Using the Graphics Processor Unit to Realize Data Streaming Operations

Konstantinos Tsakalozos
University of Athens
Athens 15784, Greece
k.tsakalozos@di.uoa.gr

Manolis Tsangaris
University of Athens
Athens 15784, Greece
mmt@di.uoa.gr

Alex Delis
University of Athens
Athens 15784, Greece
ad@di.uoa.gr

ABSTRACT

Software development kits (SDKs) and supporting tools for Graphics Processor Units (GPUs) have matured and they now enable the implementation of complex middleware that takes advantage of the additional processing power. Working in synergy with CPUs, GPUs are suitable for executing highly parallelized tasks on streams of data. In this paper, we investigate the realization of effective operations on streams of data using GPU resources. We suggest a model for computing basic SQL-like queries that include unary/binary logical operators, membership queries as well as joins based on nested-loops. We also propose a framework that exploits the above core operations to offer a generalized computing environment for managing streams of data. Through experimentation with the *NVIDIA CUDA* SDK, we show sizable benefits in obtaining shorter response times not only for simple operations but also for more complex queries on streams.

Keywords

Data Streams Operations using GPUs, Graphics Processor Unit programing, GPU Execution Model.

1. INTRODUCTION

Placing dedicated co-processors in a computing systems is not a new idea. The aim of such efforts has always been to boost system performance by offloading the work of the central processing unit (CPU). It has been traditionally the case that co-processors tend very specific tasks that otherwise would have been consuming considerable amounts of CPU cycles. The Graphics Processor Unit (GPU) is a co-processor helping in the materialization of fast visual aspects. Nowadays, the absence of such co-processors may render a system inoperable; for example disabling the DMA would with certainty result into severe performance degradation for the system as a whole.

Current GPUs feature noteworthy processing capabilities as they employ a multi-threading model whose threads can be run in truly parallel fashion. The internal design of such co-processor is aligned with the Single Instruction Multiple Data (SIMD) model used to apply a single transformation (i.e., numerical operation) on a number of points (located in the 3 - dimensional space) simultaneously. Hence, architectures and computing systems designed for stream processing [1, 3] are set to benefit from the exploitation of now available and underutilized GPUs.

Implementing applications that can effectively harvest the GUPs resources was made feasible through specialized languages such *CUDA* and others [8, 9, 2]. The public release of both SDKs and support tools by companies such as *NVIDIA* and *ATI* for programming GPUs essentially warrants new avenues for middleware to exploit existing powerful resources. In this paper, we examine the design and implementation of fundamental data stream operations, provide a framework for the deployment of such operations, and investigate their performance on a prototype. More specifically, we: a) develop a set of algorithms required in evaluating basic relational algebraic queries on streams including binary and unary operators, b) identify and present the limitations imposed when programming such operators on the GPU, c) propose a framework that effectively uses our proposed operators in a way that amortizes the above-mentioned limitations.

Our work in this paper builds on earlier efforts [6, 5] that have shown the feasibility and effectiveness of implementing database operations with low-level functionalities of GPUs. It has been shown that a GPU can outperform the CPU in the evaluation of predicates, aggregation operations and in the sorting of large numerical arrays. Thus far, such techniques have been device-specific and confined to very specific pieces of hardware without having the ability to even "move" between different versions of the same GPU product line. We overcome this limitation by using higher level languages recently available for the programming of GPUs such as *CUDA* [8]. To effectively function on streaming data, tuples and/or values, our framework also takes into account changes that have been suggested in the relational model [7] to render traditional database operations suitable for stream processing.

The rest of this paper is organized as follows: Section 2 provides an architectural overview for GPUs and Section 3 outlines our proposed operations and our framework. In Section 4, we present preliminary results from our experimentation with a prototype. Section 5 presents related work and conclusions are found in Section 6.

MDS'09, November 30, 2009 Urbana Champaign, Illinois, USA

2. GRAPHICS SYSTEM ARCHITECTURE AND PROGRAMMING MODEL

We briefly discuss how the GPU inter-operates with other components of a computing system and subsequently present its internal structure as abstracted by the programming tools available today. By presenting the GPUs Single Instruction Multiple Data (SIMD) architecture we also sketch out the applications such processors are suitable for.

Figure 1 shows a typical system with a graphics card unit featuring its own substantial buffer space. The *PCI Express* through the *North Bridge*, helps connect the GPU to both main memory and CPU at a rate of multiple GBytes per second in both directions. It is worth point out at this stage that despite the significant transfer rates that a *PCI Express* may attain, by comparison such transfers demonstrate a functional latency that has to be considered.

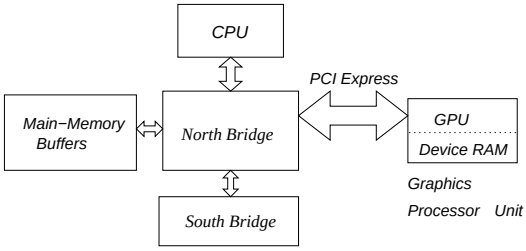

Figure 1: Overview of the chipset placement and communication in a modern computing system.

Graphics cards are built with a single operational objective: to apply the same transformation to a number of triangles and texture points at the same time. This naturally calls for the adoption of a SIMD execution model. Figure 2 depicts a high level abstract model for GPUs consisting of the GPU chip and the main-memory of the device. It is in the GPU chip that a number of threads may simultaneously execute the exact same part of code, called *kernel*. These threads have access to three types of memory a) *local registers* b) on chip memory *shared* among all threads and c) the *global* device memory. These memory resource types play respective role to those found in a classic multi-level cache system working with a CPU. However, there is a distinction here as there exist no transparent caching management mechanism. It is up to the programmer to request that specific data segments to be copied from the *global* memory to either *shared* memory and/or the *local registers* of the GPU unit. *Shared* memory can only hold a small fraction of data resident in the *global* memory. Transferring data from the *global* to the *shared* area is not negligible in terms of GPU-cycles.

Figure 3 shows the interaction between the various devices in a cooperative execution between CPU and the GPU elements. In order for the GPU to carry out a task, the CPU has to dispatch a specific code compiled for this purpose, the *kernel*. The CPU also has to transfer all requisite data to the *global* memory of the graphics card. *Kernels* are kept in specific instruction caches that are on the GPU chip and are dedicated for this purpose. Once CPU completes this initiation of the graphics processor it instructs it to commence operation and asynchronously may return to other pending tasks in the system. With its turn, the GPU executes the kernel present on the chip on data and places results back

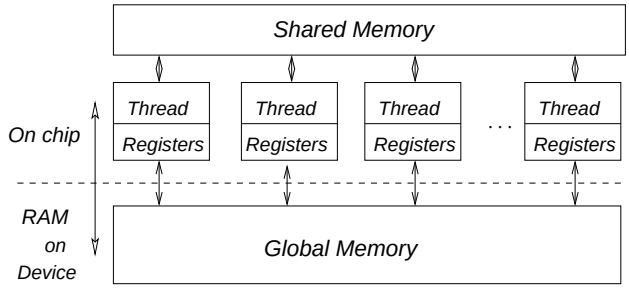

Figure 2: The GPU SIMD-architectural model along with *global, shared* memories and *registers*.

in the memory of the card. At this stage, it is the job of the CPU to *orchestrate* to fetch any results, if applicable, transport more data between the main buffers and the *global* area, and trigger once again the execution of the kernel.

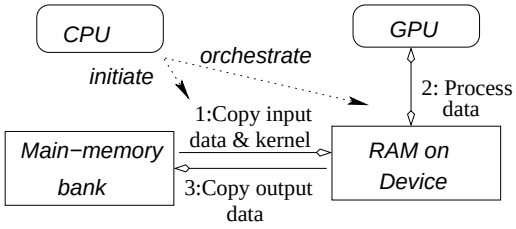

Figure 3: Flow of execution in an application taking advantage of the GPU resources.

GPUs are ideal components for handling high-frequency data streams at the application level. Overall this is achieved by the following unique characteristics:

1. the SIMD/multi-threaded co-processor architecture,
2. high throughput interconnects between main memory and device memory
3. latency between memory banks (main memory and device) that remains nearly-constant regardless of the volume of data transferred.

3. THE PROPOSED FRAMEWORK

Our framework is made of a number of GPU implemented operators and a component that orchestrates their execution. Here, we first give an overview of the most important features of the *CUDA SDK*, second we describe the implementation details of the operators and finally we present the *executor* component that wraps all operators in a framework.

3.1 The CUDA Functionalities

In a transparent to the programmer manner, *CUDA* transports the kernel from the main memory to the GPU (instruction caches) so that its execution can start. Kernels are written in *C* with the use of specialized tags. One of the most important such tag is the `__shared__` which designates that a specific variable is placed in the *shared*-area thus accessible by a number of threads. *CUDA* also makes available a set of memory management functions through which the programmer can allocate and copy memory segments in the *global* memory on the device.

The graphics card hardware provides a number of threads that may execute in "true"–parallel fashion. A layer is added over the hardware threads to seal the user from low-level operational threading aspects. *CUDA* organizes such logical threads into *blocks* whose size is programmer set. For instance while our hardware allowed for 32 parallel threads, we were using 256 threads grouped in the same block [1]. Kernels may require more threads than those of a single block, as a result the same kernel may be executing by threads belonging to different blocks. For example if we had a block of 256 threads and we wanted to perform an operation on an array with 512 elements, we could have one block where each thread would manage two elements or we could have two blocks of threads where each thread would handle a single element.

All threads in a block are able to access a single specific *shared* memory bank. We should point out that logical threads might not follow the "pure" SIMD execution model (discussed in the previous section) since the hardware may provide fewer actual threads. To overcome synchronization problems and address potential write-after-write, read-after-write and write-after-read hazards, the *CUDA SDK* provides appropriate synchronization functions.

In the modified SIMD abstraction –realized by *CUDA*–, all threads execute the same instruction. This may introduce some delays when threads follow different paths of execution. For instance, in case a kernel features an **if** statement and some threads evaluate it into *true* while others into *false*, the execution should evaluate the outcome of instructions in both the if and else code blocks. While instruction evaluation takes place in an execution path that a thread has not followed, this thread stays idle. In this way, a single thread may stall all the others causing the under-utilization of the GPU.

3.2 Relational Operations in GPU

For the time being, we assume that the streaming data are floating point numbers buffered in the main-memory of the system and are transferred in "batches" to the *global* memory region of the GPU. Processing of data mainly takes place through applying a series of relational stream operations over available data arrays. Most of operations required on streams can be expressed either as simple unary/binary `SQL` statements or simple algebraic trees involving a few data sources [3]. The *select* clause designates the shape of the result, the *from* indicates the source(s) and the *where* the predicate used for evaluation. We use bitmaps as the mechanism to communicate final results from the GPU to the CPU and intermediate outcomes from successive kernel executions. In this regard, every array of numbers (data) is accompanied with a bitmap of the same length indicating whether a number has been "selected". When the query evaluation reaches its final phase, it simply needs to check the entries of the bitmap that have been set and produce the corresponding output. The use of bitmaps allows us to perform many operations on a particular set of arrays inside the GPU without requiring the explicit return of intermediate results to the main-memory. All our GPU kernels take at least one bitmap as input and all –with the exception of join– have a bitmap as output. By using the output of a kernel as input to another, we may pipeline the GPU functions eliminating in this way the need to write data back to the memory of the host system. For instance when evaluating predicates expressed as a series of boolean operations that refer to the same data array, a single bitmap is used both as input and output in the all kernels.

While carrying out a join, the GPU produces a set of paired numbers that comply with the predicate. This set is sent back to the main memory of the host system. Clearly, this main memory access might "break" the pipeline of kernels and could hamper performance. Hence, it has to be avoided whenever possible.

In what follows we discuss: a simple binary predicate (*greater-than*) and a *join* operator based on nested-loops.

• GREATER: Algorithm 1 shows the implementations of the "*greater-than*" predicate for the GPU. The CPU implementation would be trivial, we essentially scan through the entire set of data and accordingly set the bitmap entries that correspond to qualifying data. In the case of GPU, we have one logical thread working on a single data value.

Provided that each block of threads may feature a finite number of logical such threads, we have to use a number of blocks any time we deal with arrays larger than 256 elements. Consequently, the first statement in the GPU-implementation of Algorithm 1 indicates the logical thread-*ID* that will work on a corresponding data element of array *A*. For example if *A* has 4,096 elements, the GPU will transparently deploy 16 blocks each using 256 logical threads to carry out the *greater-than* predicate computation.

The implementation of other binary operators such as $<$, $\leq$, $\geq$, and $\neq$ is identical to the *greater-than* predicate. The same is also the case for both unary (**not**) and binary (**and, or**) logical operations.

Algorithm 1 GPU-Implementation of ">" predicate

Input: `A[ ]`: Data array to be checked,
`Val` : Selected elements should be greater than this value.
Output: `Bitmap[ ]`: Bitmap of the selected elements,
Begin
  ID = Thread.ID + Block.ID*ThreadsPerBlock
  if A [ID] > Val **then**
    Bitmap [ID] = 1
  end if
End

•JOIN: Algorithm 2 depicts the realization of *join* in GPU-implementations. The equivalent CPU-version is the classic nested-loops approach where we assume that the two arrays to be joined *A* and *B* are main-memory resident.

In Algorithm 2, each thread copies an element of *A* on an on-chip register. Variable `localAelement` serves as a placeholder. Subsequently, the thread reads a portion of *B* into *shared* memory so that other threads (from the same block) can re-use it any time this is required. As we only have `ThreadsPerBlock` threads in a block, *B* is placed on the *shared* memory in parts of `ThreadsPerBlock` floating numbers. The outer loop in the GPU-implementation serves as an iterator over all parts of *B*. Before proceeding with the data processing we have to ensure that all threads have fetched their portions of *B*. Thus, we have to synchronize all threads by issuing a `CudaSynchronizeThreads` call.

A bitmap that could hold the result of this operation

[1]From the programmers perspective each thread is uniquely identified by the `block ID` and the `thread ID` within the block.

would call for unnecessary inter-memory movement: should we join two arrays of size N, the output bitmap would have been of N^2 size. As a solution to this problem we choose to produce a list of pairs with index-IDs to the array elements that are equal.

Algorithm 2 GPU-Implementation of Join

Input: `A[ ], B[ ]`: Arrays of data to be crosschecked, `BitmapB[ ]`: Bitmap of the "active" elements of B,
Output: `PairList[ ]`: List of the matching elements,
Begin

```
ID = Thread.ID + Block.ID*ThreadsPerBlock
localAelement = A[ID] {'Fetch locally'}

for Bpart=0;Bpart < Size Of B;
      Bpart += ThreadsPerBlock  do

  __shared__ ShrBitB[Thread.ID] =
            BitmapB[Thread.ID+Bpart*ThreadsPerBlock]
  __shared__ ShrB[Thread.ID] =
            B[Thread.ID+Bpart*ThreadsPerBlock]
  CudaSynchronizeThreads()
  {'Now BitmapB and B are in shared memory'}

  for Belement = 1 to ThreadsPerBlock  do
    if localAelement == ShrB [ Belement ]
    AND ShrBitB [ Belement ] > 0  then
      BitmapA [ Aelement ] = 1
      Append to PairList
          pair(ID,Belement+Bpart*ThreadsPerBlock)
    end if
  end for
end for
```

End

3.3 Operator Orchestration

The operators as described above are the building blocks of the proposed framework. It is the task of an external component to orchestrate them in an attempt to evaluate queries. Figure 4 shows the main components of our framework as well as how they are handled by the *executor*.

The input of the *executor* is a graph describing the stream operations involved in the evaluation of simple queries. For instance, the graph of Figure 4 has three nodes, two binary "*greater-than*" predicates and one *join*. The graph edges describe the input and/or output of each operator involved.

In the example of Figure 4 the two "*greater-than*" predicates require the transfer of the two arrays $T_0.C$ and $T_1.C$ and the two values D_0 and D_1 to the *global* GPU memory before they can be evaluated whereas the *join* requires that the output of the two "*greater-than*" predicates is available along with the $T_0.B$ and $T_1.B$ data arrays.

The *executor* is capable of identifying that the two output bitmaps (`Bitmap 1` and `2`) are intermediate results and thus they do not have to be transferred back to the main memory but instead they should be used as inputs to the *join* operator. In this way the operators are pipelined through the bitmaps used both as kernel inputs and outputs. By coordinating the kernel pipeline but also by setting the array size, and thus the window size of the streaming operators, the *executor* regulates the delays and latencies imposed by the hardware.

To sum-up, executor takes a graph as input and controls the graph execution by:

1. Allocating buffers both in system's and card's RAM
2. Transferring data to be used as input and/or output to/from the graphics card
3. Initiating the execution of the appropriate kernels on the GPU fed with the appropriate input data
4. Managing the bitmaps so as to correctly evaluate predicates in boolean operations

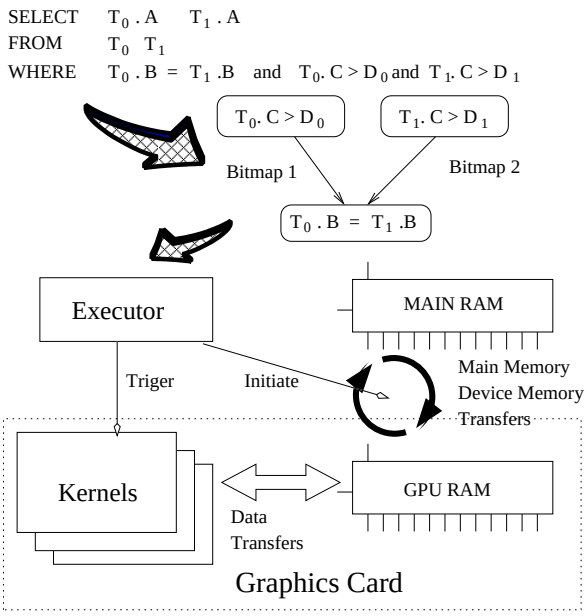

Figure 4: Simple graph of operations to take place in the GPU.

4. EXPERIMENTATION - EVALUATION

In our evaluation we first examine each kernel separately and compare it against the corresponding algorithm implemented in CPU. After kernel evaluation we move to examine the entire framework by feeding query graphs to the *executor* component.

4.1 Testing Environment and Metrics

The development, testing and evaluation of the framework took place on a Sony Vaio VGN-FZ11M laptop featuring an Intel Centrino Core 2 Duo T7100 at 1.8 GHz. The total system's RAM is 2 GB (2 x 1024 MB) DDR2 RAM at 667 MHz. The graphics card this laptop is equipped with is an NVIDIA GeForce 8400M GT with 64 MB of device dedicated RAM. The card is on a PCI Express 16x bus capable of transferring up to 6.4 GBps. The GPU has its 2 cores speed set at 450 MHz. Each core includes 16 multi-processors indicating that the hardware capabilities of this co-processor is at least comparable to the systems general purpose CPU. The internal bit representation of the stored numbers in both processors is the same, meaning that no extra cycles are lost in transformations.

Throughout our evaluation we measured execution time. This is the time required to read the input float arrays and bitmaps from the system's RAM, process them and write the results back. In the case of the CPU implementation this is the actual code execution time since the default behavior is to read and write data from/to main system's RAM. On the other hand evaluation of the GPU code segments includes

the time required to copy the input data from main memory to the device's memory and transfer the results back to main memory. As this is a time consuming task the GPU should be fast enough to overcome this handicap.

4.2 Evaluation of Basic kernel Operations

By testing the Algorithms presented in Section 3.2 we identify the circumstances under which exploiting the GPU resource is beneficial.

• GREATER: To test the performance of this algorithm we vary the input size. We gradually increase the number of values to be checked from 15,000 to 165,000 floating point numbers. The output is a bitmap whose size is of equal size with the input. Figure 5 presents the results of this evaluation.

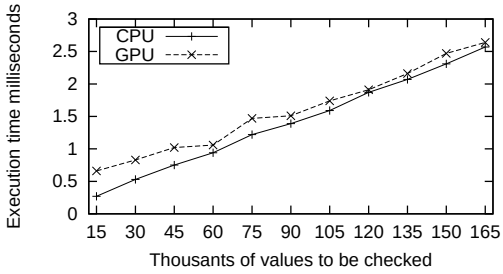

Figure 5: Evaluation of the "*greater-than*" predicate when increasing the input size (i.e., the number of values to be checked)

We find two points worth mentioning. First, the CPU has a small advantage over the GPU but as the amount of data processed increases this advantage diminishes. Second, the memory transfers between device and main memory impose an overhead that stems from the latency introduced by the PCI Express bus. This latency effect intensifies when the data to be transferred is less.

• JOIN: A join operation may produce output that by far exceeds the size of the input. Since the joined elements are copied back to the main system's RAM the output size may impose significant delays. To evaluate the performance of the implementation at hand we vary the join selectivity. First, we gradually increase the percentage of elements in one of the two input arrays that have join matches. In this step 100% matching corresponds to all elements producing a single join pair each. Here, in terms of output size, the amount of join pairs produced starts from zero and ultimately reaches the size of the input. In the second evaluation step we further increase the join selectivity by introducing element replication. In doing so we force elements to have multiple join matches and thus produce more than just one join pair. In this way the output becomes several times the size of the input. In our case we have the output ranging from one to the twenty times the size of the input arrays. Figure 6 present the first step evaluation results whereas Figure 7 present the experimentation results of the second evaluation step. When evaluating this join operator we use input batches of 9,000 elements each.

Our experimentation shows that the GPU implementation has to transfer the output from the device to the main memory thus its performance greatly depends on the join selectivity. It would have been beneficial to delegate the join operation to the graphics card when selectivity is low.

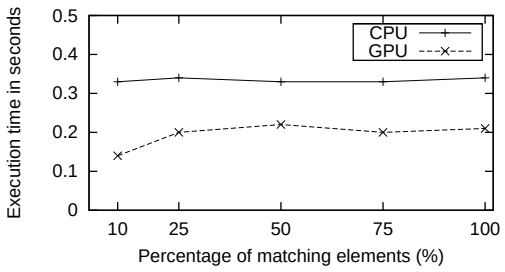

Figure 6: Evaluating the join operation when varying the element matching percentage.

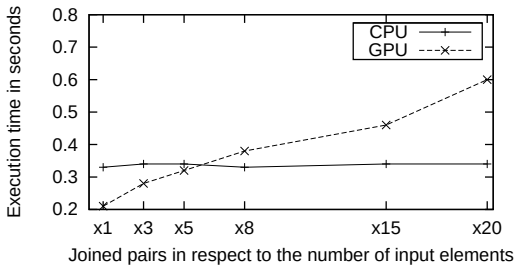

Figure 7: Evaluating the join operation when having elements produce multiple join pairs.

4.3 Framework Evaluation

To test the effectiveness of our framework we establish a test case: we prepare a graph to be fed as input to the *executor* component. The graph at hand is comprised out of a number of kernel calls and memory management operations. We measure the execution time with a) our framework in use, b) having each kernel execute as an autonomous component (i.e. return any results to the systems main memory) and c) a CPU only version of all kernels.

The graph of operations we use is presented in Figure 8. First two binary predicates are evaluated (called A and B). Their results are added by applying an "or" operation. The result along with an extra array, C, are joined in the last step. This sequence of operations could be the product of an SQL query:

```
SELECT T0.P, T1.P
FROM T0, T1
WHERE T0.C > D0 or T0.E > D1 and T0.B = T1.B;
```

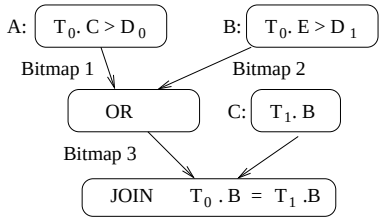

Figure 8: The graph used as a test case.

In this use-case, we chose all input arrays to be of size: 15,000 floating point numbers. During execution the first predicate (A) results in masking out 7,679 elements. After the OR operation between *Bitmap 1* and *Bitmap 2*, 13,068 entries reach the *join* operation through *Bitmap 3*. In the final step 6,790 pairs are produced through the join operator. All involved operators are triggered only once under a single input batch.

In Figure 9 we present the results of the execution time required to evaluate the graph of Figure 8. "Our GPU" represents our framework, the "One Op GPU" shows the execution time when we trigger the GPU kernels without the intervention of the *executor* component, the third column depicts the milliseconds it takes for the CPU to evaluate the same query graph. The essential difference between "Our GPU" and "One Op GPU" is that the later has to return the output data back to the main system memory. In our framework the use of bitmaps allows us to pipeline the kernels thus resulting in less main memory interactions and enhanced performance. The CPU comes third in terms of performance in our test case. In light of data whose size exceeds that of the arrays we used here, successive execution of the same execution graph is required. This will only expand the lead of the GPU has over the CPU.

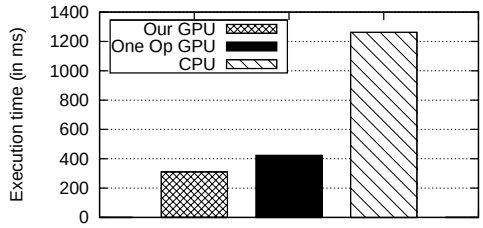

Figure 9: Evaluation of the proposed framework.

5. RELATED WORK

Early efforts to create systems capable of processing streams of data are discussed in [10, 3, 1]. The main objectives of these systems were to help in network analysis, security and serving monitoring applications. What all the above application systems share is their "modified" view of the relational algebra. This is necessitated by the streaming nature of data.

The potential for harvesting the GPU co-processor resources for data processing has been identified and even GPUs enhancements have been proposed [4]. The deployment of highly parallelized bitonic sorting algorithms has been proposed and their performance has been optimized in [5]. In a similar spirit, [6] demonstrated that it is feasible to implement fundamental relational algebra operations. The aforementioned efforts are precursors of our own work as we seek to harvest the additional GPU resources using high-level programming options available to the end-user.

6. CONCLUSIONS

Based on our GPU-implementation of fundamental operators, we proposed a framework for the materialization of (complex) algebraic query graphs built around the notions of SIMD execution of GPU threads and pipelining of intermediate results. Preliminary results using a prototype demonstrate faster wall times for the execution of various query types. Nevertheless, intensive and repetitive data transfer between the main-memory of the host system and the *global* memory of the GPU have yielded a number of occasions in which GPU materialization of the query graphs may be less effective than of its CPU-based counterpart. We plan to improve our framework by offering facilities for handling non-numeric data effectively and incorporating a hybrid approach that helps dynamically decide when to use both CPU and GPU resources simultaneously.

7. REFERENCES

[1] D. J. Abadi and et al. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 2003.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Mike, and H. Pat. Brook for gpus: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 2004.

[3] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 647–651, New York, NY, USA, 2003. ACM.

[4] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.

[5] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336, New York, NY, USA, 2006. ACM.

[6] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 215–226, New York, NY, USA, 2004. ACM.

[7] Y.-N. Law, H. Wang, and C. Zaniolo. Query languages and data models for database sequences and data streams. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 492–503. VLDB Endowment, 2004.

[8] NVIDIA. The cuda toolkit. 'http://www.nvidia.com/object/cuda_home.html, February 2007.

[9] Rapidmind. http://www.rapidmind.net/, January 2008.

[10] M. Sullivan. Tribeca: A stream database manager for network traffic analysis. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, page 594, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.