IJRAR.ORG E-ISSN: 2348-1269, P-ISSN: 2349-5138

INTERNATIONAL JOURNAL OF RESEARCH AND ANALYTICAL REVIEWS (IJRAR) | IJRAR.ORG

An International Open Access, Peer-reviewed, Refereed Journal

Density And Latency Optimized SRAM-Based Hash Join Architecture

MENDI. MAHA LAKSHMI SATYA SRINIVAS[1], Mrs. K. JHANSI RANI[2]

[1]Student, M.Tech, Dept. of ECE, University College of Engineering (A) JNTUK Kakinada, Andhra Pradesh, India

[2]Assistant Professor, Dept. of ECE, University College of Engineering (A) JNTUK Kakinada, Andhra Pradesh, India

ABSTRACT

The primary objective of this innovative endeavour is to craft a supremely efficient hash join operator. In this undertaking, we introduce an avant-garde, non-collision parallel static random-access memory (SRAM)-based hash join architecture. This architectural marvel harnesses the power of multiple hash functions and content addressable memories (CAMs) to eradicate hash collisions, thus guaranteeing a consistently swift memory access during each phase of the hash join algorithm. Consequently, it augments the hash join throughput to an impressive degree. Hash joins, indispensable in the realm of relational database management systems, sorting, and aggregation, are witnessing heightened significance in the age of the Internet of Things (IoT) and Big Data. The voracious appetite for rapid query processing has become an imperative need for modern DBMS, as the conventional growth of the central processing unit (CPU) is insufficient to tackle the exponential surge in data volume. This has precipitated a pressing demand for pioneering processing methodologies to expedite database systems. Enter the ingenious non-collision parallel hash join strategy. This visionary approach not only confronts hash collisions head-on but also ushers in a paradigm shift by facilitating insert and query operations within the hash join algorithm, all with the assurance of constant-time efficiency. We culminate this groundbreaking strategy in a parallel hash join architecture, replete with multiple channels and a CAM, which artfully disperses tuples across different hash channels, obviating the need for redundant storage. The expansion of this concept, we unveil the non-collision parallel hash join strategy, enriched by the inclusion of multiple decoded hash tables, to further enhance critical parameters. Our methodology commences with the generation of one-hot coded units based on input length. An eight-bit input address is judiciously partitioned into multiple halves to pinpoint decoded nodes. The address decoding architecture is meticulously streamlined, and latency optimizations are meticulously implemented to expedite address searches, resulting in an exquisitely efficient system.

Keywords: Hash join, Static random-access memory (SRAM), Ternary content addressable memories (TCAM), Internet of Things (IoT), One-hot coded.

INTRODUCTION

In the current landscape dominated by the Internet of Things (IoT) and the deluge of Big Data, the imperative for expeditious query processing within the modern Database Management Systems (DBMS) is palpable. As an endeavour to bridge the chasm between computation and storage, myriad previous studies have embarked on the arduous quest to accelerate database operations through innovative hardware platforms. This quest has led to the exploration of diverse avenues, including the deployment of vector architectures [9], Application-Specific Integrated Circuits (ASICs) [18], Graphics Processing Units (GPUs) [10], or ingenious hybrid solutions. Some have ventured into the static utilization of Field-Programmable Gate Arrays (FPGAs) [15], [8], [3], [17], while others have harnessed the dynamic reconfigurability inherent in FPGAs to tailor them to the specific demands of complex queries [6], [13]. The industry, too, has not been idle, yielding notable products such as IBM Netezza [2] and Teradata Kick fire [14].

However, amid this cacophony of innovation, it remains abundantly clear that hash-based operations, specifically hash join and group-by, constitute the most time-intensive components of database query processing systems. Previous studies have unveiled the stark reality that these operations often devour over 40% of the total execution time, particularly when executing queries derived from the TPC-H benchmark.

The hash join operation, for instance, orchestrates the merger of two data tables, S and T, by means of a shared key. Its intricate algorithm unfolds in two crucial phases: the build phase, which orchestrates the construction of a hash table employing the rows of table S, and the probe phase, where the entirety of table T's keys are meticulously examined within the confines of the hash table to unearth potential matches. Likewise, the group-by operation plays the role of a custodian, diligently grouping the rows of a given table based on common values in the key column, a feat accomplished efficiently using hash tables.

Yet, lurking on the horizon like a tempestuous storm, is the ominous spectre of hash collisions—the inevitable situation where two distinct keys find themselves unjustly mapped to the same hash index. This conundrum poses a potent threat to database applications, necessitating deft handling. While software fallback mechanisms [17] or rehashing [7] may offer potential remedies, they introduce additional latencies, chipping away at performance. On the contrary, addressing collision issues within the hardware context often entails chaining hash table entries, a solution that, regrettably, can undermine hash table performance, particularly when confronted with Double Data Rate (DDR) memory latencies.

In light of the scarcity of on-chip Block RAM (BRAM) resources, which cannot guarantee the accommodation of an entire hash table, prior FPGA implementations resorted to constructing the hash table in off-chip DDR memory [17]. However, in a revolutionary departure from convention, this work introduces a novel hash table caching technique. It ingeniously leverages the on-chip BRAMs of FPGAs to mitigate memory latencies, all while gracefully sidestepping the need for software fallbacks in collision resolution.

In the crucible of evaluation, we subject the hash join and group-by operations of five queries from the TPC-H benchmark suite to rigorous testing. The results are nothing short of astonishing, with performance speedups reaching a staggering 4.4 times when compared to a hardware baseline devoid of any caching wizardry. This hardware baseline, an evolved iteration of Ibex [17], exemplifies the profound strides that can be made in enhancing database systems in an age where the central processing unit (CPU) struggles to keep pace with the exponential data surge. In this age of relentless data proliferation, where the CPU's growth lags behind the rising tide, the Field-Programmable Gate Array (FPGA) emerges as a beacon of hope—a robust contender for alleviating the computational burden borne by CPUs through the offloading of data-intensive, time-consuming database operations [1–4]. Among these operations, the join operator stands out as a voracious consumer of computing resources, tirelessly amalgamating tuples from multiple relations

by virtue of a common key. Consequently, the quest to expedite this operation using FPGAs has assumed paramount importance.

Two principal algorithms, the sort–merge join and hash join, take centre stage in this hardware acceleration endeavour. The sort–merge join [5–7], a method that first sorts tuples within each table based on a key and then performs the join operation by scanning each table just once, has attracted considerable research attention. Scholars have primarily focused on optimizing the sorting phase, yielding remarkable solutions such as Chen's parallel and pipelined biontic sorter on FPGA [6] and Casper's dedicated hardware solution for complete hardware-based merge sort [5]. These innovations achieve stellar data throughput, leveraging intricate sorting structures. Zhou et al. [7], meanwhile, have delved into accelerating the merge phase by introducing hierarchical indexes to identify result-yielding regions, thereby achieving commendable performance even at low match rates.

LITERATURE SURVEY

In the realm of cutting-edge networking technologies, Narayana et al. [2017] embarked on a journey to transform a programmable switch into a cache for aggregation operations. Their pioneering work delved into the intricacies of identifying and mitigating network issues, such as congestion, and offered ingenious solutions. Notably, while their efforts sought to enhance query operations over the network, they remained conspicuously silent on the topic of SD-WAN technology.

In a parallel dimension of research, Xiong et al. [2014] cast their gaze upon Software-Defined Networking (SDN) as a conduit for amassing data to optimize query plans. Meanwhile, the likes of Binnig et al. [2016] and Salama et al. [2017] set out to harness the burgeoning speed of networks to unveil novel distributed join algorithms. These algorithms promised to unlock the latent potential of network speed in a more resourceful manner. They also posited that the network, once considered a formidable bottleneck in executing distributed queries, was losing its grip on performance constraints. However, our exploration takes a different path. We embark on a quest to unlock the programmability inherent in SD-WAN, aiming to unearth the advantages of processing join operations within a switch. Importantly, our findings align with those of Binnig et al. [2016] and Salama et al. [2017], as we discern that communication time exerts a minimal influence on the ultimate outcomes of our test cases.

In yet another dimension, Jin et al. [2018] unfurled a mechanism orchestrating data storage within programmable switches, with a singular aim: executing every query plan within these network devices. Meanwhile, Lerner et al. [2019] unveiled NetAccel, a visionary system that offloads entire queries onto switches, harnessing the programmability within these devices to elevate the execution of analytical queries. This marvel sends data to the switch, employing packet-processing hardware known as the Match-Action Unit (MAU) to establish precise packet-field matching with rows in a table. Notably, NetAccel employs resources for SRAM storage and leverages hashing for matching. Furthermore, it introduces a novel set of operations geared toward expediting processing through switches. In contrast, our work takes a different approach, introducing a comprehensive evaluation encompassing various components that seeks to orchestrate query operations on the switch while scrutinizing network communications.

In a forward-looking perspective, Lerner et al. [2020] delved into the challenges and prospects of accelerating queries through network devices. They envisioned the switch as an active participant, housing an entire aggregation table to facilitate rapid data filtering. Their argument centered on the premise that refining algorithms could unlock the potential of emerging network technologies and curtail the movement of data across the network. This foundational work laid the cornerstone for our own research. Drawing inspiration from their insights, we propose a cost-model approach, seeking to evaluate hypothetical scenarios and pinpoint bottlenecks in network-based query processing.

Shifting gears to the world of computational algorithms, hash join implementations for CPUs have been meticulously explored in the annals of literature [10]–[12]. Notably, Schuh et al. [13] undertook an exhaustive examination of thirteen distinct join implementations for CPUs in 2016. However, this extensive study remained confined to the realm of CPUs, neglecting the burgeoning domain of GPU hardware and its associated implementations. In a similar vein, the hash join implementation devised by He et al. [1] in 2008 failed to harness the modern marvels of GPU architecture, such as atomic operations, unified memory, or multiple CUDA streams. The year 2012 marked a pivotal moment, with NVIDIA GPUs widely embracing atomic operations in global memory. It was in this context that Dan et al. [14] unveiled a simple yet ingenious non-partitioned hash join implementation, constructing hash tables using global memory atomic operations. Concurrently, Pirk et al. [15] sought to expedite foreign-key joins by executing random table lookups on GPU VRAM.

Fast forward to 2012, Gregg et al. [16] illuminated the profound impact of PCIe data transfer overhead on the overall performance of GPU applications. This juncture saw vendors like NVIDIA addressing concerns regarding data movement, ushering in features like universal virtual addressing (UVA) in 2010, enabling GPU kernels to access data directly from the CPU main memory. In the same year, Kaldewey et al. [6] masterminded a system that harnessed UVA and lock-update-release-based atomic instructions (supported since 2010) to share histograms among multiple threads.

Subsequent years witnessed a plethora of studies on databases in 2013 [2], [8], [17], [18] that explored non-partitioned hash join implementations. Yet, these implementations failed to surpass their hardware-conscious partitioned hash join counterparts. It was not until 2015 that Rui et al. [3] revisited the original hash join implementation, proposing enhancements such as the use of shuffle instructions (introduced in 2012) to expedite the prefix sum operation.

EXISTING METHOD

The Hash join method typically undertakes a twofold operation comprising a build phase and a probe phase. In the former, the initial table undergoes a meticulous scan, wherein the venerable hash function is wielded to meticulously populate a dedicated hash table with the relevant tuples. In contrast, the latter phase witnesses the scrutiny of the second table, as the hash table is methodically probed in pursuit of matching results.

In this context, the paper presumes that the first and second tables boast N and M rows, respectively, with N being less than M. These tables harbour tuples, integral components of the composite records that seamlessly flow into the join architecture, bearing a distinctive {rid, key} format. In this symbiotic dance, 'rid' serves as the unique identifier for each row, while 'key' takes on the pivotal role of the join attribute. As the second table makes its grand entrance, it engenders the creation of join results, aptly encoded as {rid1, rid2, key}.

1.1. Build Phase:

In the build phase, the FPGA orchestrates a synchronized retrieval of the first table's tuples, a symphony of parallelism that unfolds in a dazzling display of computational prowess. Each channel boasts a FIFO, artfully designed to cache the incoming influx of tuples, whether sourced from local storage or other channels. Here, the magic of hash functions comes into play, meticulously hashing the key of each tuple within its designated channel. The calculated location of each tuple is then stored within the corresponding address on the hash table, a symphony of data orchestration.

These tuples find a haven within the hash tables, their format elegantly encapsulated as {status, rid, key}. Within this format, the 'status' field, a mere 1-bit wonder, performs a pivotal role, succinctly indicating whether the row within the hash table has been claimed by an occupant. However, as the saga unfolds, and

hash collisions rear their formidable heads, conflicting hash values within each channel are gracefully shifted to the next available channel, ensuring an elegant and efficient resolution. This shift process persists until a resolution is reached, marked by the successful insertion of the tuple into one of the hash tables or the attainment of the shift threshold, symbolized by the channel number. Once this threshold is breached, the tuple embarks on a journey, finding its new abode within the CAM, all while donning the {rid, key} format with pride.

As depicted in Fig. 1, an ensemble of hash channels graces the stage, each channel endowed with its dedicated resources. These resources facilitate the simultaneous distribution of the first table's contents in a dazzling display of parallelism during the build phase. Within the confines of each hash channel, a non-conflicting tuple demands a mere two clock cycles to orchestrate an update within the hash table, deftly discerning the availability of the corresponding address and executing the data write.

However, should the fateful moment of hash collision befall, the conflicting tuples are ushered into the FIFO, where they patiently await their turn for further processing. Thus, the memory access time within each hallowed hash channel remains an unwavering constant, a testament to the unwavering efficiency that permeates the build phase.

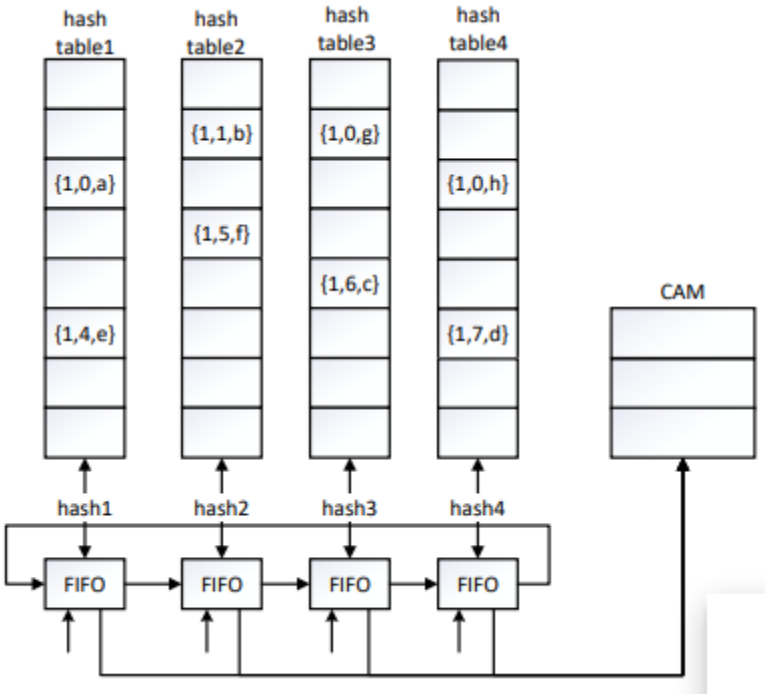

Figure 1: An Illustration of a Hash Join Architecture with Four Hash Channels and a Content-Addressable Memory (CAM), Housing Eight Tuples (a–h). Each row within the hash table boasts a triumvirate of elements: {status, rid, key}, where 'status' serves as the sentinel, diligently signalling the occupancy status of the row.

1.2. Probe Phase:

As we traverse the probe phase, the spotlight shifts to the second table, where a stream of tuples gracefully unfurls, poised for an intricate dance of comparison with their counterparts in the first table, a dance that seeks the elusive harmony of matching results. Upon the discovery of such a match, the two rows engage in a harmonious union, their union beautifully rendered in the {rid1, rid2, key} format. These tuples from the second table embark on a journey through the hash channel, elegantly orchestrated within the pipeline, with multiple hash channels operating in symphonic parallelism.

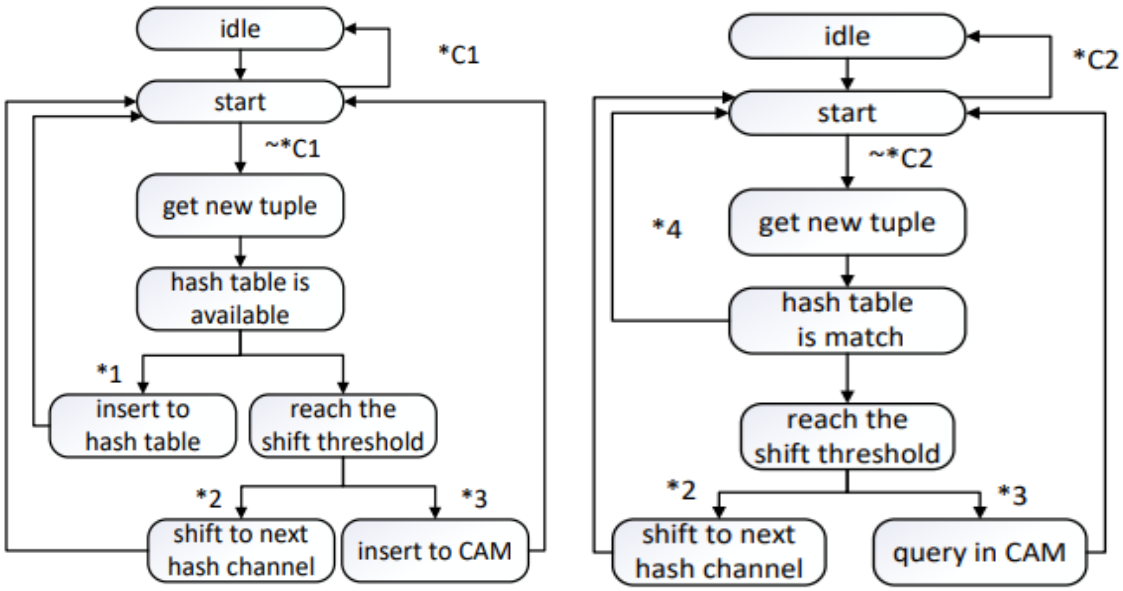

Figure 2: Behold the Finite-State Machines Unveiled for Both the Build and Probe Phases, where the zenith of access time woes is encountered. This ingenious strategy, which weaves its magic seamlessly during both phases, finds its genesis in the realm of Finite-State Machines (FSMs), elegantly depicted in Figure 2.

Much like the build phase, the keys from the second table traverse these numerous hash channels in a synchronized ballet. Within each hash channel, the tuple finds its destined path to a specific address, deftly calculated by the hash function unique to that channel, in the fervent quest for a match. This synchronized ballet unfolds concurrently across all channels, creating an elegant pipeline of simultaneous operations. Should a match be fortuitously stumbled upon in one of the channels, a join result is swiftly generated, eagerly awaiting further processing. However, if the tuple's journey through the channels yields no match, it gracefully shifts to the next channel, continuing its quest for the elusive counterpart.

In cases where a tuple exhaustively explores all available channels without finding a match, it embarks on a final journey, venturing into the CAM for a query operation. In the event that all channels and the CAM collectively declare a mismatch, the forsaken tuple meets its ultimate fate and is promptly discarded. Notably, our paper zooms in on the intriguing realm of the "N-to-1" join relationship, a scenario where, upon finding the key of the second table that aligns with its counterpart, the search process concludes with a sense of fulfilment. Throughout this intricate dance, certain components within each channel, designed to be highly efficient, require a consistent number of cycles to execute, ensuring a seamless flow without the dreaded stalls that can disrupt the pipeline. Furthermore, when it comes to exact table matching operations performed on CAMs, they elegantly complete within a single clock cycle. Consequently, the memory access time remains a steadfast constant during this captivating probe phase.

1.3. Data Shift Strategy:

At the core of our ingenious design lies the brilliant notion of offering multiple hash channels, a veritable ensemble of pathways that collectively orchestrate the distribution of the first table. This distribution takes place in the company of a compact CAM, a strategic pairing that ensures that even the tuples destined to remain unclaimed by all hash tables find their rightful abode within the CAM. The result is a meticulously crafted non-collision hash join scheme that harmoniously melds efficiency and elegance.

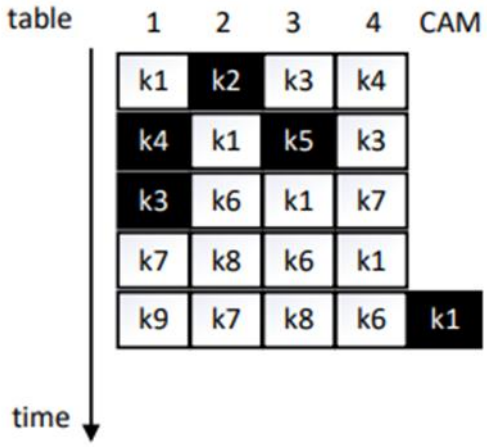

Figure 3: Data shift strategy for build phase and probe phase

In Figure 3, we unveil an illustrative timeline, a mesmerizing visualization of the memory access operations that underpin this ingenious data shift strategy. As the narrative unfolds, during the initial time slot, items denoted as K1, K2, K3, and K4 gracefully stream through the ether. Within this symphony of movement, K2 finds its rightful place, simultaneously inserted or probed. The subsequent time slot ushers in a shifting of the remaining items, K1, K3, and K4, while a newcomer, K5, confidently steps into the spotlight, supplanting its predecessor. The dance continues, with K4 and K5 finding their rightful places, inserted or probed successfully.

In the third time slot, the stage is set for two new entrants, K6 and K7, to join the performance. However, the fourth time slot remains a moment of respite, devoid of successful insertions or probes. Finally, as the curtains draw to a close in the fifth time slot, K1 takes its final bow, gracefully moving to the CAM. This shift is prompted by the attainment of the shift threshold, signalled by the hash channel number, marking the culmination of a beautifully orchestrated sequence.

2. FIFO ARCHITECTURE:

Imagine a delay buffer as a versatile conductor in an orchestra, akin to a finely-tuned jitter buffer. Its mission? To choreograph the retrieval of frames, gracefully orchestrating pauses to ensure that the caller receives a seamless stream of frames from the buffer. This ingenious mechanism proves its mettle when operations aren't evenly interleaved, such as when a caller unleashes a barrage of put() operations followed by a cascade of additional tasks. In the presence of this delay buffer, the burst frames find a temporary abode within its confines, patiently awaiting their turn to shine during get() operations. The beauty of this arrangement lies in its equilibrium; the buffer diligently ensures that each put() operation is matched by a get() counterpart, creating a harmonious rhythm.

But here's the magic: this buffer is no ordinary buffer. It's adaptive, a quick learner in the school of run-time optimization. Once it has deciphered the optimal delay required to conduct the audio flow with finesse, it applies this newfound wisdom to the audio stream. Like a maestro adjusting the tempo, it gracefully expands or contracts the audio samples as needed, maintaining the perfect cadence even when the audio samples within the buffer sway towards excess or scarcity.

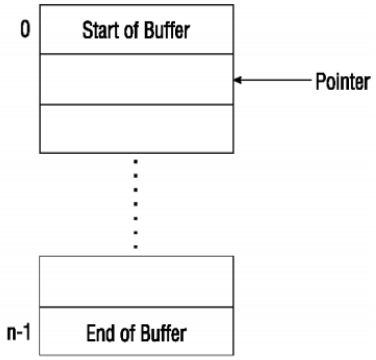

Figure 4: Buffer

3.TECHNIQUE:

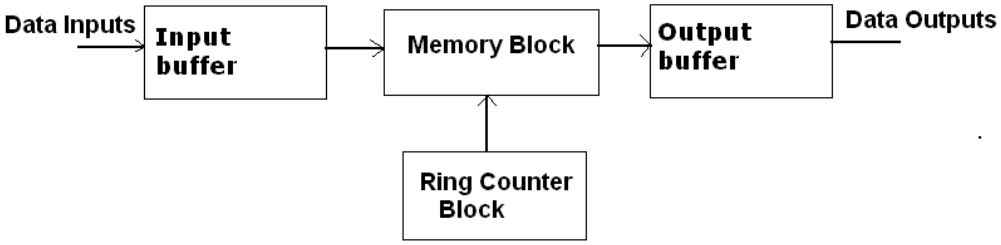

Figure 5: Memory Organisation

PROPOSED ARCHITECTURE

MEMORY ACCESS UTILIZING ONE HOT DECODER:

The art of memory access and one-hot decoding converge in a symphony of data categorization, where the orchestration is aimed at enhancing the predictive prowess of machine learning algorithms. Picture this: we embark on a transformative journey of converting categorical values into distinct columns, a process akin to giving each column a binary identity—either a 0 or a 1. These integer values, now elegantly represented as binary vectors, breathe life into the data.

Now, let's delve into the heart of computing, the hallowed realm of memory, known to many as RAM, or random-access memory. Think of your computer as a treasure trove, with 16, 32, or perhaps even 64 megabytes of RAM nestled within its confines. RAM becomes the canvas upon which programs paint their masterpieces, holding both the programs themselves and the very data they Mold and manipulate—variables, and data structures.

Conceptually, memory unfurls as an array of bytes, each byte with a unique address. The journey begins with the address of the first byte, a humble 0, followed by its sequential brethren, 1, 2, 3, and so forth. These memory addresses assume the role of stalwart sentinels, mirroring the indexes of a conventional array. Remarkably, the computer wields the power to access any address within this memory array at any given moment, hence the illustrious moniker, "random access memory."

What's more, the computer displays a flair for artistic arrangement, seamlessly grouping bytes as needed, weaving them into larger tapestries of variables, arrays, and intricate data structures. For instance, a floating-point variable takes residence in a contiguous expanse of 4 bytes within this memory landscape. In this enchanting dance of data and memory, we witness not only the transformative magic of one-hot decoding but also the intrinsic beauty of memory's organization, where every byte is a note in the grand symphony of computing.

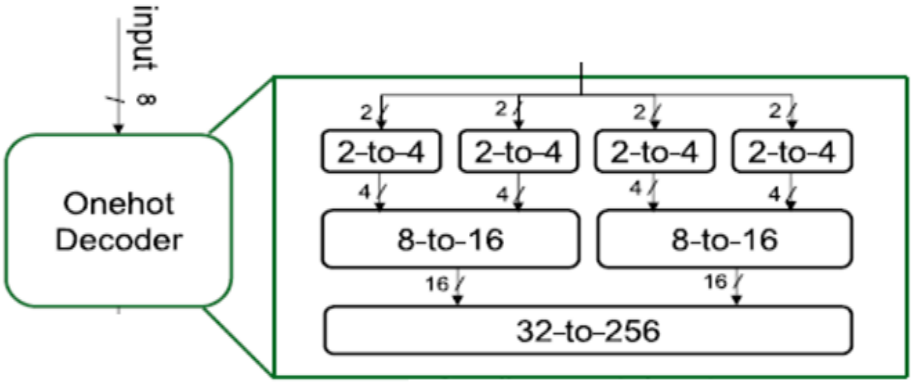

Figure 6: Memory one hot decoding architecture

Content-addressable memory (CAM), an architectural marvel etched into silicon, emerges as a purpose-built sanctuary for lightning-fast memory inquiries of a highly specialized nature. Think of it as a maestro of memory, orchestrating lookups with a conceptual kinship to the associative array logic found in data structures. However, the brilliance of CAM lies in its ability to distil the complexity, offering outputs that are elegantly streamlined. Here's the enchanting part: when a key grace the doorstep of a CAM sub-system, it opens the gates to a realm where the associated value eagerly awaits. In this digital waltz, a captivating "key -> value" pair materializes, akin to the creation of a precious object, ready to be summoned at will.

However, the true magic of CAM unfolds in the realm of speed. Unlike its RAM counterpart, which demands the passage of multiple clock cycles to orchestrate a single memory retrieval, CAM operates with an unprecedented efficiency. A mere flicker of a single clock cycle in the silicon, and the desired entry is unveiled—a stark contrast to the languid pace of traditional RAM modules. In this realm of memory, where time is of the essence, CAM reigns supreme as the embodiment of swift and precise memory retrieval.

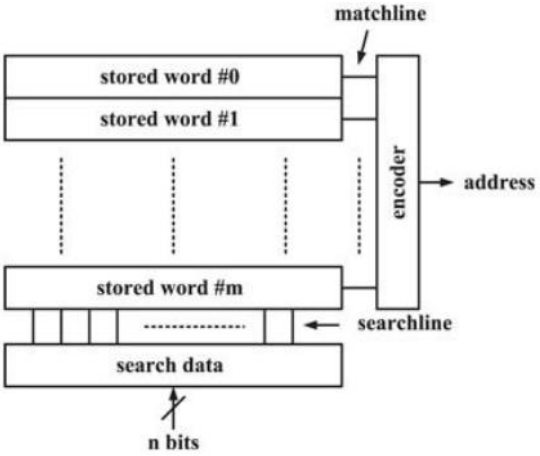

Figure 7: Conceptual View Diagram of CAM

Behold the conceptual vista, elegantly illustrated in Figure 5, offering a glimpse into the inner workings of the Content-Addressable Memory (CAM). Here, within the hallowed confines of CAM, a trove of m data words finds its sanctuary, a repository where knowledge resides. Now, picture the search word, a nimble n-bit input data, gracefully traversing the search lines, a grand procession of information, destined for a simultaneous rendezvous with the table of stored words. This symphony of comparison unfolds in real-time, as each search word embarks on a quest to find its kindred spirit among the stored data.

Within this ballet of data, a match line stands sentinel for each stored word, a sentinel poised to sound the clarion call, signalling the fateful encounter—either a triumphant match or a sombre mismatch. These match lines, like vigilant gatekeepers, await their moment of revelation. But here's where the intrigue

deepens: the match lines, each harbouring its own verdict, converge at the gates of an encoder, a digital arbiter of fate. With precision, the encoder deciphers the binary location that corresponds to the match line, effectively rendering judgment in Favor of the match case.

In scenarios where multiple match lines sing the song of victory, a hierarchy emerges, and the priority encoder ascends to the stage. With unwavering authority, it bestows its blessing upon the highest priority match line, revealing the sacred address location that aligns with this momentous match. In this enigmatic world of CAM, where data and destiny intertwine, every search is a quest for connection, every match line a herald of truth, and every encoder a master of deciphering the tapestry of memory.

RESULTS

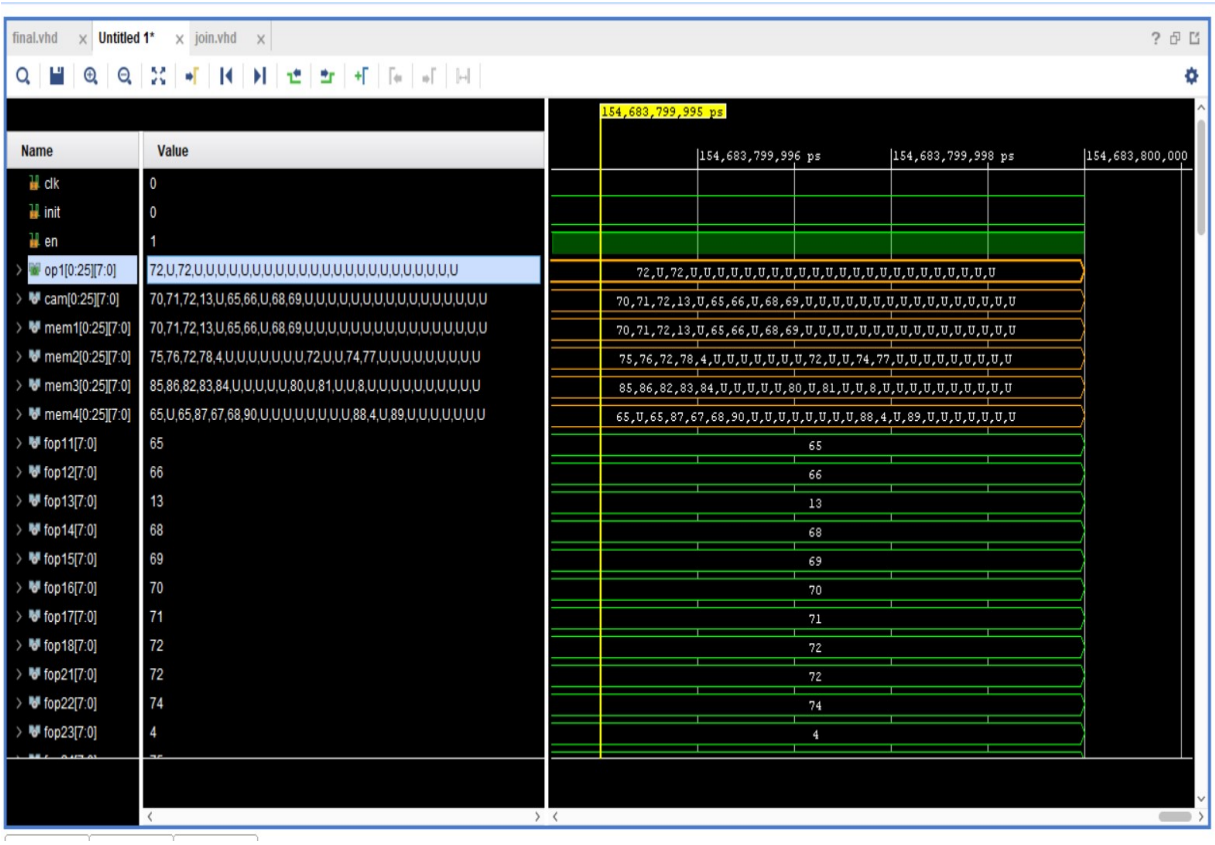

Figure 8: Proposed Simulation result

In this section, the proposed results displayed above are generated using XILINX Vivado 2018. The input signal 'clk' is synchronized with a rising edge represented as '1', while the falling edge is indicated as '0'. A comparison operation is performed between selected memories, and their shared outcomes are stored in 'op1'. And remaining values are then stored in CAM.

COMPARISON WITH PREVIOUS SYSTEM

Parameter	Existing method	Proposed method
Power (watts)	3.59 W	2.702 W
Delay (ns)	43.514 ns	41.544 ns
LUTs (slice)	231	55
LUTs (registers)	577	159

Figure 9: Comparison results

CONCLUSION

In this undertaking, we have embarked on a journey of technological ingenuity. We have introduced a multitude of hash channels, each serving as a conduit for distributing the same table. These channels operate in harmonious synchrony, processing tuples in a pipeline fashion, all the while maintaining parallel momentum. Enter the diminutive yet formidable CAM, a guardian against the tumultuous waters of hash collisions. It stands poised, offering refuge to those tuples that find themselves excluded from all channels, ensuring that no data is left adrift. To enhance the efficiency of FPGA, a shrewd data shift strategy has been deployed during both the build and probe phases, skilfully minimizing the potential for stalling. Our architectural marvel bestows the coveted gift of O(1) memory access time upon the hash table in each phase, coupling it with the swift efficiency of a constant-time CAM insert operation during the build phase.

In the probe phase, the CAM performs with the grace of a ballet dancer, completing its search operation in a single clock cycle. These innovations are orchestrated to elevate the hash join throughput, creating a landscape where the worst-case query time remains deterministic. Yet, the story does not conclude here. Our exploration extends further, introducing an extended, modified memory accessing scheme that promises parameter optimization beyond the confines of existing methods. In the realm of hash joins, we have not merely devised a solution; we have crafted a symphony of technological advancement, painting the canvas of possibility with vibrant strokes of innovation.

FUTURE SCOPE

The horizon of possibilities in this realm extends far and wide, with tantalizing prospects awaiting exploration. One of the noteworthy enhancements lies in our ability to scrutinize critical system parameters and their intricate interplay with performance, thereby illuminating a path toward future hardware advancements. A promising trajectory unfolds as memory bandwidth for FPGA stands poised for expansion, leveraging the dual forces of enhanced FPGA frequencies and a burgeoning array of memory channels. Additionally, the ascent of High Bandwidth Memory (HBM) holds the potential to usher in a new era of memory capabilities.

The canvas of on-chip RAMs, a fundamental component in the tapestry of technological progress, continues to evolve, growing larger with each passing decade. For instance, consider the Xilinx Ultra Scale+ devices, boasting a staggering 62.5MB of on-chip RAMs—a testament to the boundless possibilities that await. The OpenCL SDK for FPGA, a pivotal player in this symphony of innovation, stands on the brink of transformation. It promises to emerge as a formidable force, delivering not only better timing results but also an optimal utilization of resources. As we peer into the future, the stage is set for a grand performance, where the harmony of technology and innovation will continue to echo, unveiling new horizons and pushing the boundaries of what is conceivable.

REFERENCES

[1] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, “Relational joins on graphics processors,” in SIGMOD, 2008.

[2] J. He, M. Lu, and B. He, “Revisiting co-processing for hash joins on the coupled cpu-gpu architecture,” Proc. VLDB Endow., 2013.

[3] R. Rui, H. Li, and Y. C. Tu, “Join algorithms on gpus: A revisit after seven years,” in ICBD, 2015.

[4] R. Rui and Y.-C. Tu, “Fast equi-join algorithms on gpus: Design and implementation,” in SSDBM, 2017.

[5] M. Yabuta, A. Nguyen, S. Kato, M. Edahiro, and H. Kawashima, "Relational joins on gpus: A closer look," TPDS, 2017.

[6] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, "Gpu join processing revisited," in DaMoN, 2012.

[7] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili, "Kernel weaver: Automatically fusing database primitives for efficient gpu computation," in MICROArch, 2012. [8] J. He, S. Zhang, and B. He, "In-cache query co-processing on coupled cpu-gpu architectures," Proc. VLDB Endow., 2014. [9] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki, "Hardware-conscious Hash-Joins on GPUs," ICDE, 2019.

[10] S. Blanas, Y. Li, and J. M. Patel, "Design and evaluation of main memory hash join algorithms for multi-core cpus," in SIGMOD, 2011.

[11] J. Teubner, G. Alonso, C. Balkesen, and M. T. Ozsu, "Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware," in ICDE, 2013.

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

[13] S. Schuh, X. Chen, and J. Dittrich, "An experimental comparison of thirteen relational equi-joins in main memory," in SIGMOD, 2016.

[14] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. Owens, and N. Amenta, "Building an efficient hash table on the gpu," in GEMS, 2011.

[15] H. Pirk, S. Manegold, and M. Kersten, "Accelerating foreign-key joins using asymmetric memory channels," in ADMS, 2011.

[16] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in ISPASS, 2011.

[17] Y. Yuan, R. Lee, and X. Zhang, "The yin and yang of processing data warehousing queries on gpu devices," Proc. VLDB Endow., 2013.

[18] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl, "Hardwareoblivious parallelism for in-memory column-stores," Proc. VLDB Endow., 2013.