

Ikra: Object-oriented GPGPU Programming in Ruby with CUDA

Research Paper

Matthias Springer Hidehiko Masuhara

Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, Japan

matthias.springer@acm.org masuhara@acm.org

Abstract

We present Ikra, an array-based parallel extension to Ruby with a just-in-time compiler. The high-level goal of Ikra is to allow developers to exploit GPU-based high-performance computing without having to be familiar with intricate details of the underlying GPU infrastructure and CUDA.

Ikra analyzes parallel sections in the form of array operations (e.g., `map`, `select`, or `each`), performs type inference upon invocation, and generates a CUDA kernel. To reduce thread divergence, Ikra reorders the base array based on the type information, which is useful for programs that were designed in an object-oriented way. Objects are represented as columns, giving rise to memory coalescing. Although programmers are advised to write statically-typed Ruby code for performance reasons, Ikra can generate CUDA code for expressions that can be one of multiple types using class tags.

Categories and Subject Descriptors D.1.3 [Concurrent Programming]: Parallel Programming

Keywords GPGPU, CUDA, Ruby, object-oriented programming

1. Introduction

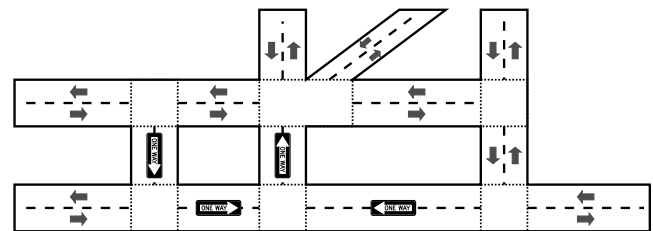
With the availability and affordability of powerful GPUs, general purpose computing on graphics processing units (GPGPU) is becoming more and more popular in high-performance computing. Nowadays, many supercomputers rely on GPUs as main processing units, because they allow for massively parallel execution of algorithms or simulations with thousands of threads per GPU. However, GPU programming differs from traditional CPU programming, mostly because of architectural differences.

The goal of the Ikra project is to make GPU programming available to researchers who are not familiar with the details of GPUs their programming languages. Ikra is a library for Ruby that translates parallel sections to CUDA code and executes them in parallel on GPUs. We target the Ruby programming language because it provides powerful mechanisms for embedding DSLs in the language, which will be useful for later experiments.

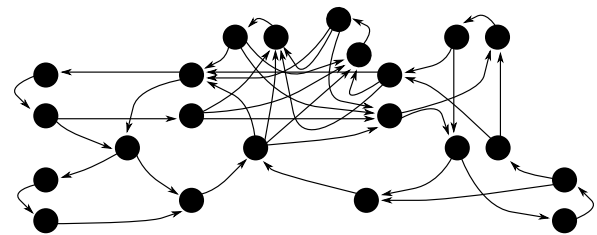
2. Example: Actor-based Traffic Simulation

A simple actor-based traffic simulation will serve as a running example in this paper. The basic idea is to simulate the behavior of a number of actors (e.g., cars, buses, pedestrians, etc.), given a street network as a directed graph (Figure 1) in adjacency list representation. Every actor is located on one street. Every street has a *length* attribute and every actor has a *progress* attribute representing the distance from the beginning of the street. Once these two attributes have the same value, the actor reached an intersection and should be moved to a different street (or make a U-turn if there is no other neighboring street).

A car moves at a constant speed of `@max_velocity`. A pedestrian moves at a random speed between $-0.5 \times \text{@max_velocity}$ and `@max_velocity`, i.e., a pedestrian can make negative progress. This is how we model strolling pedestrians. Furthermore, the progress of actors might be affected by weather conditions depending on their type. For example, cars slow down if the weather conditions are bad, whereas pedestrians are not affected by weather conditions.



(a) Actual street network (map)



(b) Street network as directed graph

Figure 1: Street Network for Traffic Simulation

Data Structure The street network and the actors are designed in an object-oriented way. Figure 2 shows the class organization of the traffic simulation. Car and Pedestrian are subclasses of Actor and provide their own move methods which will be invoked for every tick of the simulation.

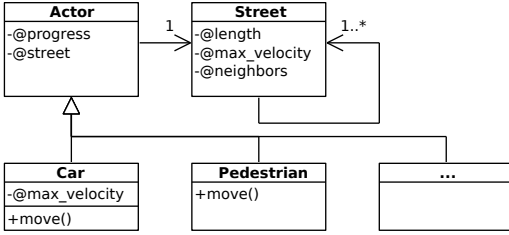


Figure 2: Class Diagram for Traffic Simulation

Main Simulation Loop The following code snippet contains the main simulation functionality. The method `pmap` designates a parallel section. Its parameter `ticks` determines how often the entire peach statement should be executed and is equivalent to wrapping the peach statement in a loop that executes it `ticks` times¹.

```

actors = [...]
ticks = 1000
weather = Weather::Rainy

actors.peach(ticks) do |actor|
  actor.move(weather)
end

```

3. Architecture

Ikra is a library for Ruby. It adds functionality to arrays to execute `map`, `select` and `each` operations in parallel. Programmers can *require* Ikra in Ruby files, upon which new parallel versions of array operations are available (e.g., `pmap`). These parallel array operations take a block as an argument and designate the only parts of a Ruby programs that are parallelized using Ikra. The default behavior is to spawn one thread per array element, which is why all these computations must be independent of each other. Every *tick* of the simulated progresses the current time by a certain constant value and actors are required to update their progress and street attributes accordingly.

3.1 Compilation Process

Figure 3 gives a high-level overview of Ikra’s compilation process. Upon invocation of a parallel section, Ikra acquires the source code of the parallel block, generates an abstract syntax tree (AST), and infers the type of all expressions. As a result, the type of every local and instance variable is known. In the best case, the type of an expression is monomorphic and primitive, but Ikra also supports arbitrary Ruby classes as types, as well as polymorphic types (see Section 4.3). The type inferer traverses invoked methods in a may-point-to fashion². Based on the type-annotated AST, Ikra generates CUDA kernel code and boilerplate code for kernel invocation, and compiles the CUDA code using the nVidia CUDA toolchain. The result is a shared library which is loaded via Ruby’s foreign function interface. Before kernel invocation, the base array along with all reachable objects (via instance variables) is transferred to the GPU’s global memory. After kernel invocation, all changed objects and the result of the parallel section (if applicable) are written back to Ruby.

3.2 Integration in Ruby

In contrast to some other projects, Ikra transforms Ruby code to CUDA code while the Ruby program is running (just-in-time

¹ Ikra does not support nested loops properly, which is why we suggest using this shortcut. See Section 7.3 for more details.

² Methods of all possible receiver types are taken into account.

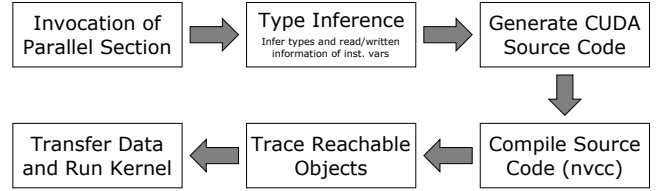


Figure 3: Overview of Ikra’s Architecture

compilation). Therefore, Ikra can determine the types of variables that are passed into parallel sections at runtime instead of doing a dataflow analysis of the entire program. This is not only faster but also more accurate in the light of reflection and metaprogramming, which is allowed outside of parallel sections but not inside them.

Two different kinds of variables can be used inside a parallel section: iterator variables and lexical variables. In the following code snippet, `el` is an iterator variable and `increment` is a lexical variable. The types of these variables are used as the foundation for type inference of the remaining parallel section.

```

increment = 10
[1, 2, 3].pmap do |el|
  el + increment
end

```

Programmers can use not only primitive objects (`Fixnum`, `Float`, etc.) but also objects which are instances of Ruby classes inside parallel sections, allowing for object-oriented modelling of the problem (e.g., a traffic simulation). Consequently, a graph of *reachable* (connected) objects must be transferred to the GPU. The *object tracer* is responsible for determining which objects should be copied to the GPU’s global memory (see Section 4.5).

After kernel execution, changed local variables and instance variables are copied back to the Ruby side (see Section 4.4).

4. Implementation and Optimizations

In this section, we give an overview of some interesting aspects of Ikra’s implementation.

4.1 Job Reordering

Before kernel invocation, Ikra analyzes all elements in the base array and reorders them according to their type. This is useful to avoid *thread divergence*, which can penalize performance when running programs on GPUs.

Thread Divergence In contrast to most CPU-based systems³, GPU-based systems are SIMD (single instruction, multiple data) systems. A GPU consists of a number of streaming multiprocessors. Such a processor has a single control unit that fetches and decodes instructions, but multiple arithmetic logic units (ALUs). Therefore, every instruction is executed in parallel on multiple chunks of data. Every ALU corresponds to one thread, but all threads that are executing on the same stream multiprocessor must follow the same control flow. In case two threads take a different branch, their execution is serialized until the control flow merges again. Consequently, jobs should be threads/jobs should be mapped to streaming multiprocessors in such a way that the control flow is unlikely to diverge among one such thread group (threads executing on one stream multiprocessor).

In CUDA, such a thread group is called *warp* and typically has a size of 32. There is no explicit interface to allocate threads to warps, but CUDA programmers try to write their programs in such a way

³ There are CPU extensions for SIMD computations, e.g., SSE.

that each consecutive group of 32 threads follows the same control flow.

Thread Allocation Ikra tries to avoid thread divergence by allocating jobs to warps automatically based on type information. Before kernel invocation, Ikra generates a *job reordering array*, such that the base array is sorted according the elements' types (Figure 4). Ikra does not actually change the order elements in the base array to ensure that other parts of the program outside of the parallel section are not affected.

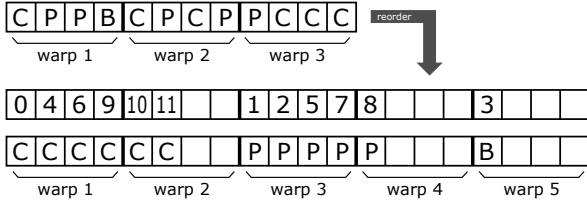


Figure 4: Example: Job Reordering. The first row is the original job order, the second row is the job reordering array, and the third row shows the resulting job order (warp size 4).

During job reordering the number of threads can increase as shown in Figure 4. Jobs are reordered in such a way that no two elements of different types are allocated in the same warp. If the number of jobs of a particular type is not a multiple of the warp size, the last warp will not be filled up entirely, but some threads will not have a job, i.e., they are not *no operation* threads. This might seem like a waste of computing power, but we expect the number of different types to be small (3 in this example).

The job reordering array can be computed in linear time by scanning all elements of the base array twice. The following pseudo code is similar to counting sort and bucket sort [5]. It generates one array of indices per type (class) and concatenates these arrays, making sure that every new array starts at a multiple of the warp size W .

Algorithm 1 Job Reordering

```

1: procedure REORDERINGARRAY(base, W)
2:   types  $\leftarrow$  Hash.new
3:   for all (el, idx)  $\in$  base do
4:     types[el.class].add(idx)
5:   end for
6:   result  $\leftarrow$  Array.new( $\sum_{arr \in \text{types.values}} \lceil |arr|/W \rceil * W$ )
7:   next  $\leftarrow$  0
8:   for all arr  $\in$  types.values do
9:     for all idx  $\in$  arr do
10:      result[next] = idx
11:      next  $\leftarrow$  next + 1
12:     end for
13:     next  $\leftarrow \lceil \text{next}/W \rceil * W$ 
14:   end for
15:   return result
16: end procedure

```

4.2 Columnar Object Layout

Objects are typically represented row-wise, i.e., every object is a contiguous chunk of data in the memory. Based on observations in previous work on GPU-powered database query execution [4] where a column-wise data organization proved to be superior compared to a traditional row-based data layout, Ikra stores objects in a columnar layout [9].

Memory Coalescing Global memory is one of the main bottlenecks of GPUs. One approach is to aim for memory access patterns where memory that is accessed in parallel by a number of threads is spatially local. Such memory accesses can be coalesced, i.e., the GPU can process such accesses in a single request, alleviating the global memory bottleneck.

Since a GPU is a SIMD system, all threads within a warp have to execute the same instruction at a time. Consequently, if one thread accesses an instance variable, then all other threads within the same warp access the same instance variable (or block because of thread divergence), probably in a different object. In this situation, a columnar object layout is superior to a row-based object layout, because parallel accesses to the same instance variable are more likely to be spatially local (Figure 5).

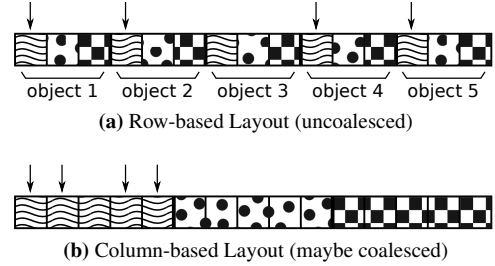


Figure 5: Example: Row-based and Column-based Object Layout. Boxes represent instance variables. Arrows indicate parallel access.

Generating Columns In the following, we present a first approach for representing object as columns. For the moment, we assume that all source code is statically typed and the types of all expressions and variables could be inferred successfully. This approach will be extended in the light of polymorphic expressions in the next section.

After running the object tracer, we know which objects should be transferred to the GPU. These objects are processed as follows.

1. Group objects by class c , resulting in arrays O_c .
2. Assign an ID to every object for all O_c , starting from 0 in every O_c , IDs being consecutive. This results in a hash map H_c mapping objects to class-specific IDs.
3. For every instance variable v of every class c , create an array (column) $A_{c,v}$ of size $m+1$, where m is the maximum ID in O_c . The base type of the array is the type of the instance variable if it is primitive, or `int` otherwise (referencing other non-primitive object via its ID).
4. Traverse the object graph. For every visited object o , determine its class c and ID $H_c[o]$. Store every instance variable v in the corresponding column slot $A_{c,v}[H_c[o]]$. If the instance variable is non-primitive, look up its ID and store it.

Note that after this transformation, the type of the base array that is passed to the kernel, is of type `int` and contains object IDs if it contains non-primitive objects.

Source Code Transformation Since objects are now represented as fields of arrays, Ikra must generate different source code for read from or writing to instance variables. We continue to assume that the type of all expressions and variables is known unambiguously. Moreover, we do not consider generating new objects at this time (see Section 7.2).

In the following, we consider reading/writing instance variables of an object and calling methods on an object, where the object is identified by its type c and its ID i . Whenever objects are passed around, Ikra actually generates source code that passes its ID around.

Passing type information is not necessary, because we assume that we know the type of every expression and variable⁴.

Reading an instance variable v of object o with type c and ID i translates to reading the column $A_{c,v}[i]$. Writing an instance variable translates to writing into the same column at the same position.

Every instance method is translated to a device function, where the type c is mangled into its name and the first parameter has type `int` and represents the ID of `self` object. Whenever Ikra encounters a method call during code translation, it determines the receiver's type and generates a call to the appropriate device function.

Representation of Arrays The previously-described mechanism for columnar objects works well with equally-sized objects, but not for variable-sized objects such as arrays. For example, the instance variable `@neighbors` of class `Street` is an array of streets.

Ikra effectively represents such $n : m$ relationships as join tables [6] that are *collapsed*. Such a table is sorted by object IDs for n . Furthermore, the n column is not stored as a full column but as RLE tuples [2] consisting of an implicit ID for n , a start offset into the m column, and a length value, distributed among multiple columns. RLE tuples are a well-known optimization in column databases.

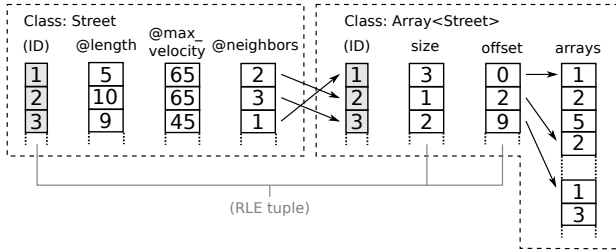


Figure 6: Example: Array Representation for `@neighbors` ($n : m$ relationship). (`Street.ID`, `Array.offset`, `Array.size`) is an RLE tuple [2].

4.3 Dynamically-typed Expressions

In contrast to CUDA, Ruby is a dynamically-typed programming language. One of the goals of the Ikra project is to allow programmers to write Ruby code in a *natural* way, i.e., programmers should be able to write the same source code that they would write in a standard Ruby environment. For this reason, Ikra should also support Ruby expression whose types cannot be inferred unambiguously at translation time.

Ikra embeds dynamic types into CUDA's static type system by generating explicit type dispatch statements at method call sites based on type tags [3] for receivers whose types cannot be inferred unambiguously. From a perspective of object-oriented design, this looks as if every object has a *type* instance variable.

Class Tags To support dynamically-typed expressions, we extend the idea of class-specific object IDs as follows. The following mechanism is applied only if the type of an expression or variable cannot be uniquely inferred. Otherwise, the previously described (non-extended) mechanism is applied.

Whenever an object ID was previously used, we now use a tuple of object ID and *class ID*. A class ID is a unique `int` identifier for a certain class. Class tags are used for object references and for method calls. A Ruby variable is now represented by two CUDA `int` variables: one for the class tag and one for the object ID.

⁴ We consider subclasses to be an entirely different type in this section.

Similarly, when passing an object as an argument, the same two arguments are passed in the generated CUDA code.

Before dispatching to a method, the generated CUDA code determines the type of the receiver in a switch statement to select the corresponding instance method. In the following example, two arrays (actor tags and actor IDs) are passed to the kernel function. The device function representing the block invokes the correct instance method.

```

__global__ void kernel(int *jobs, int ←
    *actor_tag, int *actor_id, int weather)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    block(actor_tag[jobs[tid]], actor_id[jobs[tid]], ←
        weather);
}

__device__ void block(int actor_tag, int ←
    actor_id, int weather)
{
    for (int i = 0; i <= ticks; i++)
    {
        switch (actor_tag)
        {
            case TAG_Car:
                method_Car_move(actor_id, weather);
                break;
            case TAG_Pedestrian:
                method_Pedestrian_move(actor_id, weather);
                break;
            default:
                /* same mechanism to dispatch to
                 method_missing */
        }
    }
}

```

Object Layout In the light of class tags, we extend the columnar object layout as follows. If Ikra cannot infer the type of an instance variable unambiguously, two columns are created for that instance variable: one for the class tag and one for the object ID.

Polymorphism In terms of type inference, subclasses are treated as entirely different types instead of subtypes. For example, if type inference determines that the type of a variable can be `Car` or `Pedestrian`, the type of that variable is the set $\{\text{Car}, \text{Pedestrian}\}$ instead of `Actor`. In this case, the type dispatch statements for a method call contains can handle only `Car` and `Pedestrian`, but not, for example, `Bus`, which is also a subclass of `Actor`. If a method is only implemented in a superclass (e.g., `Actor`), but not in the receiver's class itself (e.g., `Car`), the type dispatch statements can dispatch to that method directly, since that case can be detected at translation time. In addition to the object ID of the `self` (receiver) object, its class ID is also passed as an argument during method calls.

Instance variables are stored together with the first class in the superclass hierarchy, in which they were used for the first time. For example, `@street` is stored as a column of class `Actor`. Subclasses and superclass share the same IDs. If an object is an instance of a different subclass, the corresponding column values are *null* values [9]. For example, if object 15 is a `Pedestrian`, then `ACar,@max_velocity[15]` is *null*.

4.4 Read/write Analysis for Instance Variables

As part of type inference, Ikra analyzes if an instance variable is read and/or written. Only instance variables that are read or written

are transferred to the GPU. Only instance variables that are written are transferred back to the Ruby side directly after kernel invocation. Ikra performs a may-be-read/may-be-written analysis, which can have false positives in case the type of an expression cannot be determined accurately. Ikra does not copy single instance variable values, but entire columns.

4.5 Object Tracer

The object tracer is responsible for generating a list of object that must be transferred to the GPU before kernel invocation. It starts with a set of root object: all elements of the base array and lexical variables. Then, it traverses the object graph by following all instance variables. However, an instance variable value is visited only if that variable is read or written inside the parallel section. The entire process of tracing the object graph and assigning unique object IDs is similar to the *system tracer* component in Smalltalk [7].

5. Microbenchmarks

Kernel Execution Benchmark with/without job reordering and with/without columnar object layout (4 benchmarks)

Job Reordering Benchmark how long it takes to reorder jobs (2 dimensions: number of items, number of different types)

Object Tracer Benchmark how long it takes to trace a directed graph in adjacency list representation (2 dimensions: number of nodes, avg. vertex degree)

6. Related Work

7. Future Work

This section gives a brief overview of ideas for future work on Ikra.

7.1 Minimizing Data Transfers

Our current Ikra implementation transfers objects to the GPU's global memory every time a kernel is invoked. However, memory access is one of the main bottlenecks of GPUs and should be avoided. Future versions of Ikra will try to minimize data transfers by only transferring changed object during consecutive kernel invocations, even if two different kernels were invoked. Similarly, objects should only be transferred back to the Ruby side once they are actually accessed. One promising approach replaces instance variable accessor methods with code that retrieves the actual value from the GPU and caches it.

It is our vision that the parallel CUDA code is in full control of instance creation. The only reason for transferring data to the GPU should be cases where an object graph is loaded from an external source. For example, a researcher might want to load a street network of a real city from the file system. It is then not necessary to allocate this data structure both on the GPU and on Ruby side. The Ruby programs might, however, access certain objects and some of their instance variables for UI purposes or to display the result of a computation.

7.2 Data Modification

Ikra capabilities to modify data inside a parallel section are still limited, nevertheless sufficient for use cases like actor-based traffic simulations or OLAP applications, where data is mostly static.

For example, new objects can be created only on the Ruby side, but not inside parallel sections. This is because instance variable columns must be increased when adding new objects. However, increasing their size might require moving them to a different place in the global memory, which is an expensive operation that should be avoided.

As another example, it is currently not possible to add or remove elements from an array⁵. Future versions of Ikra might store arrays separately instead of using a single big array (Figure 6). Instead of storing an offset, this would require storing a pointer.

7.3 Nested Loops

Ikra does not yet support nested loops properly. Consider the main loop of the traffic simulation as an example. Putting a `ticks` loop inside the `peach` block works but contradicts intuition. In a sequential program, most programmers would formulate the simulation code as a series of simulation ticks, where every simulation tick iterates over all actors, as opposed to iterating of over all actors, where every actor is moved for a series of simulation ticks (see listing).

```
actors.peach do |actor|
  for i in 1..ticks
    actor.move(weather)
    synchronize
  end
end
```

The following code snippet is more intuitive, but would allocate one thread per tick instead of one thread per actor. However, the mechanism described in this paper takes advantage of allocating threads based on the actors' types.

```
(1..ticks).peach do
  actors.each do |actor|
    actor.move(weather)
  end
end
```

Nesting two parallel `peach` statements within each other is not supported at moment. Parallelization is allowed only on one level at the moment.

7.4 Performance Optimizations

Further ideas for performance optimizations include taking advantage of shared memory, which is much faster than global memory. However, it is not obvious what kind of data to store in shared memory because of its limited size.

Ikra's columnar object layout is similar to the data layout of column databases. Future work might investigate to what degree optimizations in the area of column databases [1; 8] are applicable to a column-based object graph. Data compression mechanisms for minimizing data transfer time look particularly promising, given the performance gap between global memory and shared memory, and have been subject to previous work in GPU computing [10; 11].

8. Conclusion

References

- [1] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.
- [2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 671–682, New York, NY, USA, 2006. ACM.
- [3] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 213–227, New York, NY, USA, 1989. ACM.

⁵ Changing an element is allowed.

- [4] Peter Bakkum and Kevin Skadron. Accelerating sql database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU-3*, pages 94–103, New York, NY, USA, 2010. ACM.
- [5] E. Corwin and A. Logar. Sorting in linear time - variations on the bucket sort. *J. Comput. Sci. Coll.*, 20(1):197–202, October 2004.
- [6] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.
- [7] Glenn Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
- [8] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. Fast in-memory SQL analytics on relationships between entities. *CoRR*, abs/1602.00033, 2016.
- [9] Toni Mattis, Johannes Henning, Patrick Rein, Robert Hirschfeld, and Malte Appeltauer. Columnar objects: Improving the performance of analytical applications. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 197–210, New York, NY, USA, 2015. ACM.
- [10] Ritesh A. Patel, Yao Zhang, Jason Mak, and John D. Owens. Parallel lossless data compression on the GPU. In *Proceedings of Innovative Parallel Computing (InPar '12)*, May 2012.
- [11] Piotr Przymus and Krzysztof Kaczmarek. *On the Move to Meaningful Internet Systems: OTM 2012 Workshops: Confederated International Workshops: OTM Academy, Industry Case Studies Program, EI2N, INBAST, META4eS, OnToContent, ORM, SeDeS, SINCOM, and SOMOCO 2012, Rome, Italy, September 10-14, 2012. Proceedings*, chapter Improving Efficiency of Data Intensive Applications on GPU Using Lightweight Compression, pages 3–12. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.