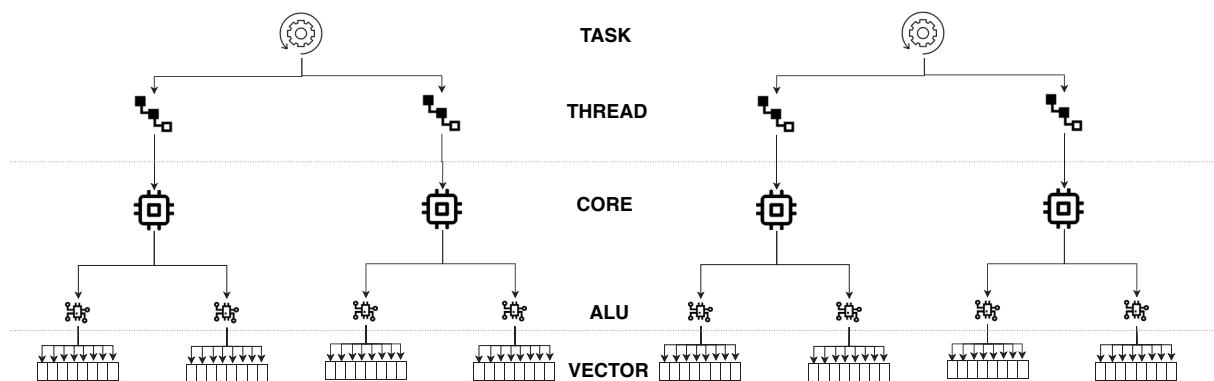


Figura 3.1: Abstraction of the levels of parallelism. The tasks are programs running on the same machine. The threads are parallel if they do not share resources. The same machine can have many cores and each superscalar core has many logical an arithmetic units

The next level starts to make it more interesting; It tells us about parallelism *inside* the cores. Modern processors are superscalar, which means that they can execute more than one instruction per clock by having multiple execution units, providing *instruction-level parallelism*. Some of those units can also be used to process large sets of data simultaneously, using the same stream of instructions, providing SIMD capabilities.

3.1 THE SIMD ARCHITECTURE

According to Hennessy and Patterson (2012), there are three primary variants of the SIMD architecture, as shown in Figure 3.2; The first one comprehends the old vectorial supercomputers, such as the Cray-1 (Russell (1978)), launched in 1978. Vector Architectures grab data dispersed on memory and place them into large, sequential register files and apply instructions over those files to later scatter them across the memory. In other words, they take sparse matrices, turn them into dense vectors, operate over those vectors in a pipeline, and store the data back on sparse matrices. Although this kind of processing seems to be highly efficient, it is only suitable for a limited amount of problems: the ones that have significant DLP. Vector architectures were also considered too expensive until recently Hennessy and Patterson (2012), since it requires larger memory bandwidth and additional transistor.

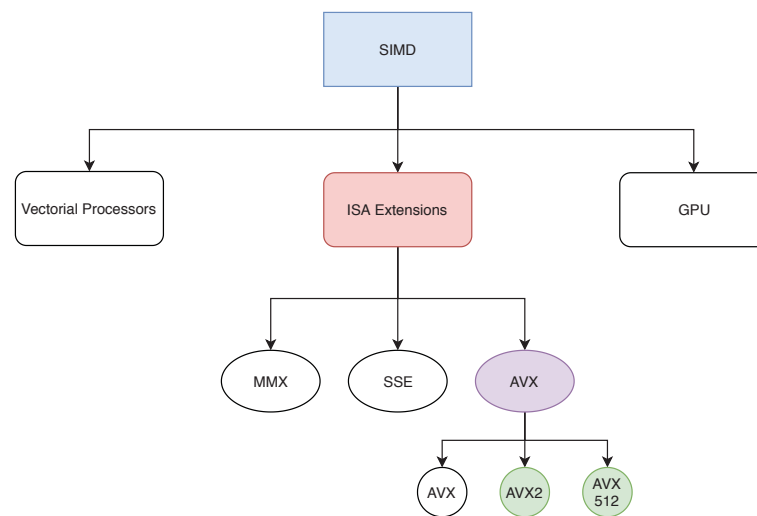


Figura 3.2: Variants of the SIMD architecture. The first variant comprehends the old vectorial supercomputers. The third one, the modern GPUs. The one that interests us is the second, the SIMD ISA extensions. There are three variants of this ISA extensions: MMX, SSE and AVX

The second variant stated by Hennessy and Patterson (2012) is the one that interests us for this dissertation: multimedia instruction set extensions. As the name says, this variant is an extension for the x86 original ISA to support SIMD for many sorts of problems. The third variant is the SIMD capabilities provided by nowadays GPUs, but that is not the focus of this dissertation.

Three main ISA extensions implement SIMD capabilities into modern processors. The first one is the MMX, released by Intel in 1997 (Peleg et al. (1997)), including eight sets of 64-bit registers packed over the existent 80-bit registers already present on the floating-point unit of the Pentium processors. It also included 47 new instructions to process integers of 8, 16 and 32 bits. There were no floating-point instructions since the MMX targeted multimedia applications, such as images and sound processing.

The next generation of SIMD extensions released by Intel was the *Streaming SIMD Extensions* - SSE (Diefendorff (1999)). Intel included eight new 128-bit registers, as well as 47 new instructions. However, the most significant improvement over the MMX technology was the capability of performing single-precision floating-point arithmetic over those extended registers. As MMX reused registers from the floating-point unit, the processors were unable to perform SIMD instructions and floating-point operations at the same time. SSE brought a solution to this problem by separating the SIMD registers from the floating-point unit. Several versions of SSE were released until the next generation of SIMD extensions.

In 2011 Intel released the new Sandy Bridge processors with a new ISA extension to support SIMD operations, the *Advanced Vector Extensions* (AVX). Wider registers of 256-bit were added to the existing set from SSE extensions, now supporting double-precision floating-point operations. A new fused multiply-add operation was also announced as one of the primary advances over MMX and SSE technologies. Requirements for memory alignment on some instructions were relaxed, allowing unaligned memory access (Lomont (2011)).

Within the introduction of the new Haswell microarchitecture in 2013, Intel released the new AVX2 ISA extension. The primary enhancement over the previous generation of AVX was the capability of performing integer arithmetic using 256-bit registers, while in the Sandy Bridge AVX extension, those registers were reserved only for floating-point operations. Bit manipulation, any-to-any permutes, vector-vector shift and gather operations were also included on the AVX2 set (Buxton (2011)). Those operations are useful for hashing, particularly the gather operations, that allows the load of data scattered across the memory.

In 2017 the Skylake family was released bringing a new generation of AVX, the AVX-512. This new technology included some 512-bit registers, as well as bitmask registers of 8, 16, 32 and 64 bits (Reinders (2017)). A whole new set of instructions targeting direct operations with masks was included. On the previous version of AVX, vectors of 128 and 256 bits are used to emulate bitmasks, using the most significant bit of each 32-bit integer of the vector as the bitmask. Another advancement is the addition of scatters operations, that benefits that the hash tables constructions by adding the capability of storing disperse data across the memory.

3.2 PROGRAMMING WITH ISA EXTENSIONS

There are two ways of programming using ISA SIMD extensions. The first one is implicit; the compiler deals with the transformation of parallelizable loops into SIMD instructions. The programmer arranges loops on a way that the compiler can be able to operate over vectors. The other way is programming using extensions to call the instructions, using vector variables explicitly. We can do this by using either assembly calls within the code or using libraries, such as Intel Intrinsics. Usually, those libraries implement data types compatible with the registers sizes provided by the extensions and most of the functions provided maps directly to one instruction of the ISA extensions. However, there are some exceptions and some functions that are used only at compiling time, such as casting operations¹.

Programming explicitly for SIMD ISA extensions forces the programmer to think sequentially on the pipeline of operations done over the vectors of data. Since all the data must pass over a pipeline of instructions sequentially, no control structures can be used to operate on a single part of the set separately. This scenario forces the programmer to use logical and arithmetic instructions, as well as operations that use masks, to control the flow of the program. The use of control structures over subsets of data is possible, however harmful for the general performance and makes the code less readable and direct.

In this dissertation, we choose to use Intel Intrinsics library to program using AVX-2 and AVX-512 explicitly. Some MMX and SSE were also used since there is retrocompatibility and AVX-512 does not handle well any to any permutations. The ViViD Cuckoo Hash method was design based on sequential circuits, keeping the data-dependency and avoiding harmful control-dependencies.

¹Intel provides a full guide for Intrinsics at <https://software.intel.com/sites/landingpage/IntrinsicsGuide>

4 VIVID CUCKOO HASH

As discussed in the previous section, extensive work has been done on making the probe of Cuckoo tables faster using either bucketized tables, parallel algorithms, and SIMD. Our scope is not the same as those previous works, where the hash table suffers insertions and lookups at the same time. When we analyze read-intensive databases running analytic workloads, it is hard to optimize everything to a single operator, e.g., we cannot store a database relation as a Cuckoo table thinking about optimizing joins when this structure is not the ideal one for indexes and select scans. Keeping in mind that a join, for example, has several flavors for a myriad of situations, storing structures for each operator is not the best approach either, leading to consistency issues and unnecessary duplication of the data. In practice, we must build a Cuckoo table every time we perform a join.

4.1 CONCURRENT AND VECTORIZED CUCKOO HASHING

Our 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, as made by Li et al. (2014).

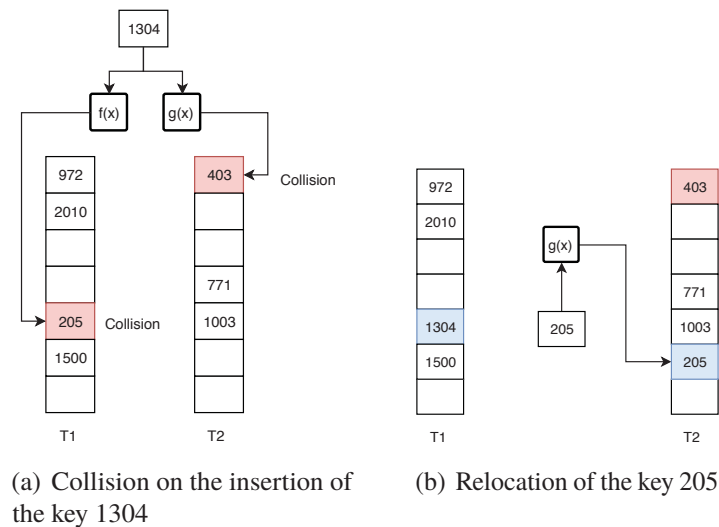


Figure 4.1: Insertion in a Cuckoo Table

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. *Cuckoo path* is the sequence of evicted keys (Nguyen and Tsigas (2014)). Every inserted key has his *cuckoo path*. Figure 4.1 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 will collide leading to inconsistencies on the final table.

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

insertion 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. It is possible to horizontally vectorize the build by applying multiple operations at the same time on one key, as shown in Figure 4.2(b). This method is not particularly suitable for the build of Cuckoo tables because some operations are inherently dependent on each other. 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. An example of a vertically vectorized insertion is presented on Figure 4.2(b), showing four keys passing through the same phases at the same time.

4.2 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. Figure 4.3 describes how this loop and branches work without further optimization and probing the two tables sequentially.

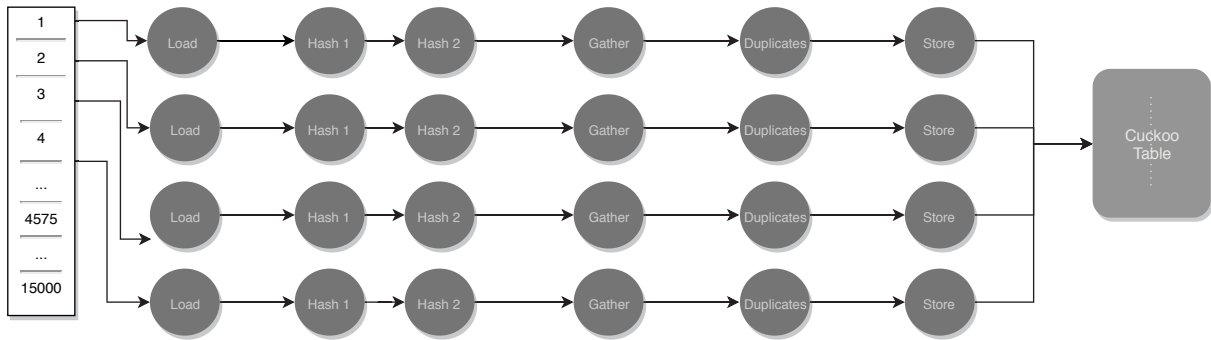
The execution flow of the five conditional statements displayed is unpredictable. By definition, each key has the same probability of hashing to any bucket independently of other keys (Cormen et al. (2009)). It causes overhead since, for each wrong branch taken preemptively, the processor has to undo all the instructions. Polychroniou et al. (2015) and Ross (2007) already discussed this issue by transforming the *control dependency* into *data dependency* (i.e., transforming the control flow structures into logical operations).

4.3 VIVID CUCKOO

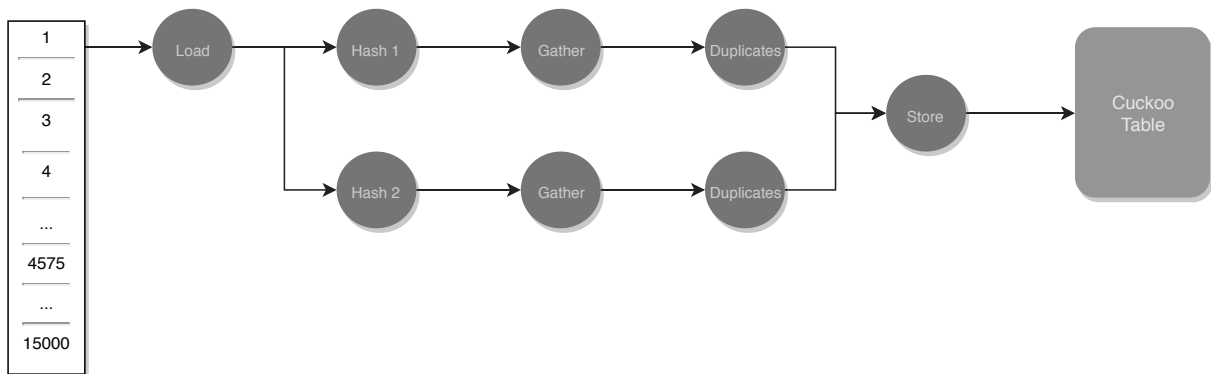
We created the *Vertically Vectorized Data Dependent Cuckoo Hash* (ViViD Cuckoo) by transforming most of the *control dependencies* into *data dependencies* and using SIMD registers to vectorize the build and the probe vertically. With ViViD Cuckoo we could address the two main issues that caused the 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 and to take advantage of logical operations and the dispersion provided by the hash function.

The ViViD Cuckoo Hash comes in two flavors, according to the AVX extension supported by the target microarchitecture. The first flavor is a more general implementation of the method, using AVX-2 extensions. Intel introduced AVX-2 in 2013 in the Haswell family of processors extending the AVX architecture¹ to support 256-bit vectors operations and adding 30 new instructions to the set. The essential operation brought by AVX-2 is the *gather*, which allows loading keys from non-contiguous memory addresses (Hennessy and Patterson (2012)). Unfortunately, *scatters* were not introduced into AVX extensions until AVX512F, which means

¹present on Intel processors since 2008, presented with the Sandy Bridge family



(a) In the vertical vectorization, multiple keys are inserted at the same time. This example shows four keys being inserted; all of them goes through the various phases of the hashing at the same time. We call this kind of vectorization vertical in counterpoint to the horizontal variation



(b) In the horizontal vectorization, one key goes through different phases at the same time. Usually, a table holds several buckets at the same position, and all the buckets are searched at the same time. In this image we show a key that is hashed simultaneously by two distinct hash functions that address to positions on different tables at the same time

Figura 4.2: Vertical versus horizontal vectorization

that AVX-2 does not allow indexed stores. Our solution is to store 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, but not enough to inflict penalties to the throughput.

The other flavor is implemented using AVX-512 extensions, that allow operations with 512-bit-vectors and also brings *scatter* operations, prefetching and opmasks. Intel MIC based coprocessors, such as Xeon Phi, have been already working with 512-bit vectors for a while, but they were not introduced on AVX extensions until the launching of the Skylake-X family. AVX-512 extends and refines the 512-bit operations present on earlier Xeon Phi's architectures, but are not binary source compatible (Reinders (2017)). Polychroniou et al. (2015) implemented vectorized Cuckoo Hashing on Xeon Phi x-100 using 512-bit-vectors and operations, but unfortunately, we could not compare the performance of their implementation using our metrics because of the reasons stated above.

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-2 has operations involving masks,

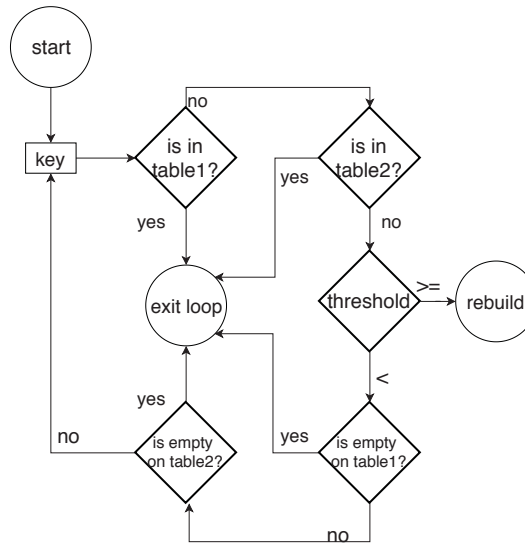


Figura 4.3: Flow chart of the insertion of a key on a Cuckoo Table. First, we check if the key is already present in one of the two tables, if it is not, we check if we reached the threshold. In the affirmative case, the table is rebuilt. We then check for empty spots on the two tables, if none is found, we deallocate a key and restart the process

but there are no special registers, so the masks are held on standard vector registers. We emulated the operation of masks instructions with logical operations.

The AVX-2 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 262k positions, each position holding a single key. It is possible to access the table using both scalar and vectorized methods.

Figures 4.4, 4.5 and 4.6 present three of the eight phases of the ViViD Cuckoo Hash and Table 4.1. Each SIMD variable is a boolean mask or holds integer values that represent keys, hashed keys, and counters. Table 4.1 shows the vector variables and masks used, their types and roles.

Variable Name	Type	Function
loadMask	Mask	Indicates the positions where new keys will be loaded
keysVector	Vector	Vector holding the keys to be inserted
hashedVector	Mask	Hashed values
valuesVector	Vector	Values retrieved from the Cuckoo table
table1Mask	Mask	Indicates which keys must be inserted on table 1
remotionMask	Mask	Indicates keys not to be inserted
storeMask	Mask	Indicates keys that will be stored on the Cuckoo table

Tabela 4.1: Vector variables and masks used to implement the ViViD Cuckoo

We split the build process into 8 phases to better understanding 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.

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

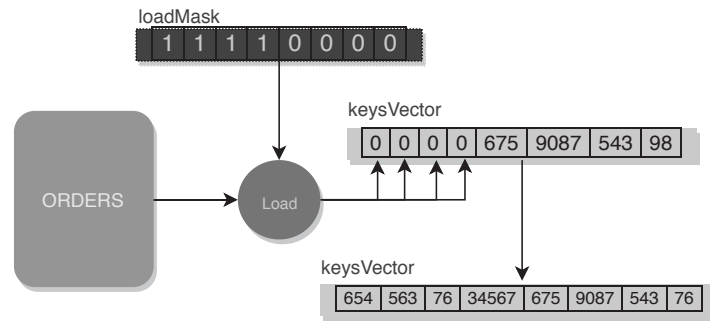


Figura 4.4: Load of keys from the relation Orders. News keys are loaded when the bit is set on the mask for that given position

to be loaded. Old keys occupying the positions indicated by the loadMask are not set, and the new keys are set. Figure 4.4 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 the 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 both loaded on this iteration respectively on the fourth, sixth and fifth positions of *keysVector*.

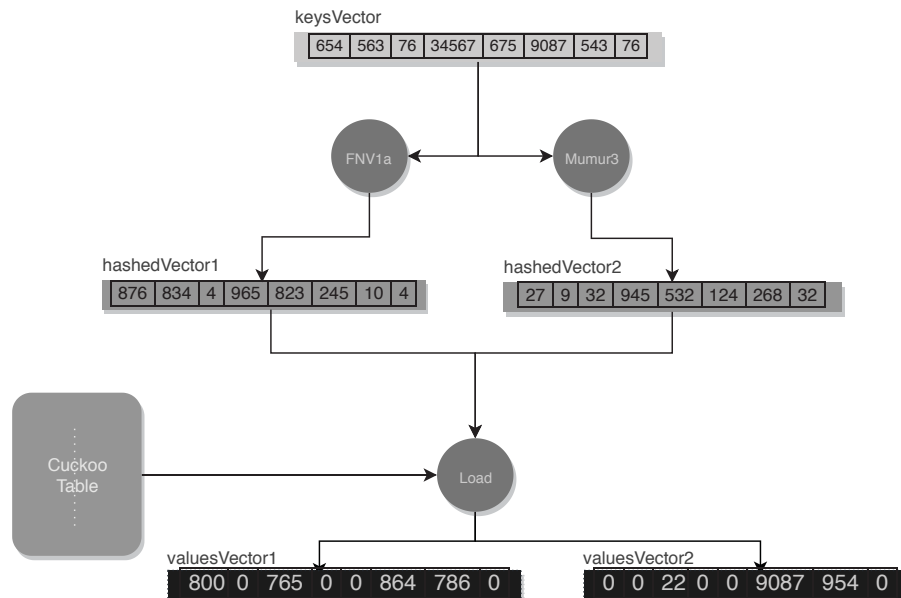


Figura 4.5: Phases 2 and 3 Hash of the keys to be inserted and gather of the keys already stored on the given positions. All the keys are hashed using the two hash functions. We then gather the keys from both tables and store them in two vectors

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. As Murmur2 was reported to have a flaw by its author (Appleby (2016)), we used the newest release, Murmur3. These two hash functions are also the most straightforward to vectorize and are fast to compute, as discussed on Chapter 2.

Figure 4.5 presents the hashing and the gathering (**Phase 3**) of the keys. 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.

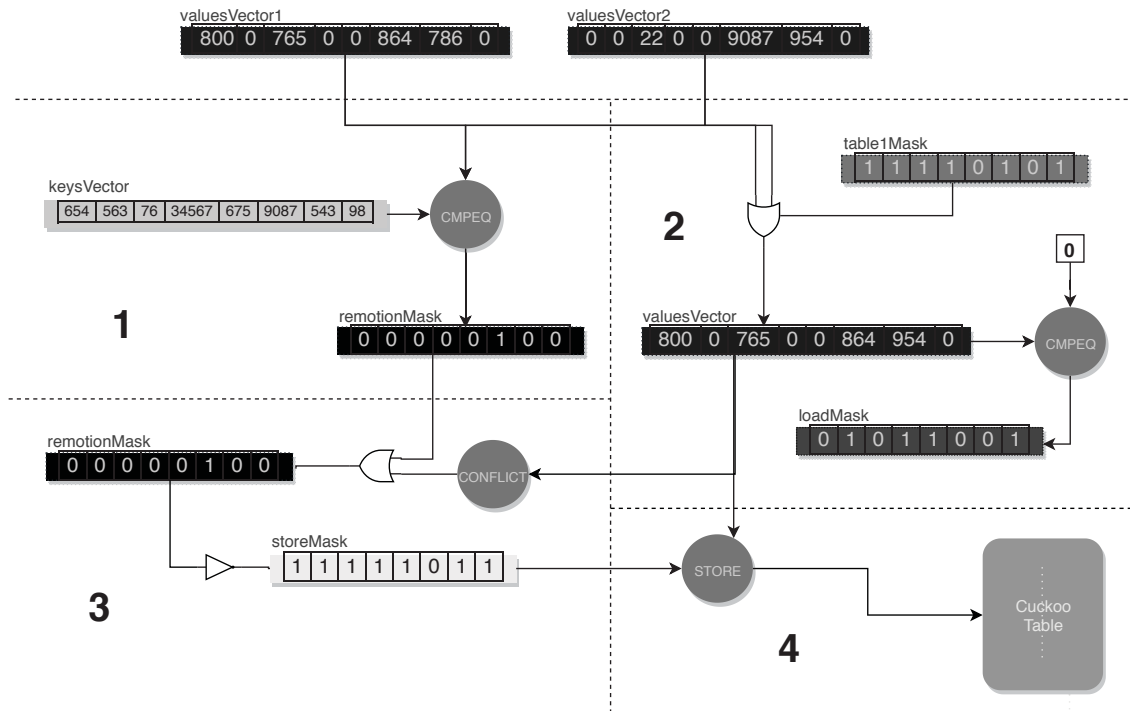


Figure 4.6: Phases 4,5 and 7. Duplicated values, conflicts and successful insertions detection and actual store of keys. Part 1 compares the keys gathered from the table with the keys we are trying to insert to detect duplicated keys. Part 2 selects the keys according to the table they must be stored and compares to zero, to find insertions that do not need reallocations. Part 4 shows the insertion of keys on the Cuckoo table

Phase 4 involves the detection of duplicated keys, conflicts, and successful insertions. Part 1 of Figure 4.6 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 4.6 on the example of the key 9087.

Part 2 of Figure 4.6 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 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-2 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 4.6 shows this rather trivial operation.

Phase 8 permutes 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.

5 EXPERIMENTAL EVALUATION

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 consumption and memory behavior. All the experiments were compiled and executed using Linux Ubuntu 18.04 LTS Bionic Beaver on an Intel® Xeon® Silver 4114 at 2.20GHz based on Skylake-X architecture. All the code¹ 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, such as Ross (2007) and Zukowski et al. (2006), have already studied the probe phase.

5.1 METHODOLOGY

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 optimizer². 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.

Query used on the experiments

```

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

```

We produced nine versions of *Orders* based on the selectivity of the joins. We did that by duplicating the keys since the attribute *CUSTKEY* is a foreign key on the table *Orders* and a primary key on the table *Customer*. The *Customer* table has 150k keys and all the *Orders* tables have 1.5M keys each. We emulated the engine of a Vectorwise database (Boncz et al. (2005)) in the query process.

The metrics analyzed are throughput, power and energy consumption, and cache bandwidth. 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), causing almost no overhead on the executions. 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 problems caused by interprocessor traffic and cache coherence.

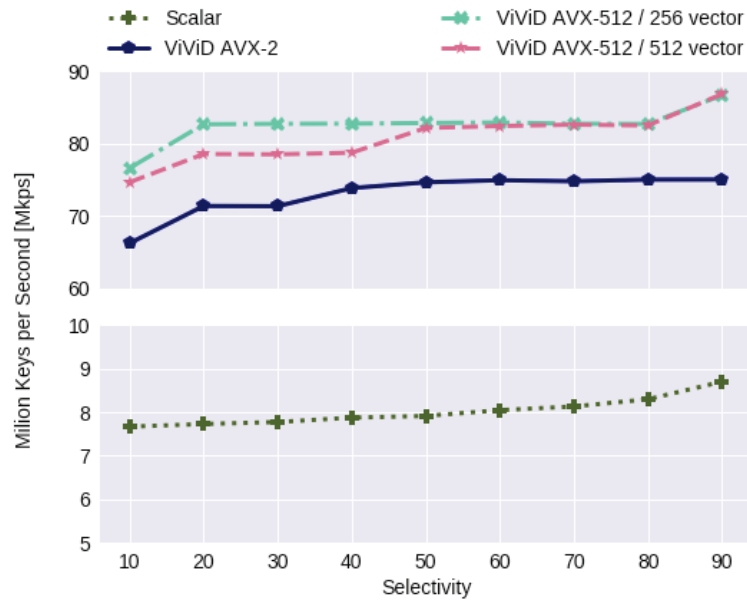


Figura 5.1: Throughput of the build measured in a million keys per second using AVX-2 and AVX-512 with 256 and 512 vectors and the scalar method. The upper plot shows the throughput of the vectorized versions; the bottom plot shows the throughput of the scalar method.

5.2 RESULTS

5.2.1 Throughput

Figure 5.1 shows the throughput in million keys per second when building a Cuckoo table using the scalar Cuckoo hashing and ViViD Cuckoo implemented with AVX2, AVX512 with 512-bit vector and 256-bit vector varying the selectivity of the join operation. Important to note that the number of duplicated keys contract according to the growth of the selectivity - this way, we maintain the same size of the relations for each selectivity. The ViViD Cuckoo in any version improved the throughput in almost ten times on average. The AVX2 version had the poorest throughput, but still shows an average improvement of 8 times over the scalar version. For selectivities bellow 50%, the 256-bit vector version of the AVX-512 implementation presented the best performance but showed the same throughput as the 512-bit vector version when the selectivity is equal or greater than 50%.

This improvement occurs because we now store 8 or 16 keys each time. Each of these keys does not depends on its neighbors *path* to be inserted. There is just one step to verify collisions within the same iteration. Without the control dependencies, we could also improve the throughput, because it does not have the overhead caused by the wrong predictions made by the branch predictor.

5.2.2 Power and Energy

Figure 5.2 shows the clock speed in MHZ. The scalar version presented the highest clock, while the three ViViD implementations presented the lowest clocks. Generally, processors decrease the clock speed to save energy. According to Intel® (2019), Skylake processors decrease

¹Available at <https://github.com/FlavScheidt/MHaJoL>

²PostgreSQL v.11.2 Optimizer: <https://www.postgresql.org/docs/current/planner-optimizer.html>

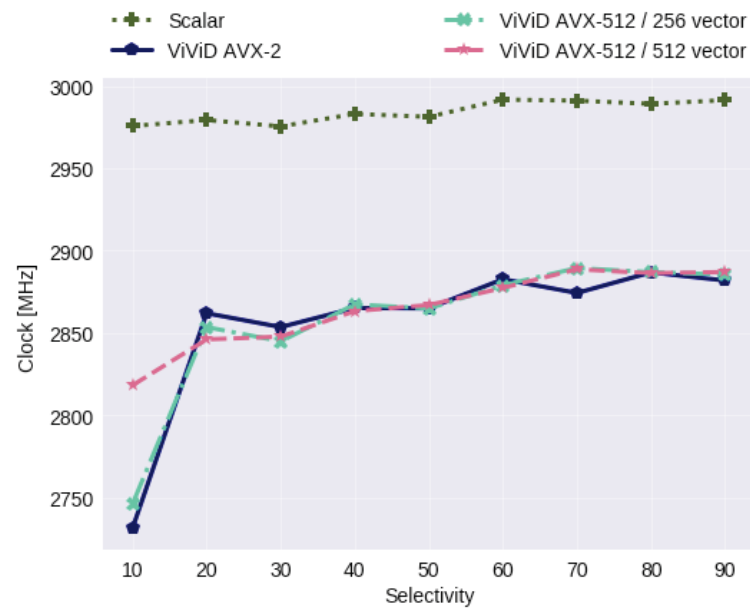


Figura 5.2: Average clock speed in MHz for scalar Cuckoo table build, ViViD AVX2, ViViD AVX-512 256-bit vector and ViViD AVX-512 512-bit vector

the clock speed to medium or low when executing AVX-512 instructions. Figure 5.3 presents the power consumed by the package (socket), corroborating that AVX-2 and AVX-512 versions have the same power consumption profile than the scalar version. Although, it is important to note that the ViViD implementations consume the same amount of power as the scalar version, but taking less time. Figure 5.4 shows the absolute amount of energy in joules consumed by each method. The three ViViD Cuckoo Hash flavors consumed ten times less energy than the scalar version, which is expected since they are ten times faster on average compared to the scalar method.

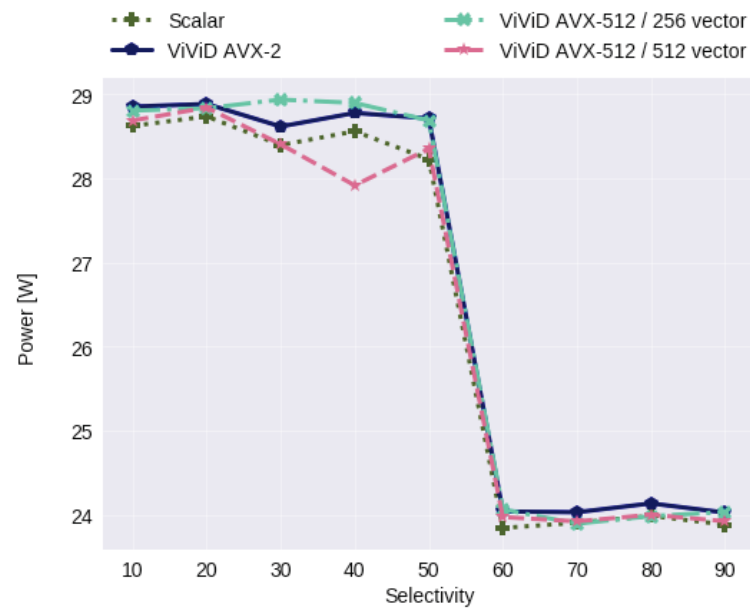


Figura 5.3: Power consumed by the package on the build of a hash table using scalar Cuckoo Hash, ViViD Cuckoo Hash with AVX-2 and ViViD Cuckoo Hash with AVX-512 (256-bits and 512-bits versions)

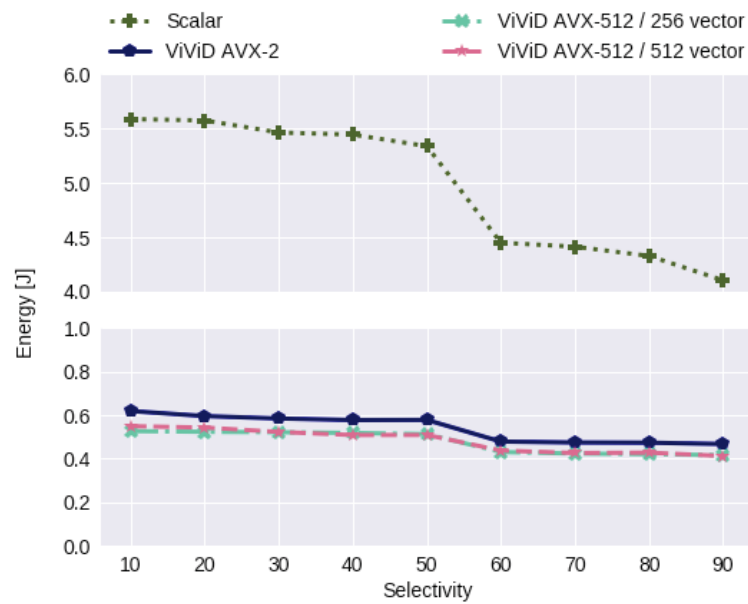


Figura 5.4: Energy consumed by the package in Joules on the build of a Cuckoo table

According to Cebrián et al. (2014), 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 each cycle is higher than on scalar executions. This pressure will cause an increase on power consumption for the memory system, as can be seen in Figure 5.5, that shows the scalar implementation with the lowest consumption and the ViViD with the higher consumption. Although, the absolute amount of energy spent by the three flavors of ViViD Cuckoo is ten times lower than the scalar version, as shown in Figure 5.6.

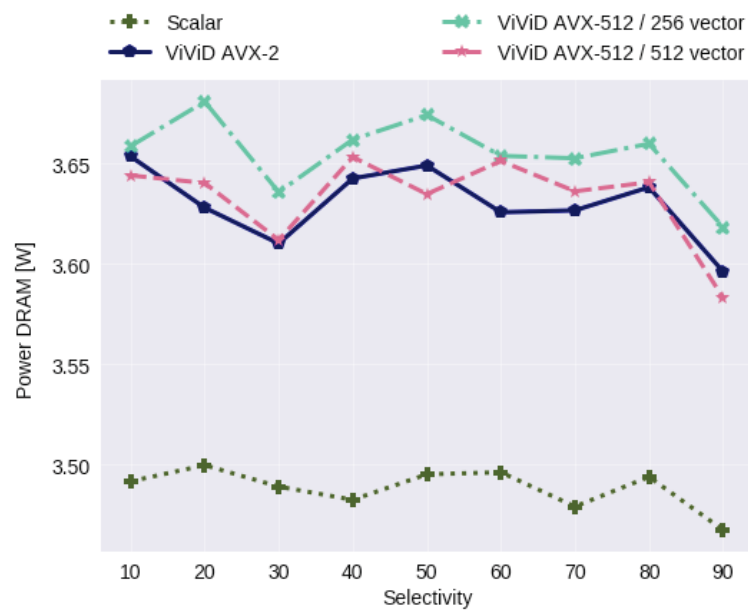


Figura 5.5: Power consumed by the DRAM in Watts on the build of a Cuckoo table

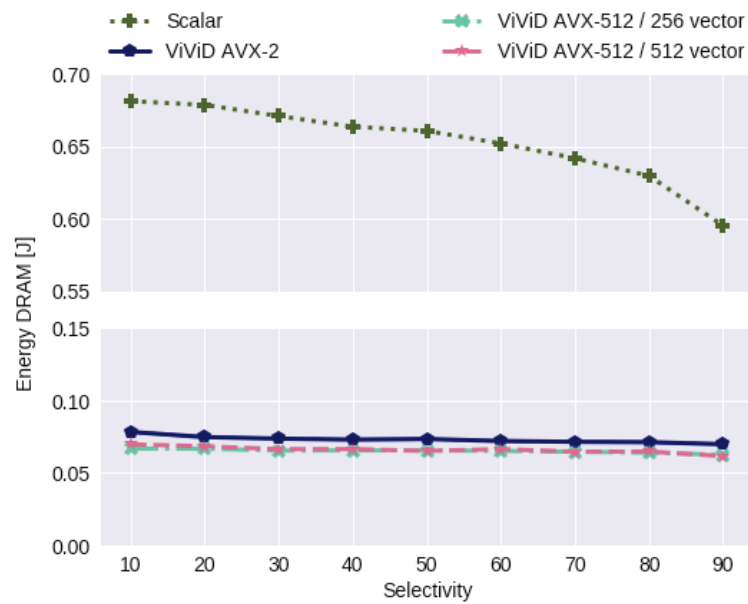


Figure 5.6: Energy consumed by the DRAM in Joules on the build of a Cuckoo table

5.2.3 Memory

Figures 5.7 and 5.8 show the bandwidth between L2 and L1, and L2 and L3 respectively. In both cases, 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 that more data will be required each instruction, and a higher volume of data will (transit) across the cache levels and the DRAM.

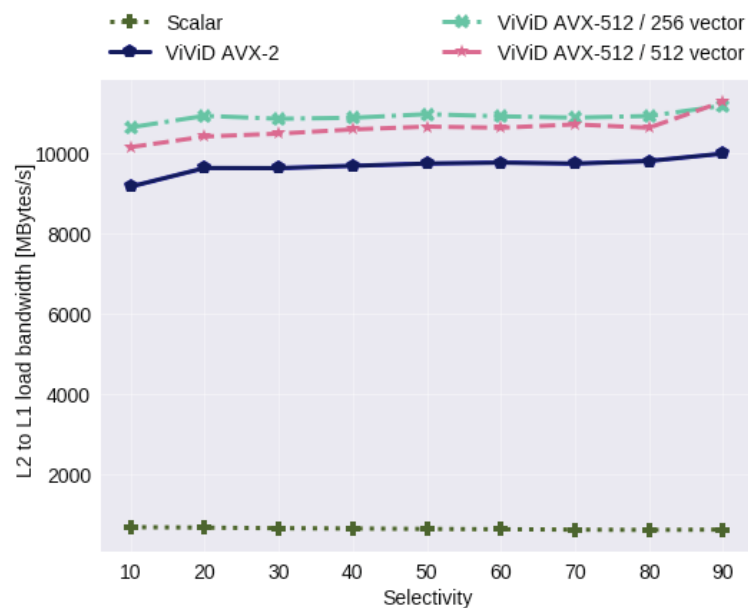


Figure 5.7: Load bandwidth of data from L2 to L1 cache on the build of a Cuckoo Table

Between the three ViViD variants, the AVX2 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 AVX2. It should be expected the 512-bit vector variant

to show the highest pressure. It is also interesting to note that the bandwidth between L2 and L1 remains stable, but grows almost linearly between L3 and L2 until the 60% of selectivity and then stabilizes.

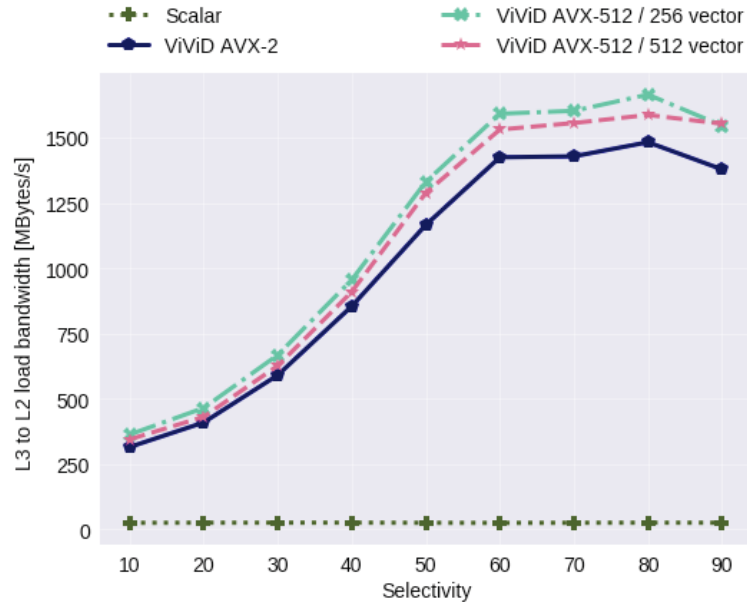


Figura 5.8: Load bandwidth of data from L3 to L2 cache on the build of a Cuckoo Table

5.2.4 Conclusion

The ViViD Cuckoo Hash shows considerable improvement over the scalar Cuckoo hash version in terms of throughput without raising the power consumption of the package. Although the pressure inflicted on the memory system caused an increase in power consumption on the DRAM, we do not consider this as a particular setback for the ViViD Cuckoo. The power consumed by DRAM is ten times lower than the power consumed by the package and the absolute amount of energy spent is ten times higher when using the scalar method to build a Cuckoo Table.

6 CONCLUSION AND FUTURE WORK

The use of Hash tables to perform joins between relations is a widespread technique to mitigate the high costs of the join operator. The so-called Hash Join does not demand multiple table scans and multiple comparisons between keys, mitigating the costs of nested loops and taking, on the best case, only one memory access to check for the existence of a key. However, hash tables are not perfect dictionaries and carry the problem of keys that collide, hashing to the same position. We can either try to avoid or deal with those collisions. In order to avoid collisions, the choice of a hash function that guarantees a proper distribution of keys along the table is vital. To deal with collisions, we can use methods as Chained Hashing - that creates chained lists to store the colliding keys - and Cuckoo Hashing. The second one is highly efficient, as stated in previous studies, guaranteeing at most two memory access to retrieve a key.

Cuckoo Hash has its bottleneck on the build process of the table. On this dissertation, we mitigated the costs of the build using several levels of parallelism. The most effective was the use of SIMD technologies to vectorize the build vertically and to transform control dependencies into data dependencies, culminating in the method we call *ViViD Cuckoo Hash*.

Experimental evaluation showed 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. However, ViViD Cuckoo increased the pressure over the memory system, since wider SIMD registers mean that more data is needed to perform each instruction. The pressure over the memory system also increased the power consumed by the DRAM. We did not consider this weakness to be significant since the power consumed by the package is almost ten times higher than the power consumed by the DRAM. ViViD Cuckoo enhanced the performance of the build phase of Cuckoo tables, maintaining the same power consumption of the scalar method and ten times less energy on average. Future work is required in the following:

- Investigation on the reasons why 512-bit registers implementations do not overcome the performance of 256-bit vectors implementations on AVX-512 implementation
- Use the ViViD method with other hashing strategies, such as Hopscotch and Robin Hood, and compare the performances with the ViViD Cuckoo
- Implement the ViViD Cuckoo Hashing to build Cuckoo filters, studying if there is some improvement over the scalar building and other filter structures already vectorized, such as Bloom filter.

REFERÊNCIAS

- Apache (2018). Apache cassandra documentation. <https://wiki.apache.org/cassandra/Partitioners>. Accessed: 2018-06-06.
- Appleby, A. (2016). Smhasher. <https://github.com/aappleby/smhasher/>. Accessed: 2018-03-22.
- Boncz, P. A., Zukowski, M., and Nes, N. (2005). Monetdb/x100: Hyper-pipelining query execution. In *Cidr*, volume 5, pages 225–237.
- Buxton, M. (2011). Haswell new instruction descriptions now available! *Intel Corporation*.
- 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):1179–1193.
- Celis, P., Larson, P.-A., and Munro, J. I. (1985). Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 281–288. IEEE.
- 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.
- Diefendorff, K. (1999). Pentium iii= pentium ii+ sse. *Microprocessor Report*, 13(3):1–6.
- elastic (2018). Elasticsearch documentation. https://www.elastic.co/guide/en/elasticsearch/reference/2.0/breaking_20_crud_and_routing_changes.html#_routing_hash_function. Accessed: 2018-06-06.
- 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):681–698.
- Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960.
- Fowler, G., Noll, L. C., Vo, K.-P., Eastlake, D., and Hansen, T. (2011). The fnv non-cryptographic hash algorithm. *Ietf-draft*.
- Garcia-Molina, H. (2008). *Database systems: the complete book*. Pearson Education India.
- Gholam, M. (2012). Raptordb documentation. <https://www.codeproject.com/Articles/190504/RaptorDB>. Accessed: 2018-06-06.
- Hennessy, J. L. and Patterson, D. A. (2012). *Computer architecture: a quantitative approach*. Morgan Kaufmann.
- Herlihy, M., Shavit, N., and Tzafrir, M. (2008). Hopscotch hashing. In *International Symposium on Distributed Computing*, pages 350–364. Springer.
- 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®.
- Jenkins, B. (2011). A hash function for hash table lookup. <http://www.burtleburtle.net/bob/hash/doobs.html>. Accessed: 2018-03-22.
- Li, X., Andersen, D. G., Kaminsky, M., and Freedman, M. J. (2014). Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, page 27. ACM.
- Lomont, C. (2011). Introduction to intel advanced vector extensions. *Intel White Paper*, pages 1–21.
- Nguyen, N. and Tsigas, P. (2014). Lock-free cuckoo hashing. In *2014 IEEE 34th international conference on distributed computing systems*, pages 627–636. IEEE.
- Noll, L. C. (2017). Fnv hash. <http://www.isthe.com/chongo/tech/comp/fnv/>. Accessed: 2019-06-09.
- Pagh, R. and Rodler, F. F. (2004). Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144.
- Peleg, A., Wilkie, S., and Weiser, U. (1997). Intel mmx for multimedia pcs. *Communications of the ACM*, 40(1):24–38.
- Polychroniou, O., Raghavan, A., and Ross, K. A. (2015). Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1493–1508. ACM.
- Polychroniou, O. and Ross, K. A. (2014). Vectorized bloom filters for advanced simd processors. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, page 6. ACM.
- PostgreSQL (2018). PostgreSQL source code. https://doxygen.postgresql.org/hashfunc_8c_source.html. Accessed: 2018-03-22.
- Ramakrishnan, 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. *Proceedings of the VLDB Endowment*, 9(3):96–107.
- Ross, K. A. (2007). Efficient hash probes on modern processors. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 1297–1301. IEEE.
- Russell, R. M. (1978). The cray-1 computer system. *Communications of the ACM*, 21(1):63–72.
- Silberschartz, A., Korth, H. F., and Surdashaan, 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 *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 207–216. IEEE.
- Zukowski, M., Héman, S., and Boncz, P. (2006). Architecture-conscious hashing. In *Proceedings of the 2nd international workshop on Data management on new hardware*, page 6. ACM.