Validating Efficient Single GPU Join Algorithm

Yezheng Shao
University of Toronto
Toronto, Canada
yezheng.shao@mail.utoronto.ca

Rongxuan Du
University of Toronto
Toronto, Canada
rongxuan.du@mail.utoronto.ca

ABSTRACT

Efficient relational Join algorithm is one of competitive advantages that database communities are pursuing. Modern graphics processing units (GPUs) provide massively parallel computing and has been proved to enhance the performance of such join algorithm on scaling-out data set. However, the earlier GPU algorithm cannot make most use of the modern GPU architecture that included larger register file and shared memory, native atomic operation, dynamic parallelism , and CUDA streams. New GPU algorithm designed by Rui and Tu [8] claimed a 10.5x speedup in comparison to the best CPU sort-merge join known to the date and be able to process large data tables that cannot fit in the GPU memory. Our group revisited the experiment and validated the algorithm on newer GPU since it has been six years after the new algorithm was designed. The revisit showed a 14x speedup in comparison to CPU join algorithm on a more recent GPU and CPU architecture. We also tried to improve the algorithm by implementing cuda stream to hide the memcpy latency between host and device.

ACM Reference Format:
Yezheng Shao and Rongxuan Du. 2018. Validating Efficient Single GPU Join Algorithm . In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In recent years, the GPU hardware design has changed drastically. The number of computing cores increase about 30X times. Also, the GPU memory space increases substantially. As a result, people starts to explore the potential of GPU join. We look specifically into the sort-merge join (SMJ) algorithm. The algorithm focus on allowing more work per thread. The main component of the algorithm is called Merge Path. It is used in both sort and merge stage. The path is used to partition the data into balanced sections that each section can be worked by a CUDA thread independently. We will discuss more details in the next section. Our experiment shows that GPU SMJ outperforms the CPU SMJ by a 14x speed-up, which matches the result from the original paper.

Conference'17, July 2017, Washington, DC, USA

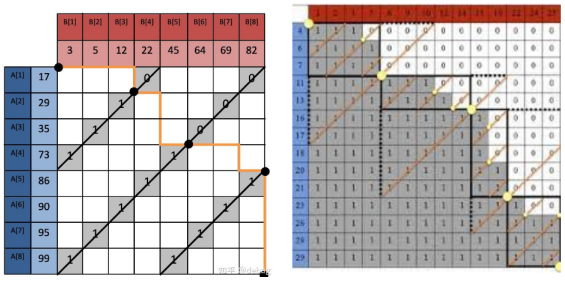
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 DESIGN

2.1 Merge Path

The main idea of the algorithm is to find the path of merge decisions of two sorted arrays A and B [6]. An important step is to find the path without actually merging it. We obtain the merge path by using parallel binary search to find its intersection between the cross-diagonals and find the starting and ending point of each section.

After finding out the starting and ending points, we use orange cross diagonals to cut the data into equal-size chunks, each chunk is assigned onto CUDA blocks. In each block, more cross diagonals are used to cut the data into smaller chunks. These chunks are assigned onto CUDA threads and serial joins can be implemented concurrently. Since each section is equal-sized, the load balancing is naturally achieved.



2.2 Limitation

This algorithm was first introduced in 2017. Both CPU and GPU architecture has changed significantly in recent years. It is unclear to know how much improvement that current GPU can achieve. Therefore, our objective in this paper is to evaluate the performance of this algorithm under today's GPU and CPU architecture.

3 METHODOLOGY

We choose a pair of Intel CPU and NVidia GPU to represent mid-range consumer market PC performance, the specifications of which are listed in Table 1. Ampere RTX3070 brings higher GFLOPS (5x) and clock rate (2x) in comparison to Kepler Titan GPU architecture. Additionally, Ampere RTX3070 supports PCI-E 4.0 16x interface while Kepler Titan uses PCI-E 3.0 16x interface. The CUDA code is compiled under NVCC 12.0 and we used the NVIDIA profiler called NSight Systems 2023.2.1 to study the runtime characteristics of the CUDA code. To compare with the CPU algorithm, multithreading is used to obtain the maximum performance on CPUs.

The validation process is to run the parallel join algorithm on different size of data chunk that ranges from 10 KB to 500 MB and use NSight system to monitor the time spent on different CUDA

Device	E5-2670	i7-11700F	Titan	RTX3070
Clock Rate	2.6 GHz	2.5 GHz	0.84 GHz	1.5 GHz
Core Count	8	8	14 x 192	64 x 92
L1 Cache L2 Cache L3 Cache	256 KB 2 MB 20 MB	80 KB 512 KB 16 MB	64 KB	128KB
Memory	64 GB DDR4	128 GB DDR4	6 GB DDR5	8GB DDR6
Memory Bandwidth	51.2 GB/s	50 GB/s	288 GB/s	488 GB/S
GFLOPS	166.4	583	4494	20310

API calls, size of data throughput, and PCIE transfer rate. Total time of CPU join is calculated by inserting timestamps at the beginning and the end of the CPU join calls. This straight-forward method visualizes the speedup of GPU vs CPu explicitly but also comes with some drawbacks. The data size only contains int value and cannot represent complicated table content in the real case. Secondly, we observe the upper bound of PCIE transfer rate when it reaches the steady state, such that the result is an approximate value. Instead of observation, a roofline model with sufficient large data set can be used to predict the upper performance of the parallel algorithm in the future.

4 RESULTS

4.1 Time usage

We run our experiment on different data size and use the NVIDIA Nsight to monitor the GPU usage.

In figure 1 and 2, we can see that when the data size is small (10K data), about 62.4% of the time in cuda is used by kernel operation, and only 37.6% is used for data transfer. However, as the data size increases to 1M, the majority of the time (95.6%) is consumed by data transfer, (memcpy). The increase in Kernel join time is disproportional to the increase in data transfer.

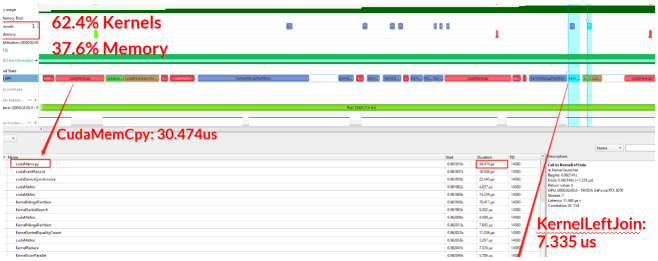


Figure 1: NSight System on 10K data

4.2 Throughput

In figure 3, we show the change in data throughput as data size increase. As the data size exceed 40M, the throughput stops increasing and stays around 22GB/s. We found that this bottleneck is relevant to limitation in the PCIe bandwidth. We used PCIe4.0 that produced in 2017, which has a maximum bandwidth of 32GB/s. We believe this limitation is one of the reason that prevent the algorithm from reaching higher throughput.

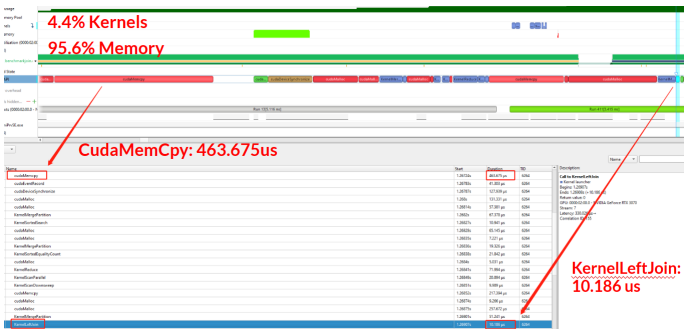


Figure 2: NSight System on 1M data

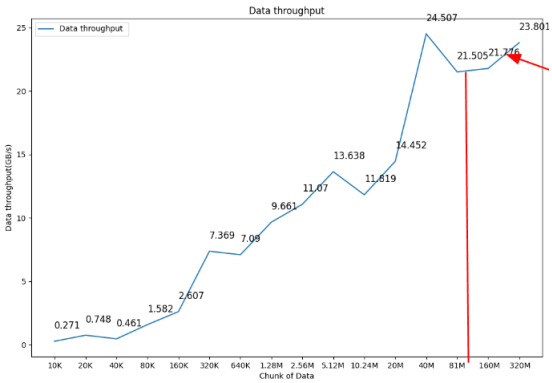


Figure 3: Data Throughput

4.3 CPU vs GPU

In figure 4(CPU VS GPU), we compare the execution time between CPU code and GPU code. We found that when the data chunk is small (about 20M), there is no improvement in the execution time. This is mainly caused by the CudaMallow overhead. Cuda requires time to allocate space for data to transfer from host to device. After the data size increases to 100M, we see a 13.4x - 14.6x speed up. However, the speedup is bounded due to the limitation at the PCIe bandwidth.

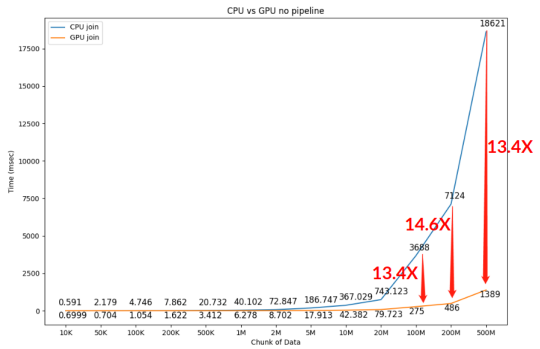


Figure 4: CPU VS GPU

5 RELATED WORK

Implementing parallel join algorithm and other operators on GPU has been an active topic in the database area.

On CPU side, Kim et al [4] optimized hash join and sort-merge join algorithm on a Core i7 processor by using SIMD instructions. They concluded the trend of wider SIMD instructions brings more potential scalability to sort-merge join. Years later, Balkesen [2] made a counterclaim to Kim et al's that hash join outperforms sort-merge join and the sort-merge join catches up only when the data set is large enough. Indeed, many selective parameters (table size, tuple sizes, the underlying architecture, using sorted data, etc.) affect the performance and hardware-conscious optimization is necessary to achieve a satisfying performance on CPU. Albutiu et al. [1] proposed a NUMA-affine parallel sort-merge join based on partial partition-based sorting. Expensive cross-region communication can be avoided as all the sorting is carried out on local memory partitions

On PCIE side, Lutz et al. [5] compared NV-link to PCIe 3.0 and shows a 18x speedup (or a 9x speedup on PCIe 4.0). With this speedup in transfer rate, the bottleneck can be shifted from the interconnection to GPU memory throughput or calculations

On GPU side, He et al [3] designed four parallel join algorithms that were suitable for early CUDA-enable GPUs. Specifically, a 2.4x speedup was achieved when transferring sort-merge join algorithm onto GPUs. Several years later, Tu [8] improved He's work and adapt the algorithm to the newer GPU architecture with larger global memory and shared memory, and native atomic operations. His work concluded that the GPU was 5-10x faster than the state-of-the-art implementation on the CPU. Tu [7] also demonstrated a multi-GPU version algorithm named global sort-merge join that boost a 2.8X speedup compared to single GPU version. Panagiotis [9] implemented a hardware-conscious join algorithm and concluded that the throughput was close to the memory bandwidth limit when the tables are located in device memory.

6 FUTURE WORK

During our experiments, we found that the algorithm is significantly bounded by the PCIe bandwidth. One solution is to improve the PCIe through hardware design. However, the data transfer between device is still gonna be a huge overhead when the data size is large. We estimate GPU speedup after mask the memcpy time in cuda. As shown in figure 5, GPU can achieve a 70x speedup when there is no memcpy overhead. To achieve this, we propose to pipeline the algorithm using cuda stream. By creating independent stream and operating concurrently, we believe that we can hide some of the data transfer overhead as shown in fig.5(desired outcome).

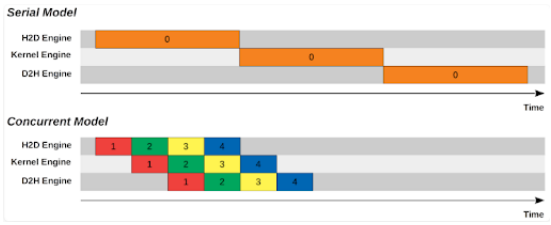


Figure 5: Cuda Stream

REFERENCES

[1] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *Proc. VLDB Endow.* 5, 10 (jun 2012), 1064–1075. https://doi.org/10.14778/2336664.2336678

[2] G. Alonso C. Balkesen, J. Teubner and M. T. Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. (2013), 362–373. https://doi.org/10.1109/ICDE.2013.6544839

[3] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational Joins on Graphics Processors *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 511–524. https://doi.org/10.1145/1376616.1376670

[4] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proc. VLDB Endow.* 2, 2 (aug 2009), 1378–1389. https://doi.org/10.14778/1687553.1687564

[5] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. *Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects.* Association for Computing Machinery, New York, NY, USA, 1633–1649. https://doi.org/10.1145/3318464.3389705

[6] Robert McColl Oded Green and David A. Bader. 2012. GPU Merge Path: A GPU Merging Algorithm. , 331-340 pages.

[7] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. Efficient Join Algorithms for Large Database Tables in a Multi-GPU Environment. *Proc. VLDB Endow.* 14, 4 (dec 2020), 708–720. https://doi.org/10.14778/3436905.3436927

[8] Ran Rui and Yi-Cheng Tu. 2017. Fast Equi-Join Algorithms on GPUs: Design and Implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management* (Chicago, IL, USA) *(SSDBM '17)*. Association for Computing Machinery, New York, NY, USA, Article 17, 12 pages. https://doi.org/10.1145/3085504.3085521

[9] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 698–709. https://doi.org/10.1109/ICDE.2019.00068