Implementation of the Bloom Filter on GPU using CUDA*

Venkatesh Rao†
Electrical Engineering Department
University of Minnesota Twin Cities, USA
raoxx136@umn.edu

ABSTRACT

The project aims to port the Bloom Filter data structure on to the GPU using Nvidia's Compute Unified Device Architecture (CUDA). Bloom Filters are popularly used in Distributed Systems, Database Applications and Networks. The inherent parallelism in the construction and querying of the Bloom Filter is a strong motive to use the GPU's computation power for achieving speed up. This would be crucial for several applications like Data deduplication where one could avoid going to the disk by use of a Bloom Filter. This project proposes an algorithm to achieve this goal and presents the corresponding experimental results.

1. INTRODUCTION

Bloom Filter is a probabilistic data structure that supports set membership query. If a key is present then a query returns true. But a key that is not inserted may return true upon query. Thus there exist false positives and no false negatives. A Bloom filter consists of k hash functions that insert n keys into an m bit array. For a given key, k positions given by $h_i(k)$ are set in the m bit array. For querying, the k positions are checked. If any of them is 0, then the key is reported as 'absent' else reported as 'present'. The process is shown in figure 2. A key could be queried or inserted but not deleted. (Exception: Counting Bloom filters). The optimal value of k is given as:

$$k = \frac{m*ln(2)}{n}$$

For the above value of k, the false probability rate (fpr) is given as:

$$fpr = \frac{1}{2^k}$$

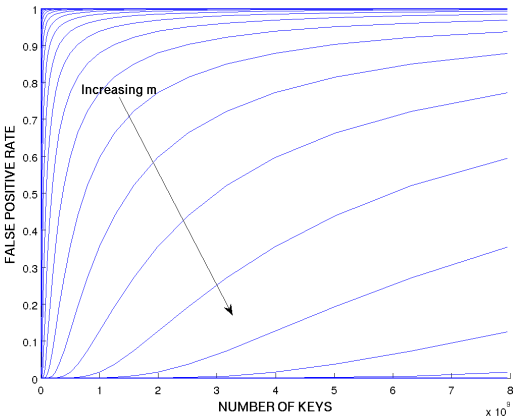
There is a linear relationship between the number of keys n and the number of filter bits m is given by:

$$n = \frac{-m*ln(2)^2}{ln(fpr)}$$

*The author is thankful to Prof. Weijun Xiao and PhD candidate Biplob Debnath for their guidance and feedback on this project. This project was completed as part of course requirement for EE5940 GPU Computing, May 2010 at the University of Minnesota Twin Cities.

†Student ID: 4114086

Figure 1: This plot depicts the variation of fpr with n for different values of filter size m.



The fpr decreases as m increases or as n decreases. Figure 1 shows the variation of n with fpr. The filter thus represents interesting trade-offs that are exploited as per the application[3]. There are several variants of the Bloom Filter that researchers have come up with suited to specific applications. This paper however deals with porting the generic Bloom Filter on to the GPU.

2. MOTIVATION

Bloom Filters have several applications in Distributed Systems, Database and Networks. They were traditionally used in Unix Spell Checkers. Instead of storing the entire dictionary of words in the memory, which was scarce in those days, a Bloom filter for the dictionary was stored in the memory. Bloom filters also find use in querying Distributed Databases like a company storing the employee information, where one host might have the employee salaries and the other storing the employees' addresses [15]. Google uses the Bloom filter in its distributed storage system called 'BigTable'. Data Deduplication algorithms too, benefit from Bloom filters. An expensive disk look up could be avoided by storing a corresponding Bloom filter in cache memory. The use of Bloom filters for Web caching was demonstrated in [6]. When a proxy gets the request for a certain webpage (that result in a cache miss), it queries the other proxies to search for the page instead of accessing the web. For this to work, each proxy must know the cache contents of the other proxies.

Figure 2: Bloom Filter Construction/Query process.

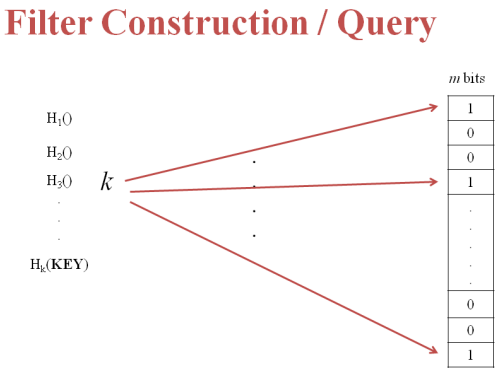
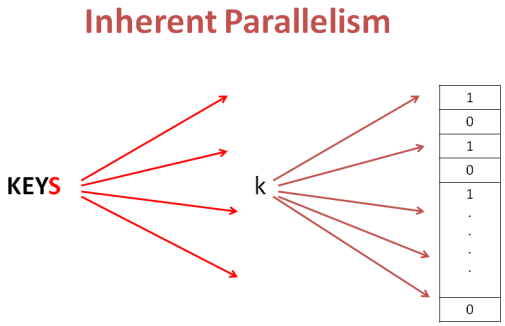


Figure 3: Parallelism in the Bloom filter operations.



Due to network delays it is not possible to broadcast entire cache contents for each proxy. Hence each proxy periodically broadcasts a Bloom filter representing its cache contents. So if a proxy needs to check if a particular page is present in its neighboring proxy, it simply queries the Bloom filter. Although a false positive condition does result in unnecessary overhead, but then the benefit associated with the implementation far outweighs this. [4] gives a thorough review of the various Bloom filter applications.

3. RELATED WORK

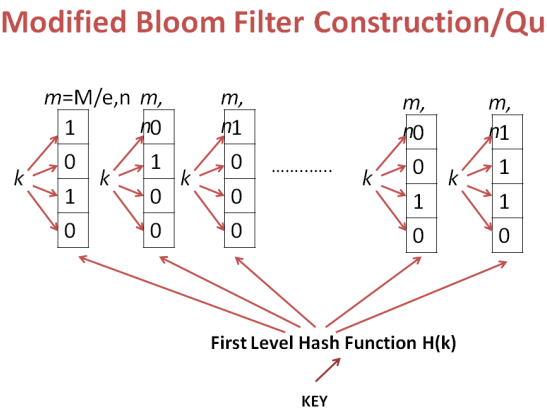
Hash Tables are used for random access to sparse data. For creating a dynamic data structure with efficient worst case access times Cuckoo Hashing was introduced in [10]. It divides the hash table into three sub tables. Thus each key has multiple possible locations. If the location for a key is occupied, it evicts the key and places itself like the Cuckoo bird in nature. The evicted key then occupies one of its alternate locations by eliminating any key in that place. This process continues until all keys find a place or until some maximum number of iterations is reached after which the hash function is replaced and the process is restarted. This method has worst case insertion time making it unsuitable for real time applications. As several moves may need to be performed this complicates the implementation on hardware. This problem is overcome in [7] that proposes a conservative 'one move only' approach and the 'second chance scheme'.

Bloom filter was proposed by Burton Howard Bloom in [3] as a probabilistic space efficient data structure. Several variants to the original Bloom filter exist that exploit tradeoffs between space efficiency, access time and error probability rate to cater to specific applications. When keys are continuously input as a stream, it becomes essential to delete the older keys to make room for the new ones. This is done with Counting Bloom filters that store an array of bits corresponding to each position in the Bloom filter table. Blocked Bloom filters divide the original filter into sub filters that fit exactly into a cache line [12]. A key first determines which blocked filter it belongs to and gets the block into cache and then performs the k lookups. Spectral Bloom filters allow querying if the frequency of a key is below a threshold or not [5]. This is useful in applications where there is a stream of incoming keys and the threshold for comparison is known only dynamically. Attenuated Bloom filters consist of an array of Bloom filters where the i^{th} Bloom filter stores the shortest path information up to i nodes [13]. Choose Bloom filters return true when a maximum of p bits out of the k bits are 1 [11]. This filter is useful for larger values of k. The filter requires a preprocessing step where a given filter bit is set to one only if a large number of keys map to it. Compressed Bloom filters are used to transmit original Bloom filters over a network. The penalty is paid in terms of the time spent on compression/decompression [8]. Scalable Bloom Filters allow the filter to adapt dynamically to the number of keys [2]. [14]Presents a simplistic implementation of Bloom filters on the GPU. This project aims to present a generic Bloom filter architecture on the GPU hardware that could potentially accelerate the table construction and query time as the number of keys becomes very large.

4. IMPLEMENTATION

There is inherent parallelism in the Bloom filter operations of construction and query that could be exploited with the GPU Architecture. As figure 3 shows, given a *key*, the computation of the k values could be done in parallel. Then the k lookups could be done simultaneously. These operations could be performed for several keys at once. The goal of the project is to provide a generic Bloom filter architecture on the GPU that could speed up current applications that use the filter. The GPU implementation adopts a two level Bloom Filter as done in [1] for a normal Hash Table. The first level consists of buckets. A given *key* uses a first level hash function to determine its bucket. This is performed on the host side. The approach basically partitions the linear m-bit array into smaller arrays that fit into the device shared meomry. This process is depicted in figure 4.Depending on the application, *key*s may be of variable size. Hence transferring *key*s to the device would incur significant variable overhead. Each block in the GPU is assigned one bucket in the later stage. Hence the number of buckets allowed is constrained by the GPU hardware. For the threads, there could be two possibilities. Assigning a thread to every *key*,

Figure 4: Filter Construction/Query process using the first level hash function. The original filter is divided into e filters, each having m bits and holding n keys.



in which case each thread would perform the k computations and look ups sequentially. The other case would be to assign each *key* with k threads to parallelize the computations. Although the latter could speed up the filter operations, it would limit the number of *key*s that could be processed by one block in a single launch. After filter construction,each thread writes a part of the filter from shared memory to global memory. we need this step as data is not retained in shared memory between successive kernel launches. However, the data is stored in global memory as long as we do not call *free*(). The consecutive Bloom Filters are merged using the bitwise-OR operation.

4.1 Choice of Hash Functions

Mitzenmacher et al have proved that using just two random hash functions and generating a linear combination of their outputs works similar to the case of using k independent hash functions. Hence we use only two hash functions h_1 and h_2 for a given *key*. The k values are calculated are follows:

$h_i(k) = h_1(key) + i * h_2(key)$ where $i = 0$ to k - 1

The implementation benefit of this method comes from the fact that only two unsigned integers $h_1(key)$ and $h_2(key)$ need to be transferred per *key* from host to device instead of the whole *key* which may be of variable length. The $h_1(key)$ and $h_2(key)$ values are stored in the device constant memory. Constant memory is chosen because there is both temporal and spatial locality in accessing $h_1(key)$ and $h_2(key)$ values by successive threads. As constant memory is cached, the access time is a single cycle. Since $h_1(key)$, $h_2(key)$ values are not reused among threads, storing them in shared memory does not yield any benefit. Allocating the filter to shared memory on the other hand, would speed up the k lookups.

4.2 Choice of First Level Hash Function

The quality of the first level hash function plays a crucial role in the total execution time. If a particular bucket overflows, the hash fucntion is discarded and the whole process has to be restarted with a different hash function. The maximum number of keys a bucket can support is given by $\frac{m*ln2}{k}$. One obvious method is to not try to fill the buckets to capacity. A load factor of 80 to 90% could be used. Writing a Hash Function is an altogether different Math problem and this project did not delve into the details of it. Instead experiments were conducted on available Hash Functions to find their convergence times for different loads. The inputs to these functions were randomly generated strings. The BKDRHash Function gives the best performance among others and is hence used as the first level hash function. The results are as shown in table 1. A '++' in table 1 indicates that convergence takes a long time and is of not much practical use.

Table 1: Convergence time of some Hash Functions measured as the number of iterations

	80%	85%	90%	95%
RSHash	0	1	6	++
JSHash	0	0	20	++
PJWHash	++	++	++	++
ELFHash	++	++	++	++
BKDRHash	0	0	1	++
SDBMHash	0	2	14	++
DJBHash	0	0	3	++
DEKHash	0	0	18	++
BPHash	++	++	++	++
FNVHash	0	1	5	++
APHash	0	26	++	++

4.3 Power of 2 Approach

An alternative to the first level hash function approach is the Power of 2 algorithm. It is a simple extension that uses two first level hash functions $g_1()$ and $g_2()$. Given a *key*, we evaluate both $g_1(key)$ and $g_2(key)$. This gives us two choices for the bucket and the *key* is inserted into the bucket that is comparatively less loaded. Thus we need to store a variable for each bucket, that determines the number of keys that have been hashed into a given bucket until that point of time in filter construction. This approach drastically reduces the convergence time and yields better uniform distribution of keys into buckets. Figure 5 shows the filter operation implementing the Power of 2 algorithm. Although this seems a good method, querying a key now becomes non-trivial. For querying, we need to check the filters associated with both the buckets and thus we get two outputs of the query, one from each bucket. For both filters returning positively or negatively there is no problem. But in the case where the two outputs contradict, we have no choice but to take the final output as true. This increases the *fpr*. We cannot choose the output as false because this gives rise to false negatives which is not desired. This step represents the trade-off between filter construction time and the false positive rate for our GPU algorithm.

4.4 Proposed GPU Algorithm

The algorithm is depicted in figure 6 and described below.

1. Allocate memory *inputkeys*[] for storing the two unsigned integers $h_1(key)$ and $h_2(key)$ for each key. The

Figure 5: Bloom Filter Construction/Query process.

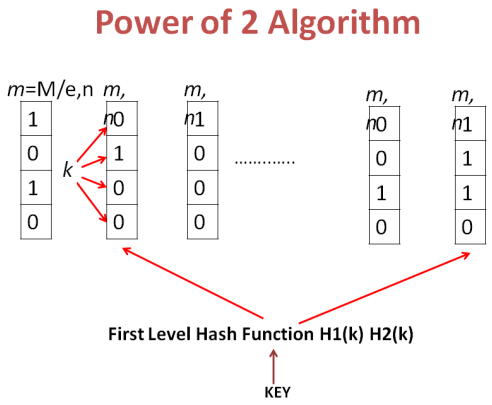


Figure 6: The GPU Algorithm.

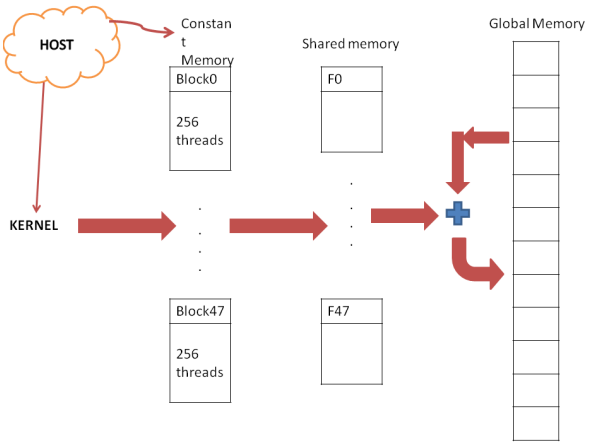


Table 2: Query: Using Power of 2 approach

$Filter_1$	$Filter_2$	Output
false	false	false
false	true	**TRUE**
true	false	**TRUE**
true	true	true

output of the first level hash functions determines the address in memory *inputkeys*[] where the values are to be stored. Array *offset*[b_k] stores the current pointer for the bucket b_k. The value *offset*[b_k] also indicates the load factor of each bucket.

2. This is referred to as the pre-processing step.

   For each key,

   - Determine the bucket b_k it belongs to using the Power of 2 algorithm. If any of the buckets overflow, we restart the whole process again by choosing a different hash function.
   - Compute $h_1(key)$ and $h_2(key)$ for the *key* using the two random hash functions.
   - Determine the offset into *inputkeys*[]. The offset is given as 512*b_k + *offset*[b_k].
   - Increment *offset*[b_k] by 2. (as $h_1(key)$ and $h_2(key)$ occupy two bytes)

3. The array *inputkeys*[] is copied into the device constant memory.

4. Launch the kernel.

5. Repeat steps 1 to 4 for each batch of keys.

6. After a fixed number of batches, transfer the filter from device global memory to host. The allowed number of batches is constrained by the device resources.

4.5 Kernel Algorithm

For each thread:

1. Read $h_1(key)$ and $h_2(key)$ from constant memory. (Using variables *threadIdx.x* and *blockIdx.x* a thread can determine the key it has to work on.)
2. Perform k computations: $h_1(key)$ + i*$h_2(key)$
3. Set k locations given by the above values in shared memory.
4. Transfer the filter from shared memory to global memory using the bitwise-OR operation.

4.6 CPU Code

The CPU code implements a *class* BLOOM that has m (number of bits), n (maximum number of keys allowed), k (number of hash functions) and *filter*[] (the m-bit array) as its members. The data type for *filter*[] is *unsigned char*. When an object of the *class* BLOOM is created, each element of the array *filter*[] is intialized to zero. The functions *insert_key*() and *query_key*() insert and qeury keys respectively. Both these functions internally call the function *getHashValue*(). This function calculates the k values given by $h_1(key)$ + i*$h_2(key)$ and returns them in an array *hash_values*[]. For filter construction and query we are concerned with only one bit at a time in the array *filter*[]. However C++ allows access to one byte at a time. Thus the following logic is used to set/query for a single bit position within the array:

SETBIT(*filter*, *pos*) => (*filter*[*pos*/8] |= (1≪ (*pos*%8)))

GETBIT(*filter*, *pos*) => (*filter*[*pos*/8] & (1≪ (*pos*%8)))

Pos is the output of the k computations that return the position in the m-bit array. SETBIT and GETBIT are implemented as macros. GETBIT returns either *true* or *false* depending on whether the bit is 1 or 0.

5. EXPERIMENTAL RESULTS

The challenege is to extract maximum performance out of the GPU for the above proposed algorithm. Thus, the CUDA Occupancy calculator, a profiling tool provided by Nvidia was used to find the optimum kernel configuration.

5.1 CUDA Occupancy Calculator

This is a profiling tool that is used to determine the usage / occupancy efficiency for the GPU hardware [9]. The goal is to keep the multiprocessors busy all the time to get maximum performance. The device used for this project is the GeForce 9800 GTX+. The properties of the device are summarized in table 3.

Table 3: GeForce 9800 GTX+

#SM	16
#Processors per SM	8
Max. #blocks per SM	8
Max. #blocks	128
Max. #threads per block	512
Max. #threads per SM	768
Shared Memory per SM	16KB
Constant Memory	64KB
Global Memory	nearly 512MB
Compute Capability	1.1

The parameters were chosen as follows for 100% occupancy:

- 3 blocks per multiprocessor
- Each block runs 256 threads
- Each block handles an independent Bloom Filter
- So in total there are 3 filters per multiprocessor, 48 filters for a kernel launch. Thus there are 48 buckets as well.

5.2 Individual Filter Parameters

For each filter, the parameters are as follows:

m = 5KB * 8 = 40960 bits

Select m = 40949 bits (prime number)

Assume fpr = 1% = 0.01

k = 6.644 Select k = 7

n = 4054

Thus the total number of keys supported in on the device = 4054 * 3 *16 = 194592. This numebr is further reduced by the load factor chosen for the buckets (90%). If more number of keys need to be added then the entire Bloom filter is shifted out of the device to host and the insertion process begins again as per the algorithm. Each thread block is responsible for inserting n keys in batches of 256.

A random string generator function that outputs 41-byte keys was used for this experiment. The entire program is also written on the host side to compare execution times. For the given GPU device, each batch processes 256*48 = 12288 keys. Thus in each batch we transfer 12288*2 = 24576 bytes of preprocessed data (unsigned integers $h_1(key)$ and $h_2(key)$) to device constant memory. After this the kernel is launched, filter constructed in shared memory is merged with the previously computed filter in global memory. This process is repeated for the maximum possible number of keys. (4054*48*0.9 = 175132). The results are summarized in the table 4.

Figure 7: Computation time for preprocessing step, CPU code and their absolute difference.

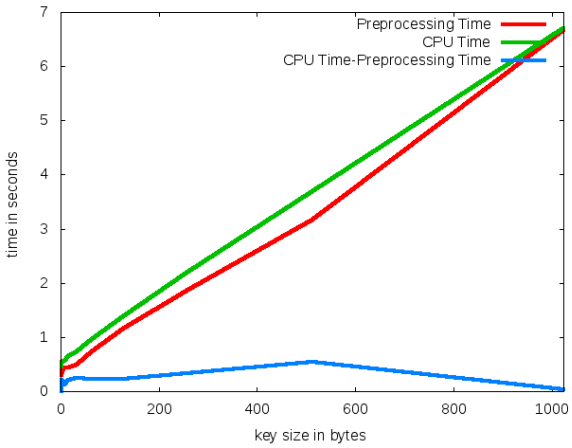


Table 4: Results for Filter Construction

CPU Time	0.24 seconds
GPU Algorithm = 0.122 seconds	
Preprocessing time in host	0.11 seconds
Constant Memory Transfer	0.4 milliseconds
Kernel Time	1.6 milliseconds
Transfer filter back to host	10 milliseconds
Total Time	0.122 seconds
Speed Up = 1.96	
If preprocessing ignored, Speed Up = 20	

The preprocessing time and the CPU time is a function of the key size. Figure 7 shows a plot of the preprocessing time on the host side, the entire filter construction on the host side and their absolute difference as a function of the key size. The larger the difference, the higher the speed up. A key size of around 512 yields the maximum speed up. The other components of execution time, filter transfer to host, kernel time and constant memory transfer time, are independent of the key size. They depend on the number of keys alone.

6. CONCLUSION

The results of the experiment in table 4 evoke some interesting discussion. If one excludes the preprocessing step, the speed up is significant and the use of the GPU is justified. The preprocessing step however is an integral part of the algorithm to port the Bloom Filter on to the GPU. Thus we need to come up with better ways to preprocess a given set of keys. One more noteable observation is that the actual filter construction time and communication latency between GPU and CPU is independent of the key size. As only two unsigned integers are involved in computations for all key sizes.

7. FUTURE SCOPE OF THE PROJECT

The main drawback of the algorithm proposed in this paper is that it supports filter construction or query but not both simultaneously. Applications like data deduplication demand online construction and query. The pre-processing step of hashing keys into buckets is the clear bottle-neck. Since this step is done on the host one could try out different aggressive optimizations like using the '-O3' compiling option or OpenMP. An ambitious approach would be to offload this computation to another device. Then the preprocessed keys could be transferred from one device to another without needing host CPU cycles. This would require transferring the keys to the device. Distributing work among devices would need a scalable algorithm that could handle a larger number of keys efficiently. This approach would succeed if the speed up gained by offloading the computation overpowers the communication overhead in transferring keys to device and preprocessed keys from one device to another. This is open to further research and experimentation.

8. REFERENCES

[1] D. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. Owens, and N. Amenta. Real-time parallel hashing on the GPU. *ACM Transactions on Graphics (TOG)*, 28(5):1–9, 2009.

[2] P. Almeida, C. Baquero, N. Preguiça, and D. Hutchison. Scalable bloom filters. *Information Processing Letters*, 101(6):255–261, 2007.

[3] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[4] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

[5] S. Cohen and Y. Matias. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, page 252. ACM, 2003.

[6] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):293, 2000.

[7] A. Kirsch and M. Mitzenmacher. The power of one move: Hashing schemes for hardware. In *IEEE Infocom*, pages 565–573. Citeseer, 2008.

[8] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking (TON)*, 10(5):604–612, 2002.

[9] Nvidia. CUDA Occupancy Calculator.

[10] R. Pagh, N. M. Bldg, D.-A. C, and F. F. Rodler. Cuckoo hashing, 2001.

[11] E. Porat. Google Tech Talk on Bloom Filters, nov 2007.

[12] F. Putze, P. Sanders, and J. Singler. Cache-, hash-and space-efficient bloom filters. *Experimental Algorithms*, pages 108–121.

[13] S. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *IEEE INFOCOM*, volume 3, pages 1248–1257. Citeseer, 2002.

[14] H. Shi, B. Schmidt, W. Liu, and W. M "uller-Wittig. A Parallel Algorithm for Error Correction in High-Throughput Short-Read Data on CUDA-Enabled Graphics Hardware. *Journal of Computational Biology*, 17(4):603–615, 2010.

[15] P. Valduriez and G. Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM Transactions on Database Systems (TODS)*, 9(1):133–161, 1984.