Combining Distributed Computing and Massively Parallel Devices to Accelerate Stream Data Processing

David Bednárek, Martin Kruliš, Petr Malý, Jakub Yaghob, Filip Zavoral, and Jaroslav Pokorný
Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic
email: {bednarek,krulis,malyp,yaghob,zavoral,pokorny}@ksi.mff.cuni.cz

***Abstract*—Data streaming systems have been successfully employed for various data processing tasks. Their main benefit is that they simplify the design of data-intensive applications and they introduce many opportunities for task, data, and pipeline parallelism. In this work, we are proposing an enhancement for data streaming systems that allows distributed processing of the data streams and also introduce parallel accelerators, which can be utilized for data parallel subtasks. The viability of our approach is verified by integrating the support for heterogeneous accelerators into the Bobox system, which is a parallel framework for data stream processing.**

Keywords–parallel processing; stream computing; distributed computing; parallel accelerators

I. Introduction

A significant part of the scientific community and high performance computing (HPC) community has become fascinated by the growing potential of GPU (graphics processing unit) accelerators in a number crunching applications. However, the GPUs excel only in specific tasks while in others their performance is inferior. In particular, tasks involving complex data structures struggle with the complexity of the GPU memory hierarchy. The utilization of GPUs is also limited by the necessity of transferring the data between the host system and the internal GPU memory, which brings additional overhead to the data processing. Consequently, complex data manipulation (which often occurs in database systems) is in most cases performed faster by CPUs than by GPUs and the fact that more efficient GPU algorithms exist for many subproblems is outweighed by the additional costs imposed by the GPU architectures.

At the same time, the growing size of data sets means that even applications that focus solely on numerical computations require sophisticated methods for data manipulation, which were previously known only in database systems. Although this does not necessarily mean that all applications must use a relational database or a database management system at all, an application must directly or indirectly use elaborate data structures to store its data on external media and to explicitly cache working sets of this data in operating memory. In addition, a scalable application must be able to balance the workload over a number of computing nodes, which results in intensive communication between the nodes conducted either directly or indirectly through a distributed file system or similar abstraction layer.

As a consequence of the growing potential of the GPUs and the expanding data sets, almost every computationally intensive application combines parts, which are best suited for GPUs, as well as parts that must be done by CPUs. In some cases, the performance of GPUs and CPUs on a particular subtask may be approximately equal; consequently, such a subtask may be subject of load balancing between both platforms.

These considerations lead to increased interest in heterogeneous computing systems, which combine both large number of CPUs (or at least a CPU with multiple cores), as well as one or more GPUs. In addition, many-core platforms like Intel Xeon Phi have emerged recently. They represent another step in CPU-GPU convergence trend which combines the generality of CPUs and the massive parallelism of GPUs. While the heterogeneous systems are already available as hardware, the software that is responsible for efficient hardware utilization is still immature. It is quite improbable that one solution will fit for every application, thus a number of thoroughly different approaches is being attempted.

In this work, we investigate the applicability of the stream data processing paradigm in heterogeneous computing platforms. The semantics of a stream naturally fits to the data flow between individual heterogeneous computing nodes, meaning both the communication between servers in a distributed system and the data movement between host memory and accelerators such as GPUs or Xeon Phi devices. Since the scheduler of such a system must be aware of all communication and synchronization inside the application, it requires that every communication between individual subtasks is explicitly expressed using streams. Although this restriction requires a different approach to application decomposition than traditional procedural programming, there are numerous examples of problems that were successfully modeled in the stream paradigm. In particular, almost all database systems convert every query to an execution plan, consisting of operators chained to a tree or directed acyclic graph, which almost exactly fits to the decomposition imposed by the stream systems. Given the necessity of database-like processing of large data sets, computational applications may often employ presented approach (or a very similar one) directly.

The paper is organized as follows. Section II summarizes related work and Section III revises most important facts about parallel hardware. The architecture of the distributed Bobox and necessary modifications of the existing implementation are

Copyright (c) IARIA, 2015.

presented in Section IV. The integration process of dependent accelerators is proposed in Section V and Section VI concludes the paper.

II. Related Work

A. Bobox

The Bobox framework [1] was designed to support development of data-intensive parallel computations. The main idea behind Bobox is to divide large tasks into many simple tasks that can be arranged into a nonlinear pipeline while preserving transparency of the distribution logic. The tasks are executed in parallel and the execution is driven by the availability of data on their inputs. The developers do not need to be concerned about technical issues such as synchronization, scheduling, or race conditions [2].

The system can be easily used as a database execution engine; each query language requires its own frontend that translates a request (query) into a definition of the structure of the pipeline that corresponds to the query [3]. Bobox uses its own language called Bobolang [4] in which the execution plan in described. The bobolang code can be generated by database frontends, but it can also be created directly by the developer of a parallel application.

B. Streaming Systems

Stream processing has no formal definition and this term is used to describe a variety of systems [5]. One of the founding systems in this domain is StreamIt [6], which uses streaming in combination with general parallel data processing. It provides a domain specific language that follows the Synchronous Data Flow paradigm [7] and it is able to utilize various hardware architectures such as distributed multi-core systems.

Data stream management systems (DSMS) form a particular category of streaming systems. Systems as Borealis [8], STREAM [9], or NiagarsST [10] are used for real-time querying of data streams, which are usually continuous and infinite. These systems typically provide a high-level runtime API accompanied by either a set of predefined operators or by a specialized query language.

The class of hardware-specific streaming languages and systems is represented by StreamC/KernelC [11], which is designed for the Imagine processor [12] or BrookGPU [13] designed for GPUs.

The Auto-Pipe [14] is intended to simplify development of complex streaming applications on various architectures. It is based on specification of operators, resources, and a topology of these resources. The system provides an optimal mapping between the operators and the resources.

Intel Threading Building Blocks (TBB) [15] is a framework for parallel computing. It cannot be strictly regarded as a streaming system; however, it provides similar functionality through its Flow Graph component [16]. In addition, it provides basic algorithm templates, so it can be employed for parallel data processing in more general way. Another widely used framework is OpenMP [17] with an extension for data stream processing [18] that enables description of the pipeline stages of an algorithm. However, more complex constructions are tricky and rather difficult to implement.

The MapReduce programming model [19] is another example of a specific data streaming system with fixed number of stages. It is mainly intended for processing of large data in a distributed environment [20]. Despite the fact that it is considered a step back [21] in parallel database engines, this model gained significant attention since it can be easily utilized for large tasks and it provides sufficient scalability and robustness. Unfortunately, it does not support more complex execution plans like non-linear pipelines. This drawback is partially solved by the Apache Pig [22] environment together with the Pig-Latin language [23], which is able to transform non-linear pipeline into a sequence of MapReduce programs.

C. Accelerated Database Engines

Many-core parallel accelerators (especially GPUs) have proven useful for accelerating individual database operations. For instance, GPUTeraSort [24] is an algorithm developed to sort database rows, and demonstrated significant performance improvements over serial sorting methods. Similarly, the relational join operation has been successfully implemented for GPUs [25].

Using external procedures in big commercial databases, such as Oracle [26] or PostgreSQL [27], GPU hardware can be employed to accelerate certain critical operations. In [28], the authors propose a design of a GDB database accessed through a rich set of individual parallelizable operations.

III. Computing on Parallel and Distributed Architectures

In this section, we revise important facts related to the parallel computations and parallel architectures which became popular in the past few decades. We will mention mainstream architectures, which are employed widely in various applications.

A. CPU and Multi-CPU Architectures

A multicore CPU is perhaps the most typical representative of current parallel hardware. It comprises several physical cores, which are further divided into logical cores by means of hyper threading (Intel) or dual-core module (AMD) technology. Logical cores share most of the units of the physical core in order to improve utilization of the physical core. Most of the current CPUs employ three-level memory cache, which reduces the latency of operating memory. In the parallel processing, we must consider the fact that some levels of the cache are shared among cores. Therefore, the overall cache capacity available solely for individual cores is reduced; however, the shared cache can improve situations where the cores read the same memory or need to exchange small amounts of data among themselves.

Multiprocessor systems are composed of several multicore CPUs. The systems which use an independent memory controller are usually organized as *symmetric multiprocessing* systems. If each CPU has a memory controller integrated, the system has to be organized as *nonuniform memory architecture* (NUMA) and each CPU manages its part of the (shared) operating memory. In modern NUMA systems, the hardware is able to maintain cache coherency (ccNUMA); thus, the memory access is transparent. Nevertheless, the latency and bandwidth are not uniform across the memory range, since any access to memory managed by different CPU has to be routed through interprocessor communication grid.

B. Parallel Accelerators

Parallel accelerators, especially GPUs, form an important group of architectures used for parallel computing. An accelerator is usually an extension card connected to the host system which controls its utilization. Most accelerators are capable of performing computations concurrently to the host CPU; however, the host system is still responsible for providing (or at least passing on) the input data. In this work, we are particularly interested in GPUs and Intel Xeon Phi devices, which are typical representatives of parallel accelerators of the day.

GPU devices have become quite popular platform for data parallel processing. A GPU processor consists of several *symmetric multiprocessing units* (SMPs). Each SMP comprises multiple GPU cores, single instruction decoder and scheduler, L1 cache and shared memory. The cores of the SMP are tightly coupled since they share its internal resources and they all execute the same code in a SIMT (Single Instruction, Multiple Threads) fashion, i.e., in a way that all cores perform the same instructions. This model makes GPUs perfect devices for *data parallelism*, where the same routine is executed many times on different data objects.

The memory hierarchy of GPUs is significantly different from current CPUs. A GPU device has its own *global memory*, each SMP has *shared memory* which is much faster but accessible only by the cores of the SMP, and finally, each core has its own registers (assigned from the SMP registry pool). This hierarchy is particularly important for fine-tuning and optimizations of the GPU code. Furthermore, the GPU processor cannot access data in the *host memory* directly, but the data must be transferred to the global memory first. These transfers have to be carefully planned, so they will not present a bottleneck of the data processing. Fortunately, modern GPUs are capable of performing two concurrent data transfers (in and out, since PCIe bus is duplex) while the GPU processor is computing.

Intel Xeon Phi is a many-core device usually delivered in the form of accelerator card. It resembles GPUs in the main aspects, since the accelerator has its own memory and its processor is designed to be massively parallel. The most important difference is that the Xeon Phi processor is based on x86 architecture being used in common multicore CPUs. Therefore, the device is capable of running more general code and it even hosts an internal operating system, which makes it more independent on the host.

C. Distributed Computing

The performance scaling of individual servers (i.e., vertical scaling) has its limits, since we can incorporate only limited number of accelerators and employ only limited number of CPUs in a single host. In order to increase performance further, we need to utilize multiple servers, which are connected by some networking technology. This approach is called horizontal scaling.

Horizontal scaling brings new challenges, since the data transfers over the network require special handling and introduce additional possible bottlenecks to the whole system. Furthermore, incorporating many nodes in one system increases the chance of individual failures and also introduces new types of failures caused by communication.

In the remainder of this paper, we will assume that the system is composed of multiple nodes, which all employ some technology for message passing (e.g., MPI). On the other hand, we have no additional requirements on the individual nodes, and the whole environment could be quite heterogeneous. In other words, the distributed system may employ different servers with different configurations.

IV. Architecture of Distributed Bobox

The Bobox framework evaluates a data streaming program, which is expressed in a form of oriented graph called *execution plan*. Vertices of the execution plan (denoted *boxes*) perform individual data processing operations and each operation is implemented as a sequential routine in C++ language. The *edges* prescribe the data flow among the boxes. The data transfers on the edges are aggregated for efficiency reasons and transferred in bundles called *envelopes*. An envelope is a fragment of a data stream which consists of multiple rows, where each row has prescribed number of columns of prescribed type. Internally, an envelope uses column-oriented format, so it is organized as a tuple of columns where each column is an array of cells and columns of one envelope have the same number of items. The types and numbers of columns are defined in the *edge descriptor*, which is embedded in the execution plan specification. An example of execution plan is depicted in Figure 1.

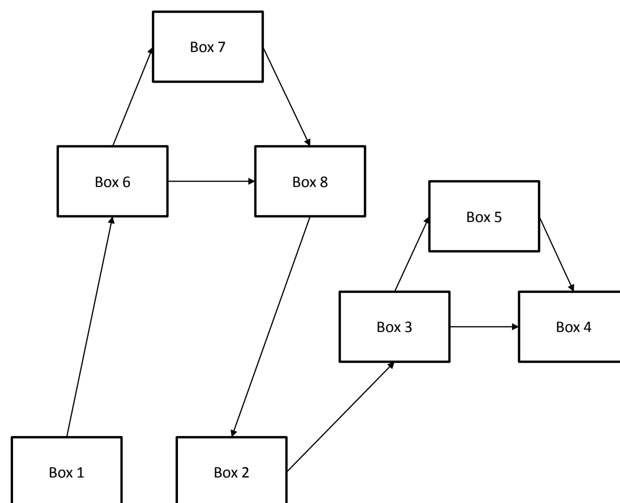

Figure 1. An example of Bobox execution plan

Manual construction of an execution plan in a procedural or object-oriented language would be tedious. Furthermore, the plan has to be tuned for the hardware configuration (e.g., number of available CPU cores) to achieve optimal performance. For these reasons, we have developed the Bobolang language [4], which simplifies the creation of execution plans. Bobolang is used for the definition of execution *model*. The model expresses the evaluation logic of a streaming data application, but it can generalize some details, which can be specified later (e.g., according to the hardware properties) when the model is instantiated into an execution plan. A model is either generated by a database front-end (e.g., a SPARQL engine [29]) or it may be created manually by a developer.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-408-4

Modifications described in this paper require updates of both the parallel runtime that processes execution plans and the Bobolang language that describes the model of a streaming application. Individual issues of these updates are addressed in this section.

A. Considering Distributed Hardware

Distributed Bobox runs on a *cluster*, which is defined as a set of computers connected with each other by one *connection technology*. In this case, the term connection technology refers to the API used for communication regardless of its physical realization in hardware. Each computer connected to the cluster is denoted a *node*. A node may have one or more accelerators and the nodes in a cluster do not have to be homogeneous – i.e., nodes may be built on different architectures and may utilize variable numbers and types of accelerators. An accelerator is accessible locally from its node (which we also denote *host* when emphasizing its relation to the accelerator) and the node must provide either explicit management of the accelerator or extend the connection technology so the accelerator can be attached to the cluster as a nested node. An example of a cluster is presented in Figure 2.

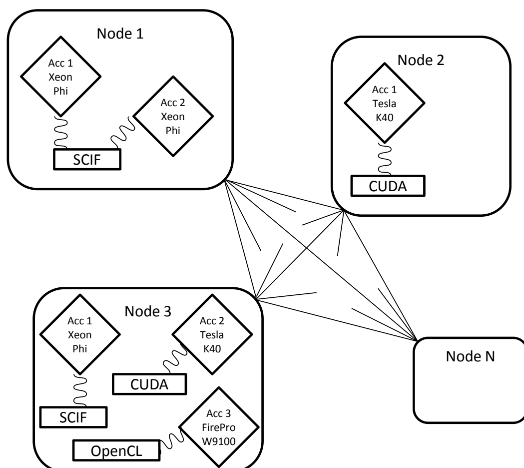

Figure 2. Example of cluster architecture from the Bobox perspective

For the purposes of our work, we will distinguish between *dependent* accelerators and *independent* accelerators. Dependent accelerators are fully controlled from the host system (i.e., from a code running in a CPU thread) thus they require additional management from our distributed framework. An example of dependent accelerator is a GPU, which can perform computations concurrently with the host system, but all computational tasks must be issued from a host CPU. Integration of dependent accelerators is detailed in Section V.

Independent accelerators can execute more complex code which is capable of managing the internal workload and which may also initiate communication and data transfers to and from the host system. Intel Xeon Phi is an example of independent accelerator, since it is operated internally by a Linux system and it can be perceived as a separate computer, which is interconnected with the host system via a PCI-Express bus. In the remainder of this work, the independent accelerators are treated the same way as cluster nodes.

Bobox relies on fine grained parallelism, so the execution plans of data queries [3][29] usually comprise hundreds or even thousands (single-threaded) boxes. Moreover, many front-end query compilers are capable of providing additional estimates regarding the amount of data being transferred on the edges. This information can be used to divide the execution plan into parts, which are then distributed over the nodes of the cluster.

This work focuses solely on the static distribution of the execution plans. Dynamic distribution and load balancing can be additionally employed; however, it raises many additional issues such as workload measurement, migration of box internal states, etc. Most of these details are beyond the scope of this paper.

An example of a static distribution of the execution plan is depicted in Figure 3. The plan is divided into three parts and distributed among three nodes. Nodes 1 and 2 are independent servers, while node 1 is also host for a Xeon Phi device.

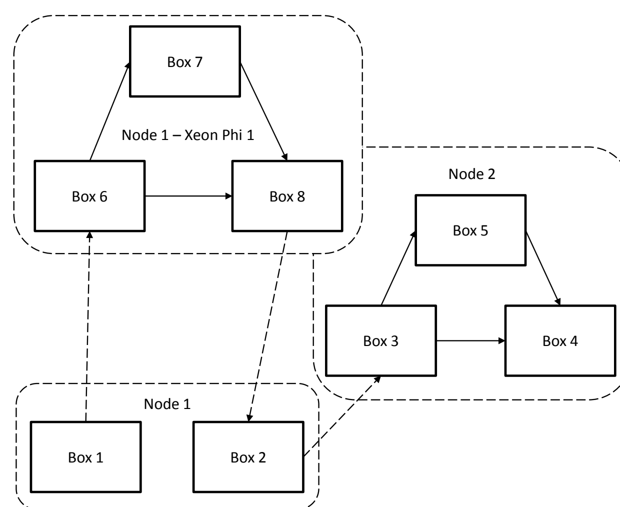

Figure 3. Example of an execution plan distributed on a cluster

The distribution of the execution plan requires additional method of envelope transfers. In its original parallel mode, the Bobox runtime assumes that all boxes can share data via the operating memory of the node. Therefore, passing an envelope from one box to another is only a matter of passing one memory pointer. Furthermore, shared memory allows immutable data to be shared among boxes.

In the distributed environment, the envelope must be hard-copied whenever crossing a border of a cluster node. In order to minimize modifications of the original runtime, we introduced two specialized boxes.

- A *send box* (S-box) consumes envelopes, performs data serialization, and send the data to another node using connection technology.
- A *receive box* (R-box) receives data from the connection technology, deserializes them, and emits them as envelopes.

The S-Boxes and R-boxes form inseparable pairs, where each pair represents one unidirectional peer-to-peer connection. Different pairs may use different types of communication technologies, but both boxes of a single pair must use the same

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-408-4

technology. The example from Figure 3, augmented with a set of corresponding send/receive boxes is presented in Figure 4. The solid arrows represent normal edges of the execution plan (i.e., where the envelopes are passed via shared memory) and the dashed arrows symbolize connections between transport boxes.

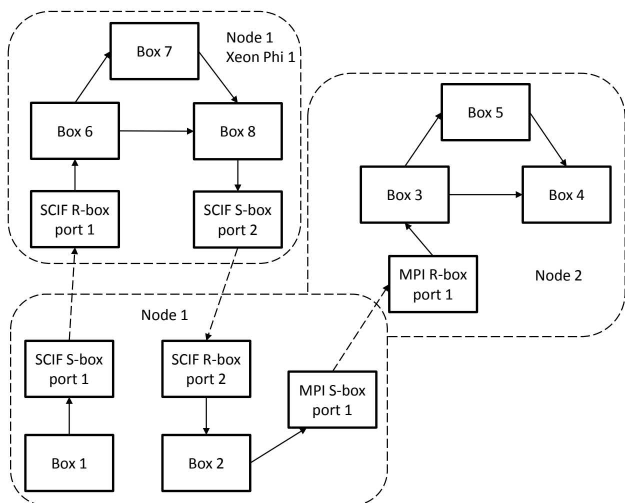

Figure 4. A distributed execution plan connected with communication boxes

B. Construction of Distributed Execution Plan

As mentioned before, the Bobox execution plan is not assembled manually by the programmer, but it is created automatically as an instance of a given application model. The model is written in Bobolang and it may also be generated from a query language such as SPARQL or SQL by a Bobox frontend module. In a single-node environment, the instantiation of a model into the execution plan is straightforward. In the distributed environment, the situation becomes much more complex, since the execution plan has to be divided among individual nodes and appropriate send and receive boxes must be added at the node boundaries. The situation gets even more complicated when dependent accelerators are being utilized in the system.

In order to maintain generality and sufficient level of abstraction, we have modified the execution plan instantiation process in the following way. A new intermediate plan abstraction called *bound model* is introduced and the term 'bound' emphasized that the model is designated only for the current cluster configuration. The bound model is also described in Bobolang and it is created from the model by the *concretization* process, which is performed by a *placement binder*. The complete Bobox abstraction overview and terminology is summarized in Figure 5.

The placement binder is a separate component which has complete information regarding the cluster topology, hardware properties of the nodes, and properties of the attached accelerators. Its concretization process transforms the original model to the bound model by replacing selected edges by S-box/R-box pairs and by replacing selected boxes or subgraphs by accelerator boxes that contain nested execution plans for selected accelerators. Furthermore, the placement binder assigns each box to a specific node in the cluster and each accelerator box to

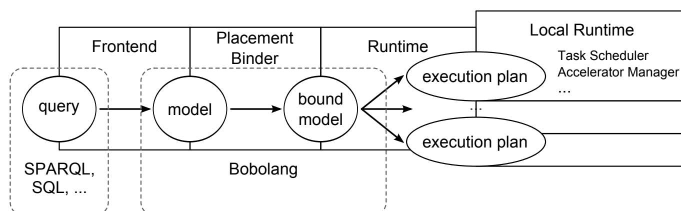

Figure 5. Bobox model abstraction and execution plan construction

an accelerator. These assignments are represented as Bobolang box annotations and they are recognized by the Bobox runtime that instantiates the models.

The bound model is subsequently distributed to every node of the cluster and passed to their respective Bobox runtimes. The local runtimes work in the same way as the single-node runtime, except that each runtime instantiates only those boxes (and their adjacent edges), which are designated to its respective node.

C. Technical Details

In the remainder of this section, we address several technical issues, which are essential for the distributed implementation. The first problem is caused by the diversity of addressing schemes that the underlying communication technologies use in S-box/R-box transfers (MPI uses message tags, SCIF uses ports, TCP/IP uses address:port pairs, etc.). Since the placement binder has to be independent on the communication technologies, it assigns each S-box and R-box an identification number called *port*. The runtime is responsible for mapping these port numbers to identification tokens of particular communication technologies. The mapping is stored in a distributed form, so the corresponding pairs of S/R-boxes can establish their data connections. The placement binder is responsible for correct port assignments (i.e., that there is exactly one S-box and one R-box for each issued port number).

The Bobox runtime relies on an external startup mechanism (e.g., such as Hydra or MPD, which are used by MPI technology), which is responsible for starting a Bobox runtime process on every node of the cluster. After the local runtime starts and detects the local accelerators, it starts the appropriate versions of the Bobox runtime on the independent accelerators and the accelerator management threads for each dependent accelerator. Finally, the local runtimes use group communication primitives to exchange their configurations, so that each node knows the configuration of the whole cluster.

Section IV-B describes the process of execution plan creation and distribution. The initial model can be submitted to Bobox via any node, since the placement binder is replicated on every host and the bound model is distributed via distributed communication primitives of the runtime. If necessary, the Bobox front-end modules responsible for translating database queries into models can be replicated as well and the Bobox can be used as platform for a distributed database management system.

The proposed solution is highly scalable and it is expected to be applicable for large clusters. It has no single point of

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-408-4

failure since the hosts are mutually interchangeable. Nevertheless, the current implementation does not handle node failures and reliability is a subject of our future work.

V. Parallel Accelerators

In the previous section, we have described our proposed solutions for distributed processing and for independent accelerators, which are treated as independent nodes of the cluster. In this section, we address the issues of the dependent accelerators.

A. Integrating Accelerators into System

Dependent accelerators are more complicated, since they need to be managed from the host system. In case of GPUs (which are typical representatives of dependent accelerators), there are two basic approaches that are supported by current frameworks such as CUDA [30] or OpenCL [31]:

- The GPU can be managed by a dedicated thread and all operations are invoked as blocking operations (i.e., the thread is resumed by the operating system, when the issued work on the GPU has concluded).
- The GPU operations are issued in asynchronous manner so the thread which has issued them can process other tasks. The GPU events (such as work completion) can be either polled by the host code, or reported via asynchronous callbacks executed in a service thread.

In our system, we primarily use the first approach, but the second approach is additionally employed in special cases. Bobox runtime (i.e., its task scheduler) already support blocking operations [32] without limiting CPU utilization. The thread dedicated for GPU workload management uses internal API functions to notify the task scheduler that the thread is about to invoke blocking operation. The scheduler can wake up a replacement thread which supplants the blocked thread in the thread pool, so the size of the thread pool always corresponds to the available computational cores.

Modern GPUs [33] are capable of overlapping data transfers with kernel execution or even executing multiple kernels if they consist of only a few thread blocks. However, the GPU driver may exploit these features only when the overlapping work has been issued without explicit synchronization at the host side. To comply with these requirements of GPU API, we have additionally employed asynchronous operations in special cases when the host code needs to issue overlapping tasks.

B. Exploiting Data Parallelism

The parallel accelerators present a potential complication regarding their optimal utilization. The original Bobox framework and its distributed version have presented themselves very efficient for the parallel processing of OLAP workload; however, the framework itself employs task parallelism, which can be used to process data parallel and pipeline parallel problems as well. On the other hand, current accelerators (especially GPUs) are suitable mainly for data parallel problems and other forms of parallelism are difficult to express or even encumbered with serious overhead.

In order to avoid this problem, the execution plan should be designed or generated in a way that allows maximal applicability of the accelerators. In other words, the data parallel parts of the execution plan should be primarily assigned to accelerators whilst the task parallel or even sequential parts of the plan should be processed by CPUs. In order to simplify the design of the application model, Bobox framework introduces two techniques:

- operator composition
- envelope aggregation

Operator composition is one of the key features, which has been present in Bobox since its first versions. A composed box is treated as regular box in the execution plan, but it holds a nested execution plan instead of C++ routine. We have extended this feature to designate parts of the execution plan for accelerator processing. These parts are represented as specialized operators called *accelerator boxes* and these boxes hold a nested execution plan dedicated solely for an accelerator. Furthermore, the accelerator box carries annotations that identify the target accelerator which is responsible for the processing of the box. These annotations can also hold additional execution details, such as number of accelerator threads or required internal buffer sizes. The whole idea is depicted in Figure 6.

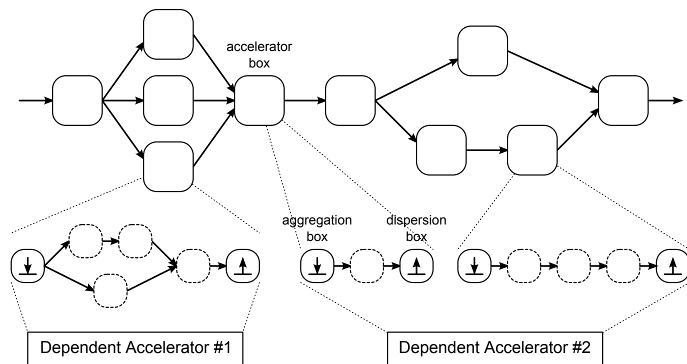

Figure 6. Designating parts of execution plan for accelerators

Regular boxes process data in portions called envelopes. Envelope size is carefully selected by the runtime based on the hardware properties of the target system – especially the CPU cache size. Accelerators need to process data in lager batches, in order to exploit the data parallelism principle. Therefore, an accelerator plan has an *envelope aggregator box* at its very beginning. This operator use heuristics and information from operator annotations to select the batch size for its current accelerator box. It aggregates data from multiple incoming envelopes and the aggregated envelope is transferred into the internal memory of the accelerator. Analogically, the plan represented in an accelerator box is ended by a *dispersion box*, which transfers the result data from the accelerator memory back to host memory and divides them into envelopes of regular size (suitable for CPU processing).

Finally, we need to specify how the accelerator plans are executed on accelerators. Even though the accelerator boxes are assigned to specific accelerators, the workload cannot be planned strictly statically since it is heavily data-driven. Therefore, the placement binder can employ oversubscription – i.e., assign multiple accelerator boxes to the same accelerator. The accelerator management thread plans the workload dynamically to maximize the overall utilization of the accelerator

and to increase the chance of data transfers and computations overlapping.

C. Memory Allocation

Perhaps the most problematic issue of the accelerator utilization is its memory allocation and management. Current accelerators are usually equipped with several gigabytes of internal memory, which is approximately $10 - 100\times$ smaller amount than the operating memory of modern NUMA servers. Therefore, an accelerator cannot replicate the entire content of the host memory and the data has to be copied only as necessary.

Internal execution plan of an accelerator box requires four buffers. The buffers may have complex internal structure, or even be composed from several memory blocks. However, we are omitting technical details and focusing on the purpose of these buffers. These buffers need to be allocated in the internal memory of the accelerator:

- buffer for input data,
- buffer for output results,
- buffer for internal data (operator state),
- and buffer for temporary data.

The input data buffer stores data from the aggregated envelopes which are transferred from the host. Analogically, output buffer is designated for data yielded by the accelerator box, which are transferred back to the host memory. Both these buffers are present in at least two instances, so that one pair of input/output buffers can hold data that are being processed by the accelerator, whilst the other buffers are engaged in concurrent data transfers (the following input envelope is being copied to the accelerator and the preceding result envelope is being copied to the host memory).

The buffer for internal data holds the current state of the operator which needs to be maintained across the processed envelopes. The consistency of the internal operator state has to be achieved by internal synchronization implemented in every part of the execution plan of the accelerator box. The scheduler ensures that at most one envelope is being processed by the accelerator at any time (except for the external data transfers).

The temporary buffers are designed to hold intermediate data that may be required by the execution plan of the accelerator and that cannot be accommodated by internal storage capacities of the computational units such as registers or shared memory of an SMP in case of GPUs. Unlike the internal state buffer, data in the temporary buffer does not have to be kept between the processing of two subsequent envelopes nor they have to be initialized before the first envelope is processed.

All buffers are allocated when the accelerator box receives its first incoming envelope. Furthermore, the internal data buffer has to be filled with the initial state of the operator. The buffers are disposed when the accelerator box receives termination envelope (the poisoned pill) and when the last aggregated envelope is processed.

D. Deadlocks

Multiple accelerator boxes can be assigned to a single accelerator and each one will attempt to allocate all its memory buffers when it receives first portion of data to process. If the overall size of the buffers required by all boxes assigned to an accelerator exceeds its available memory and if the data flow pattern requires that all the accelerators are working simultaneously, the overall execution will end by either failure or by a deadlock.

A naïve solution to this problem is to instruct the execution plan designer not to exceed the combined memory capacity of any accelerator, except in cases when boxes assigned to the same accelerator (and which possibly exceed its total memory capacity) are strictly divided by an implicit barrier such as a global sort operator placed between them in the execution plan. In this approach, a failure to allocate memory buffers for an accelerator box should be treated as a global failure of the whole execution plan and the responsibility of the frontend.

Unfortunately, the naïve solution limits the utilization of the accelerators quite severely. Hence, we propose a more elaborate model that allows allocator memory oversubscription and ensures successful execution. When an accelerator box fails to allocate its buffers, its execution is postponed until another box occupying the accelerator concludes its work and releases its buffers. Such behaviour may naturally lead to deadlock which has to either avoided or recovered. There are two possible approaches:

- The accelerator manager monitors the workload of all active boxes assigned to the accelerator and all requests from boxes that cannot be accommodated due to the limited memory capacity. When a deadlock is detected, the manager swaps internal state of selected accelerator box to the host memory and activate one of the waiting accelerator boxes. The box swapping can be even performed preemptively, when an accelerator box does not have any input data for some time. Let us emphasize, that only the internal data buffer needs to be swapped. If the box does not have any internal state or the internal state is relatively small, the box swapping can be very fast.
- Every accelerator box has an alternative backup execution plan that consist of regular boxes (i.e., boxes executed on CPU). If an accelerator box fails to allocate memory on an accelerator, it switches for the backup execution plan and starts processing its workload on the host system. Since the boxes used in the execution plans are usually developed so that Bobox does not require accelerators, constructing an alternative backup plan for each accelerator box should be rather simple.

Both presented approaches can also be combined. Furthermore, if the internal state of the accelerator box and its backup execution plan can be maintained in a binary compatible format, an accelerator box can switch between GPU plan and the CPU backup plan dynamically by swapping the internal state to/from the accelerator.

VI. Conclusions

The main contribution of this paper is a description of the architecture of a heterogeneous distributed system. The unique feature of the system is a combination of three modern computing platforms – multi-core CPUs, many-core accelerators like Intel Xeon Phi, and GPU-based accelerators. In particular, we investigated the differences between these two types of accelerators and suggested both static distribution of execution

plans, as well as dynamic swapping of work between hosts and their accelerators.

We are currently experimenting with all the described approaches; however, the dynamically evolving properties of accelerators requires larger empirical data to make clear conclusions on the strategy. In our future work, we will measure various types of workloads processed on various hardware configurations, in order to compare different load-distribution strategies.

ACKNOWLEDGMENT

This work was supported by the Czech Science Foundation (GACR), projects P103-13-08195S and P103-14-14292P, by Charles University Grant Agency (GAUK) project 122214, and by Specific Research project SVV-2014-260100.

REFERENCES

[1] D. Bednarek, J. Dokulil, J. Yaghob, and F. Zavoral, “Bobox: Parallelization Framework for Data Processing,” Advances in Information Technology and Applied Computing, 2012, pp. 189–194.

[2] D. Bednárek, J. Dokulil, J. Yaghob, and F. Zavoral, “Data-flow awareness in parallel data processing,” in Intelligent Distributed Computing VI. Springer, 2013, pp. 149–154.

[3] Z. Falt, D. Bednarek, M. Cermak, and F. Zavoral, “On Parallel Evaluation of SPARQL Queries,” in DBKDA 2012, The Fourth International Conference on Advances in Databases, Knowledge, and Data Applications, 2012, pp. 97–102.

[4] Z. Falt, D. Bednárek, M. Kruliš, J. Yaghob, and F. Zavoral, “Bobolang: A language for parallel streaming applications,” in Proceedings of the 23rd international symposium on High-performance parallel and distributed computing. ACM, 2014, pp. 311–314.

[5] R. Stephens, “A survey of stream processing,” Acta Informatica, vol. 34, no. 7, 1997, pp. 491–541.

[6] W. Thies, M. Karczmarek, and S. Amarasinghe, “StreamIt: A language for streaming applications,” in Compiler Construction. Springer, 2002, pp. 179–196.

[7] E. A. Lee and D. G. Messerschmitt, “Synchronous data flow,” Proceedings of the IEEE, vol. 75, no. 9, 1987, pp. 1235–1245.

[8] D. J. Abadi et al., “The Design of the Borealis Stream Processing Engine,” in CIDR, vol. 5, 2005, pp. 277–289.

[9] A. Arasu et al., “STREAM: the stanford stream data manager (demonstration description),” in Proceedings of the 2003 ACM SIGMOD international conference on Management of data. ACM, 2003, p. 665.

[10] D. Maier et al., “NiagaraST,” 2010, http://datalab.cs.pdx.edu/niagara/ [Online, retrieved: February, 2015].

[11] A. Das, W. J. Dally, and P. Mattson, “Compiling for stream processing,” in Proceedings of the 15th international conference on Parallel architectures and compilation techniques. ACM, 2006, pp. 33–42.

[12] P. Mattson, “A programming system for the imagine media processor,” Ph.D. dissertation, Stanford University, 2002.

[13] I. Buck et al., “Brook for GPUs: stream computing on graphics hardware,” in ACM Transactions on Graphics (TOG), vol. 23, no. 3. ACM, 2004, pp. 777–786.

[14] R. D. Chamberlain et al., “Auto-Pipe: Streaming applications on architecturally diverse systems,” Computer, vol. 43, no. 3, 2010, pp. 42–49.

[15] J. Reinders, Intel Threading building blocks. O’Reilly, 2007.

[16] “Flow Graph,” http://www.threadingbuildingblocks.org/docs/help/reference/flow_graph.htm [Online, retrieved: February, 2015].

[17] R. Chandra, Parallel programming in OpenMP. Morgan Kaufmann, 2001.

[18] A. Pop and A. Cohen, “A stream-computing extension to OpenMP,” in Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers. ACM, 2011, pp. 5–14.

[19] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” Communications of the ACM, vol. 51, no. 1, 2008, pp. 107–113.

[20] R. Stewart and J. Singer, “Comparing fork/join and MapReduce,” Technical Report HW-MACS-TR-0096, Department of Computer Science, Heriot-Watt University, Tech. Rep., 2012.

[21] M. Stonebraker et al., “MapReduce and parallel DBMSs: friends or foes?” Communications of the ACM, vol. 53, no. 1, 2010, pp. 64–71.

[22] “Apache Pig,” http://pig.apache.org/ [Online; accessed 2014-03-18].

[23] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, “Pig latin: a not-so-foreign language for data processing,” in Proceedings of the 2008 ACM SIGMOD international conference on Management of data. ACM, 2008, pp. 1099–1110.

[24] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, “Gputerasort: high performance graphics co-processor sorting for large database management,” in Proceedings of the 2006 ACM SIGMOD international conference on Management of data. ACM, 2006, pp. 325–336.

[25] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, “Gpu join processing revisited,” in Proceedings of the Eighth International Workshop on Data Management on New Hardware. ACM, 2012, pp. 55–62.

[26] “Oracle,” http://www.oracle.com [Online, retrieved: February, 2015].

[27] “PostgreSQL,” http://www.postgresql.com [Online, retrieved: February, 2015].

[28] B. He et al., “Relational query coprocessing on graphics processors,” ACM Transactions on Database Systems (TODS), vol. 34, no. 4, 2009, p. 21.

[29] Z. Falt, M. Cermak, J. Dokulil, and F. Zavoral, “Parallel sparql query processing using bobox,” International Journal On Advances in Intelligent Systems, vol. 5, no. 3 and 4, 2012, pp. 302–314.

[30] C. Nvidia, “Compute unified device architecture programming guide,” 2007.

[31] A. Munshi et al., “The opencl specification,” Khronos OpenCL Working Group, vol. 1, 2009, pp. 11–15.

[32] M. Kruliš, Z. Falt, D. Bednárek, and J. Yaghob, “Task scheduling in hybrid cpu-gpu systems,” Informačné Technológie-Aplikácie a Teória, p. 17.

[33] NVIDIA, “Kepler GPU Architecture,” http://datalab.cs.pdx.edu/niagara/ [Online, retrieved: February, 2015].