GPU-based PostgreSQL Extensions for Scalable High-throughput Pattern Matching

Grant Scott
Center for Geospatial Intelligence
University of Missouri
Columbia, Missouri, USA
Email: GrantScott@missouri.edu

Matthew England,
Kevin Melkowski
and Zachary Fields
Department of Computer Science
University of Missouri
Columbia, Missouri, USA

Derek T. Anderson
Dept. of Electrical and Computer Engineering
Mississippi State University
Mississippi State, MS, USA
Email: anderson@ece.msstate.edu

***Abstract*—Numerous fields require large-scale pattern matching to achieve a variety of computational goals. Herein, we present novel graphics processing unit (GPU) extensions that facilitate high-throughput pattern matching in a PostgreSQL database. We have developed an extension framework to perform data block processing of large pattern data sets, using a stream processing design that results in global k-nearest neighbor matches. This framework was specifically designed to support pattern matching on GPU from within the database environment. This approach avoids the necessity of storing an entire data set onto GPU hardware, which facilitates significant scale-up of pattern databases. This provides enormous potential to incorporate or exploit auxiliary (meta)data as part of the pattern matching process; as well as pipelining the results into traditional relational algebra expressions. By pipelining pattern matching results into a relational expression, the power of the database can be leveraged to build result sets based on various parameterized correlations between the query pattern(s) and the results. In this preliminary work, we have integrated GPU-based high-throughput *p-norm* metric functions into the database server. This allows one to design heterogeneous data processing techniques that combine large-scale content-based image retrieval (CBIR) with traditional data processing capabilities of the database such as relational, spatial, or text search. We present timing characteristics for various pattern sizes and metric combinations, as well as address the balancing of database and GPU parameterization. Our feature vector datasets range from 18 to 85 GB in database table storage size, reaching 100 million 128 dimensional vectors. We are able to efficiently execute global top k searches from within the database.**

Keywords*—Pattern matching, heterogeneous data, graphics processing unit (GPU), PostgreSQL, high-throughput computing (HTC), high-performance computing (HPC)

I. Introduction

Numerous fields require large-scale pattern matching to achieve a variety of computational goals. The specifics of matching patterns are highly varied, ranging from indexing to pruning to brute-force matching. Scalability is often a concern as content and pattern databases are growing increasingly larger, into the terabyte-scale. This growth is driven by many factors, such as increased database members (e.g., quantity of images) as well as increasing numbers of measurements and measurement encoding sizes.

High-performance solutions to large-scale pattern matching problems often utilize distributed systems or computing clusters and network programming. These are typically composite hardware-software solutions, which utilize a combination of distributed hardware, networking technologies, and software to achieve robust, scalable pattern matching. Map-Reduce architectures (e.g., Hadoop) are a common approach to achieve scalable pattern matching. In both [1] and [2], Hadoop facilitates searching large time series sets. String patterns and regular expressions are matched using Hadoop in [3]. However, distributed approaches such as these incur increased overhead from networking and management of the reductions. Additionally, these approaches often rely on uniquely customized indexing or partitioning data structures to facilitate scalable pattern matching. However, relying on customized data structures and distributed platforms presents limitations when considering integration with other data or metadata. For instance, it is difficult to utilize image metadata of any nature to reduce the content-based image retrieval (CBIR) search space in such designs.

The aforementioned approaches are often necessary for pattern matching against large data sets to overcome the processor, storage, and/or memory limitations of a single machine. Often the true limitation is a scalability requirement that simply cannot be achieved by a particular solution without significantly increasing the hardware resources. In contrast to customized distributed architectures or data structures, open source database management systems, such as PostgreSQL, already support large datasets and use of heterogeneous restriction criteria, e.g., relational attribute filtering with spatial constraints. Additionally, PostgreSQL is naturally scalable as both a data storage and processing framework. However, database management systems are not traditionally equipped for high-throughput numerical computations, such as needed for large-scale pattern matching.

General purpose programming on graphics processing units (GPGPU) represents a promising composite hardware-software solution for pattern matching problems. GPGPU programming has been leveraged against various problems related to pattern matching and retrieval, e.g., [4]–[8]. Various research efforts, such as [9]–[11], to list a few, have utilized GPU hardware to perform descriptor matching, a common task in CBIR applications. GPGPU techniques have been used in closely related domains such as classification [12] and string matching [13]. Unfortunately, one of the common drawbacks of many existing GPU-driven pattern matching approaches is the need

```
 1: procedure PgPatternStreamMatch($Q$,$D$,$k$)
 2:                                          ▷ $Q$ query pattern
 3:                                          ▷ $D$ pattern data set
 4:                                          ▷ $k$ desired result size
 5:     Build SPI cursor to pull pattern stream
 6:     repeat
 7:         Fetch cursor block of size $X$
 8:         Score ← pattern match function
 9:         for Each row of block do
10:             Load ID fields into structure
11:             Load match-score into structure
12:             Push structure into $k$-sieve
13:         end for
14:     until End of Cursor
15:     Unpack $k$-sieve into Tuple Store
16: end procedure
```

Fig. 1. PostgreSQL pattern stream matching algorithm that leverages the server programming interface (SPI). This algorithm returns a table expression for subsequent database operations. The cursor block size, X, is configured to optimize the performance of the database.

to materialize the entire database onto the GPU.

In [14], the authors highlight the need to augment GPU memory with fast persistent memory, such as modern solid state drives (SSD). This is increasingly necessary as the growth in size of the pattern databases is out pacing the growth of GPU memory. Similarly, the authors of [15] saw the importance of being able to process massive amounts of data on the GPU. They investigate kernel splicing and scheduling to illustrate the importance of using smaller chunks of data and calling a GPU kernel several times. Using this approach, the authors achieved a 7-23% increase over copying all of the data at once then computing with a single kernel invocation. Similar experience has motivated our approach herein, whereby we address scalability of pattern matching data sets as well as maximization of throughput.

In this paper, we present novel GPU-based stream processing extensions for the PostgreSQL database which facilitate high-throughput pattern matching to generate global k-nearest neighbor matches. This framework is designed to support pattern matching on a GPU from within the database environment; wherein the results are a table expression, which can be utilized in subsequent relational algebra operations. Furthermore, we use the database's native storage facilities for our pattern data, which provides scalability and integration with related data. This gives us the ability to apply relational or other constraints to the data prior to high-throughput pattern matching, i.e., passing table expressions into the pattern matching framework.

The remainder of this paper is organized as follows. Section II provides an overview of our extensions for PostgreSQL database to support high-throughput pattern matching. Then, Section III provides the details of our work to exploit GPGPU within this high-throughput framework. We detail our preliminary experiments of timing based on metric, pattern, and GPU kernel parametrization characteristics in Section IV. This section also provides experiments from an initial CBIR data set. Finally, Section V offers some concluding remarks and discussion of future work and extensions.

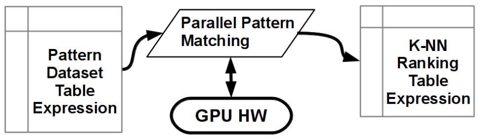

Pattern Dataset Table Expression
Parallel Pattern Matching
GPU HW
K-NN Ranking Table Expression

Fig. 2. Our pattern matching extensions utilize GPU kernels to perform massively parallel data processing to process and produce these table expressions. *Table expressions* are the fundamental input and results of database operations.

II. Pattern Stream Matching Framework for PostgreSQL

Our initial goal was to develop a general algorithm to achieve stream-style processing of pattern blocks within the PostgreSQL environment using the server programming interface (SPI). The SPI facilitates the extension of the PostgreSQL server using very low-level access to the database internals. Figure 1 provides the algorithm used to process a stream of patterns. This algorithm is built into a shared object library and linked into the database through a bridging function using standard C-language PostgreSQL extension functions. Each pattern in the stream is compared against the query pattern, the resulting distance is then combined with other fields of the source tuple to form a potential row of the result table.

A key capability of our design is that we generate the global *k-nearest neighbors* while processing the pattern stream. This is accomplished by implementing a result collection structure, which we refer to as a *k-sieve*. Whereas, a physical sieve processes a stream of sediment, retaining only the largest members; our k-sieve processes the stream of patterns, retaining only the k best pattern matches. The k-sieve is a max-heap based data structure designed to collect the k best matches from a set in near-linear time complexity $O(k + n \log(k))$, where n is the number of patterns compared. We use the first k elements of a set to fill the k-sieve, then each element thereafter is compared with the top of the heap. The ability to reject such candidates with a single comparison, allows us to immediately ignore all values greater than the largest value of our set and to accept elements from any number of input groups; atomic or otherwise.

The k-sieve is an extremely effective tool for collecting a subset of the best matches from one or many sets of data. This property of the k-sieve works efficiently with the ability to fetch blocks of data from PostgreSQL via cursors using SPI. Each *pattern block* from the cursor can be processed individually through the k-sieve without disrupting the integrity of the current contents of the sieve. Ultimately, the final set of k items collected by the k-sieve will be unpacked into a sorted array, which forms our resulting *table expression*.

Our set of patterns are stored in a simple database structure, as modeled in Figure 3. In this model each object is associated with one or more descriptor patterns. These descriptors are stored natively in the database as an array column. This array storage simplifies the pattern stream processing. This design allows the reduction of the pattern stream using standard *join* and *restrictions* operations which are native to the database. These operations may be used to perform attribute-based filtering via the *Object Metadata* table shown, as well as using any number of additional tables that can be added to

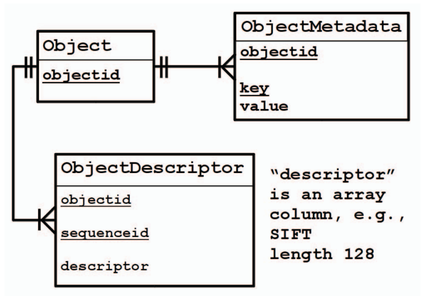
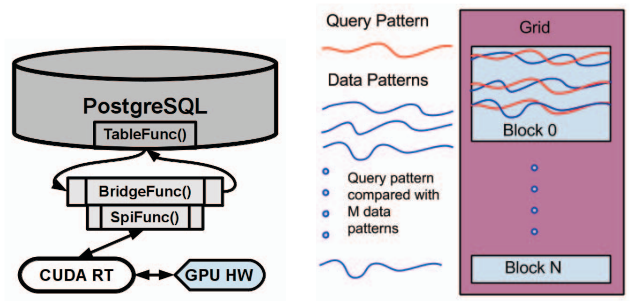


Fig. 3. The very simple pattern database design, where the *Object* is an arbitrary concept which has associated metadata and one or more pattern descriptors. The *descriptor* is a numerical array (i.e., feature vector) column.



(a) (b)

Fig. 4. Integration of GPU with DBMS: (a) A *table expression* returning function is defined within the database to reference a C-language function of a shared object. The initial entry point into the shared object is a bridge function which invokes the SPI function, which in-turn, utilizes the GPU through CUDA kernels. (b) The SPI function performs massively parallel dissimilarity measurements against cursor blocks of size M, dividing the work into GPU optimized blocks of patterns.

this design and contain more advanced data. For instance, if *Objects* have associated spatial attributes, PostGIS extensions and spatial indexing could be utilized to filter the pattern stream. In a similar fashion, other existing extensions could be leveraged such as text search capabilities or content based retrieval techniques.

Our technique allows the design of heterogeneous data processing techniques, both prior to and after pattern matching. As a basic example of input table expressions, consider *table views*; any number of related data (attributes, spatial, joins, etc.) could be used to produce the input table expression. Since the output is also a table expression, it naturally fits into the standard relational algebra model (restrictions, joins, aggregations, etc). This concept is depicted in Fig. 2, where the results of pattern matching can be utilized with the traditional data processing capabilities of the database such as relational, spatial, or text search.

III. GPGPU PATTERN MATCHING WITHIN POSTGRESQL

Now that we have outlined an algorithm to extend the PostgreSQL database for high-throughput pattern processing, we shift focus to incorporating high performance computing (HPC) techniques into this extension. The goal is to achieve the pattern matching speedup that is capable through GPGPU techniques when processing the patterns from within the

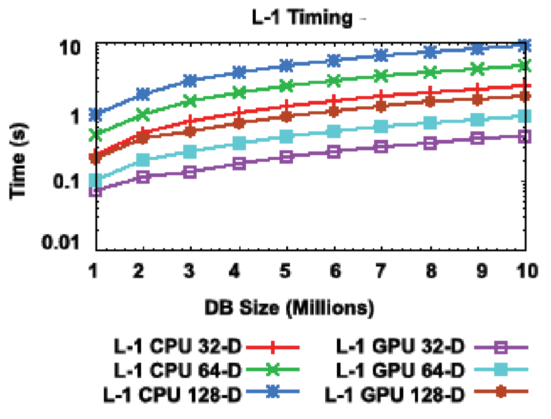
```
1:  procedure PgGpuPatternStreamMatch($Q$,$D$,$k$)
2:                                        ▷ $Q$ query pattern
3:                                        ▷ $D$ pattern data set
4:                                        ▷ $k$ desired result size
5:      Build SPI cursor to pull descriptor stream
6:      repeat
7:          Fetch cursor block of size $M$
8:          for Each tuple from cursor block do
9:              Load ID fields into temp ID block
10:             Load pattern row into temp block
11:         end for
12:         match-scores ← GPU kernel on temp pattern block
13:         for Each row of block do
14:             Load ID fields into structure
15:             Load match-score into structure
16:             Push structure into $K$-sieve
17:         end for
18:     until End of Cursor
19:     Unpack $K$-sieve into Tuple Store
20: end procedure
```

Fig. 5. GPGPU enhanced SPI pattern stream matching algorithm. Returns a table expression for subsequent database operations. The cursor block size, X, is configured to optimize the performance of the database.

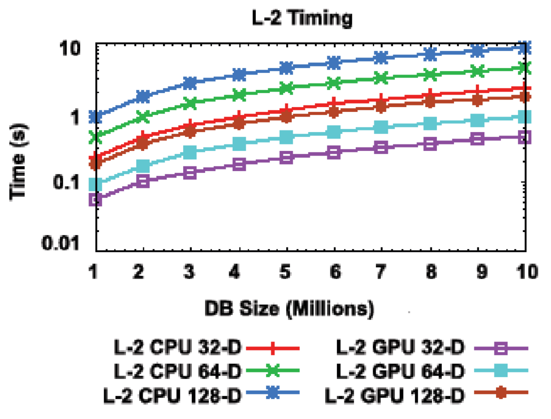
database environment. Matching millions of pattern feature vectors against a query pattern using a metric function is a natural fit for processing on the GPU. The GPU has hundreds of cores which allow it to simultaneously measure a query pattern against millions of feature vectors. In comparison to the CPU, which has to loop through each feature vector and compare against the query pattern, the GPU only needs a small amount of time and resources to match patterns. Figure 6 shows the orders of magnitude acceleration of pattern matching achieved by GPU over CPU.

We use CUDA [16] as our GPGPU framework for developing massively parallel pattern matching database extension modules. In the context of CUDA, various logical parallelization strategies can be applied to partition the data processing by organizing a kernel grid. A *kernel* function is the unit of work that is to be computed in parallel on the GPU hardware. In this respect, the solution space is mapped into a logical *grid* which is composed of processing *blocks*. A single block is composed of a set of *threads*, where each thread will be invoked with the kernel function to process one datum. Modern GPU will typically be processing multiple blocks across the available computing units simultaneously. Thus, we use thousands of blocks processing hundreds of patterns each, to effectively parallelize the matching of millions of patterns within a grid.

To integrate a GPGPU solution into our pattern stream matching, we extend the algorithm of Figure 1 to incorporate generation of a pattern block suitable for GPU kernel functions. Figure 4(b) provides a conceptual depiction of how we map a particular pattern block (see Fig. 5, step 12) onto the CUDA grid. We define the *threads per block* (TPB), and therefore the number of blocks in the grid. In our current implementation, the TPB defines the number of patterns per GPU logical block. Note, the GPU logical block (32 TPB) is much smaller than the SPI cursor pattern block (1.5 million), where the SPI pattern block is the data underlying the CUDA grid. We have empirically evaluated various TPB to find the optimal



(a)



(b)

Fig. 6. GPU vector pattern matching timings using (a) Manhattan and (b) Euclidean respectively.

performance curve for our GPU hardware (see Fig. 7). Each large pattern block is pushed to the GPU device and then the kernel is launched, after which the results (i.e., metric distances) are copied back. The result measures are merged with the other row data to form a *result tuple* which is passed into the k-sieve as detailed in Section II. These result tuples are critical to integrate the pattern matching within a database management system with heterogeneous data processing, such as relational or spatial operations. Figure 5 illustrates this algorithm, whereby the SPI cursors are used to construct pattern blocks that are passed to a GPU for pattern matching.

One of the key novelties of our approach is that we avoid the necessity of storing the entire database of patterns on the GPU hardware. Figure 4(a) illustrates the organization of the database, the dynamically linked shared object, and the CUDA runtime. As discussed in Sect. II, the result of the pattern matching process is a table expression within the database environment.

IV. Experiments

We begin our experimentation by verifying that incorporation of GPU powered HPC techniques is a sensible approach. As discussed in Section I, numerous research into GPGPU

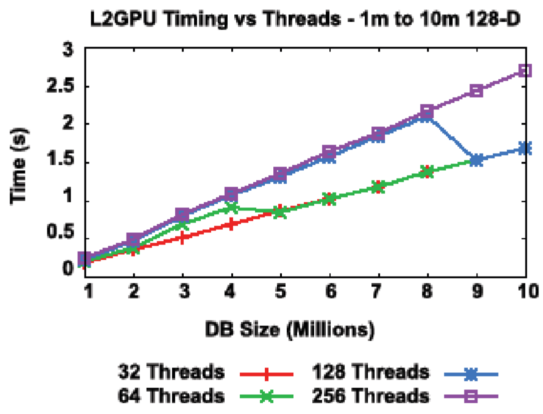


Fig. 7. GPU vector pattern matching timings, using 128-D patterns over increasing database size, in light of differing CUDA threads per block.

instantiates an entire data set on the GPU. However, this is obviously not a practical, cost effective solution for truly large-scale pattern data sets. As discussed in Section III, our extension relies on copying data from the native database tables onto the GPU hardware and then copying the results back. We therefore conducted timing experiments using multiple metric functions, varied feature vector dimensionality, and increasing database sizes.

A. Baseline GPU Timing

Figure 6 shows a timing trend comparison of CPU- and GPU-based *p-norm* metric functions computed on data sets of increasing size and different vector lengths. It is clear that the timing grows at a much faster rate on the CPU versus the GPU. In each plot, the data set size grows from 1 million to 10 million vectors. We show timing of 32-D, 64-D, and 128-D feature vectors for comparison purposes, all using 32 threads per block for GPU kernel functions. Figure 6(a) shows the trend of computing *Manhattan distance* and (b) shows *Euclidean*. Figure 7 provides performance relative to another key parameter of GPU kernel functions, namely the TPB. In this plot, the 128-D patterns are used and the database size increases from 1 million to 10 million. We have included GPU host-device data and result transfer times in these results, as this is a critical component of processing the stream of database pattern blocks. We can see that even considering the data transfer onto and off the GPU hardware, the GPU based matching is nearly an order of magnitude faster than the CPU (note the y-axis is log-scale).

B. GPU-enabled PostgreSQL Timing

It should be noted that the timing measurements of matching patterns from within the database uses data set sizes an order of magnitude larger than the Section IV-A experiments, as we want to process data sets that simply cannot be materialized onto the GPU. Furthermore, there are additional costs associated with pulling data from the database, namely reading the data from disk and unpacking from native tuples. There exists numerous database tuning techniques that can be applied to accelerate pattern data access (*table scans* in our

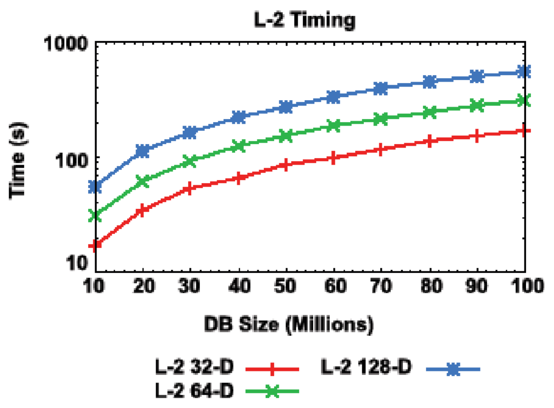


Fig. 8. GPU-enabled database pattern match comparison timing trends using the Euclidean distance metric.

TABLE I. Global k-nearest neighbor, with $k = 1000$, for the L_1 and L_2 metrics against the increasing database sizes of 10 million to 100 million.

Metric	L_1			L_2		
Pattern Length	32	64	128	32	64	128
10 million	15.10	30.19	55.07	16.54	30.31	54.48
20 million	34.37	60.98	109.08	33.54	60.87	111.93
30 million	48.90	91.00	167.81	51.75	90.71	162.26
40 million	64.45	121.43	225.90	64.86	120.64	217.48
50 million	86.16	151.83	272.62	85.56	151.24	269.67
60 million	100.11	182.31	326.16	98.14	181.97	328.86
70 million	118.95	213.50	379.46	115.46	211.91	383.31
80 million	133.69	242.84	439.82	135.19	243.34	437.58
90 million	149.95	274.12	506.19	150.30	274.27	491.29
100 million	166.48	304.72	549.83	165.40	303.49	547.11
Database Size	**Time (s)**					

experimental case), however these discussions are beyond the scope of the present paper and are left to future work.

We conducted timing experiments using 32-, 64-, and 128-dimensional floating point feature vectors. Our timing in the database start with pattern data sets of size 10 million, and increases to 100 million in increments of 10 million. The GPU hardware used was a Nvidia Tesla C2075 GPU co-processir, with the kernels developed using CUDA Toolkit 5. Our database management system was PostgreSQL version 9.2.4 on *x86_64* Linux. The database storage tablespaces are on a RAID 5 file system. The database table sizes for each X-dimensional feature set are 18GB, 36GB, and 54GB, respectively; which far exceeds any current GPU hardware device storage capacity. The pattern block size used by the database extensions is a factor of the memory parameters configuration of PostgreSQL and the descriptor size. For our timing experiments, we fixed the pattern block at 1.5 million patterns to accommodate all pattern sizes and provide consistency during testing of the other parameters of the system. PostgreSQL is configured with modest memory limits, i.e., 2GB shared buffers, 24MB working memory, and other settings as defaults. We use a CUDA block size of 32 threads per block in experiments that measure variable database and descriptor sizes. During these experiments, we were able to compute the global *k-nearest neighbors*, with $k = 1000$ in an average of 166.5 seconds against a database of size 100 million for the 32-D patterns. Table I provides the complete timing statistics from our testing of pattern database sizes from 10 million upto 100 million. As expected, since the database is significantly larger than the desired number of top pattern match results, k, the global k-nearest neighbors are found in linear time.

C. Image Matching with SQL

Additionally, we have performed preliminary experiments using image descriptors. We have begun building an image retrieval database from an image data set provided by [17]. The images were processed using publicly available Hessian-Affine region detector and SIFT key-point descriptors, from [18]. To process the data set, we extracted the SIFT feature descriptors for key-points found through the Hessian-Affine region detector. Each image produces an output data file, with one key-point per line. We parsed the data files, creating *Object* records, then loaded the *Object Descriptor* table (see Fig. 3). Once the data is loaded into the *Object Descriptor* table, it is immediately searchable using our GPU-enabled pattern stream matching extensions. As expected, the number of key-points varies significantly across images, with 98103 images loaded and the number of key-points per image ranging from 1 to 8686. The resulting database has over 85 million SIFT key-points. The *Object Descriptor* table is approximately 50 GB.

If a query image has m keypoints, we can define a relational algebra expression to rank the images within the database as

$$\text{id}\ g_{\text{count}(*)}\Big(\bigcup_{i\in 1..m} match(kp_i, data, k)\Big); \qquad (1)$$

where kp_i a keypoint's feature vector, *data* is the pattern data set table expression, k is the number of matches per keypoint to pull into the result, and the result of *match* is a table expression. *ObjectId* is the unique image identifier. Using SQL, this could be expressed as the following:

```
SELECT matches.id,count(*) as score
FROM  (
   SELECT id,score FROM match(kp_1,data,k)
   UNION
   SELECT id,score FROM match(kp_2,data,k)
   UNION
    ...
   SELECT id,score FROM match(kp_m,data,k)
) as matches
WHERE matches.score <= kpMatchThreshold
GROUP BY matches.id
ORDER BY matches.score DESC;
```

In this query, the count, k, of nearest neighbors and the keypoint match quality threshold, *kpMatchThreshold*, are the tunable parameters. We acknowledge that this approach is not the *state of the art* use of keypoint features for CBIR (e.g., visual code books, etc.). We are simply providing an illustrative example of the integration between GPU hardware and PostgreSQL in a relevant application domain.

This novel integration of GPU hardware and the PostgreSQL database represents a significant alternative to existing architectures which support large-scale pattern matching.

These techniques will allow researchers to fully leverage heterogeneous data; fusing the results of pattern matching with more traditional data processing algorithms. We see significant opportunities to exploit this research in various pattern recognition domains such as content-based retrieval, biometrics, image and video analysis, and other signal processing.

V. Conclusion

In this article, we integrated GPU-based high-throughput *p-norm* metric functions into a PostgreSQL database server. We introduced our technique to extend the PostgreSQL backend with GPU-enabled pattern matching. This facilitates large-scale pattern matching using GPU hardware because the data set size is not limited to what can stored on the GPU. Additionally, this allows one to design heterogeneous data processing techniques that combine large-scale pattern matching with traditional data processing capabilities of the database such as relational, spatial, or text search. This has enormous potential for future work in a variety of fields including, CBIR, biometrics, and large data set analytics, just to list a few.

Our extensions provide pattern matching results as a table expression within the database environment, allowing the resulting tuples to factor into traditional relational algebra expressions. This inherently leverages the the power of the relational database engines to correlate information across numerous tables using the traditional SQL of the database. We have implemented *p-norm* metric functions, for $p \in \{1, 2\}$, i.e., Manhattan and Euclidean.

In future work, we will conduct additional timing and database tuning experiments. We will determine the optimal memory settings for the database, and how these settings affect the parameters used to build the GPU extensions for various pattern sizes. We intend to develop these techniques as open source extensions to PostgreSQL and generate comprehensive documentation of how to incorporate extensions into various application domains that require pattern matching. We will begin to explore more specialize pattern metrics and measures in the future, such as Mahalanobis distance and others. Additionally, we will investigate alternative kernel and block organizations such as using multiple threads to compute a single pattern matching score following parallelized reduction methodologies.

Furthermore, we will utilize these extensions to create scalable heterogeneous content-based retrieval systems which will use image content, image metadata, and other information available for query construction. We will develop the PostgreSQL integrations to support visual codebook approaches as well, i.e., as the pre-matching table expressions. It is also necessary to develop GPU coordination strategies, as databases are inherently multi-process concurrent systems. Our goal is to provide a robust set of open source extensions to support large-scale, high-throughput pattering matching within PostgreSQL to empower researchers to explore novel exploitations of patterns matching with other data.

References

[1] A. Berard and G. Hebrail, "Searching time series with hadoop in an electric power company," in *Proceedings of the 2Nd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, ser. BigMine '13. New York, NY, USA: ACM, 2013, pp. 15–22.

[2] C.-W. Lu, C.-M. Hsieh, C.-H. Chang, and C.-T. Yang, "An improvement to data service in cloud computing with content sensitive transaction analysis and adaptation," in *Computer Software and Applications Conference Workshops (COMPSACW), 2013 IEEE 37th Annual*, 2013, pp. 463–468.

[3] N. Rizvandi, J. Taheri, and A. Zomaya, "On using pattern matching algorithms in mapreduce applications," in *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*, 2011, pp. 75–80.

[4] C.-H. Lin, C.-H. Liu, and S.-C. Chang, "Accelerating regular expression matching using hierarchical parallel machines on GPU," in *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*, 2011, pp. 1–5.

[5] K. Derpanis, M. Sizintsev, K. Cannons, and R. Wildes, "Action spotting and recognition based on a spatiotemporal orientation analysis," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 35, no. 3, pp. 527–540, 2013.

[6] L. Wang, S. Chen, Y. Tang, and J. Su, "Gregex: GPU based high speed regular expression matching engine," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*, 2011, pp. 366–370.

[7] C. Beleznai, D. Schreiber, and M. Rauter, "Pedestrian detection using GPU-accelerated multiple cue computation," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, 2011, pp. 58–65.

[8] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "Parallelization and characterization of pattern matching using GPUs," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, 2011, pp. 216–225.

[9] A. Kooijman and J. Vergeest, "GPU implementation of the lft shape matching algorithm," in *Parallel Computing in Electrical Engineering (PARELEC), 2011 6th International Symposium on*, 2011, pp. 111–116.

[10] M. Rauter and D. Schreiber, "A GPU accelerated fast directional chamfer matching algorithm and a detailed comparison with a highly optimized cpu implementation," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2012 IEEE Computer Society Conference on*, 2012, pp. 68–75.

[11] D.-D. Truong, V.-T. Nguyen, A.-D. Duong, C.-S. N. Ngoc, and M.-T. Tran, "Realtime arbitrary-shaped template matching process," in *Control Automation Robotics Vision (ICARCV), 2012 12th International Conference on*, 2012, pp. 1407–1412.

[12] M. Nabiyouni and D. Aghamirzaie, "A highly parallel multi-class pattern classification on gpu," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, May 2012, pp. 148–155.

[13] N.-P. Tran, M. Lee, S. Hong, and J. Choi, "High throughput parallel implementation of aho-corasick algorithm on a gpu," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, May 2013, pp. 1807–1816.

[14] A. Cevahir and J. Torii, "Gpu-enabled high performance online visual search with high accuracy," in *Multimedia (ISM), 2012 IEEE International Symposium on*, Dec 2012, pp. 413–420.

[15] J. Zhong and B. He, "Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 99, no. PrePrints, pp. 1–11, 2013.

[16] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, June 2011.

[17] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman, "Object retrieval with large vocabularies and fast spatial matching," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2007.

[18] K. Mikolajczyk and C. Schmid, "Scale and affine invariant interest point detectors," *International Journal of Computer Vision*, vol. 60, no. 1, pp. 63–86, 2004. [Online]. Available: http://www.robots.ox.ac.uk/~vgg/research/affine/index.html