Submitted for review to ARRAY 2016

Object Support in an Array-based GPGPU Extension for Ruby

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

This paper presents implementation and optimization techniques to support objects in Ikra, an array-based parallel extension to Ruby with dynamic compilation. The high-level goal of Ikra is to allow developers to exploit GPU-based high-performance computing without paying much attention to intricate details of the underlying GPU infrastructure and CUDA.

Ikra supports dynamically-typed object-oriented programming in Ruby and performs a number of optimizations. It supports parallel operations (e.g., map, each) on arrays of polymorphic objects, allowing polymorphic method calls inside a kernel by compiling them to conditional branches. To reduce branch divergence, Ikra shuffles thread assignments to base array elements based on runtime types of elements. To facilitate memory coalescing, Ikra stores objects in a fields-of-arrays representation (columnar object layout). To eliminate intermediate data in global memory, Ikra merges cascaded parallel sections into one kernel using symbolic execution.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming; D.3.4 [*Processors*]: Code generation, Compilers

Keywords GPGPU, CUDA, Ruby, object-oriented programming

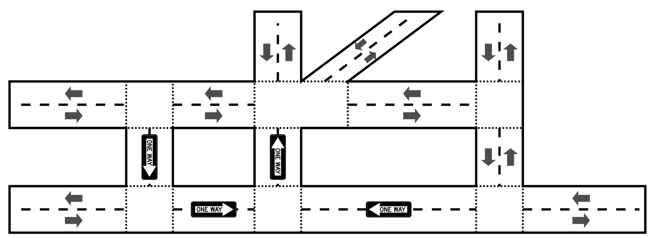
1. Introduction

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.

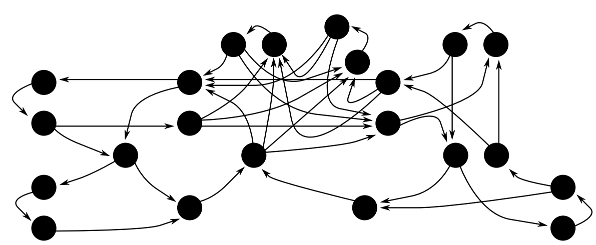
The goal of the Ikra project is to make GPU programming available to developers who are not familiar with the details of GPUs and their programming languages. Ikra is a library for Ruby that translates parallel sections to CUDA code and executes them in parallel on GPUs. It extends our previous work [12] with a dynamic compilation approach to allow for a larger number of optimizations and tighter integration with Ruby. We target the Ruby programming language because it provides powerful mechanisms for embedding DSLs in the language, which will be useful for future work.

2. Example: Agent-based Traffic Simulation

A simple object-oriented agent-based traffic simulation will serve as a running example in this paper. The basic idea is to simulate the behavior of a number of agents [8] (e.g., cars, buses, pedestrians, etc.), given a street network as a directed graph (Figure 1) in adjacency list representation. Every agent is located on one street. Every street has a *length* attribute and every agent has a *progress* attribute representing the distance from the beginning of the street. Once these two attributes have the same value, the agent reached an intersection and should be moved to a different street (or make a U-turn if there is no other neighboring street).



(a) Actual street network (map)



(b) Street network as directed graph

Figure 1: Example: Street Network for Traffic Simulation

A car moves at a constant speed of $\min(M_c, M_s)$, where M_c is the maximum velocity of the car and M_s is the maximum speed allowed on the current street. A pedestrian moves at a random speed between -2 mph and 4 mph, i.e., a pedestrian can make negative progress. This is how we model strolling pedestrians. Furthermore, depending on their type, the progress of agents might be affected by weather conditions. For example, cars slow down if the weather conditions are bad, whereas pedestrians are not affected by weather.

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

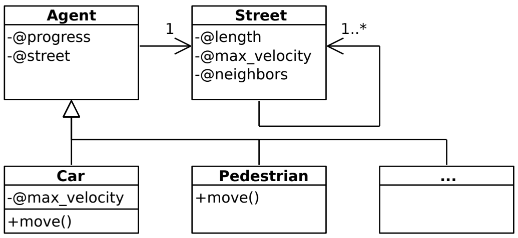


Figure 2: Class Diagram for Traffic Simulation

Main Simulation Loop The following code snippet contains the main simulation functionality. The method `peach` 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` times[1].

```
agents = # load scenario from file system
ticks = 1000
weather = Weather::Rainy

agents.peach(ticks) do |agent|
   agent.move(weather)
end
```

Every *tick* of the simulation progresses the current time by a certain constant value and agents are required to update their progress and street attributes accordingly.

3. Architecture

Ikra is a library for Ruby. It adds functionality to arrays to execute `map`, `reduce`, `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, designate the only parts of a Ruby programs that are parallelized using Ikra.

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 4.4). The type inferer traverses invoked method bodies for all possible receiver types (*union type*). Based on the type-annotated AST, Ikra generates CUDA kernel code and initialization 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 and lexical variables along with all reachable objects (via instance variables) are 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.

[1] There are currently certain limitations for loop nesting. See Section 7.3 for more details.

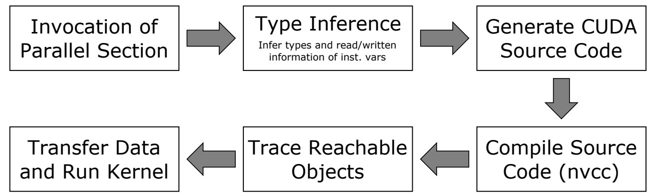


Figure 3: Overview of Ikra's Architecture

3.2 Integration in Ruby

Ikra transforms Ruby code to CUDA code at runtime when a parallel operation is invoked (dynamic compilation). Therefore, Ikra can determine the types of elements/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 robust 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[2]. In the main loop of the traffic simulation example, `agent` is an iterator variable and `weather` is a lexical variable. The types of these variables are used as the basis for type inference of the remaining parallel section.

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 modeling 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 4.6).

After kernel execution, changed local variables and instance variables are copied back to the Ruby side (see Section 4.5).

4. Implementation and Optimizations

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

4.1 Symbolic Execution

Parallel array operations (except for `peach`) are executed symbolically in Ikra. They can be cascaded and are executed only if the result is actually accessed. The default behavior is to spawn one thread per array element, which is why all these computations must be independent of each other.

Ikra performs *kernel fusion* [20; 19], i.e., cascaded parallel operations are merged into a single CUDA kernel to avoid writing intermediate results to the global memory. Instead, they can be kept in registers. The process of merging two parallel blocks is not relevant in the scope of this paper and omitted.

4.2 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 *branch divergence*, which can penalize performance when running programs on GPUs.

Branch Divergence In contrast to most CPU-based systems[3], 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,

[2] Instance variables can be used when calling an instance method.

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

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 streaming 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 divergence among one such thread group (threads executing on one streaming multiprocessor).

In CUDA, such a thread group is called *warp* and 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 that each consecutive group of 32 threads follows the same control flow.

Thread Allocation Ikra tries to avoid branch 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.

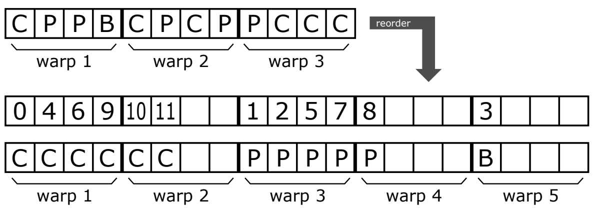


Figure 4: Example: Job Reordering. The first row is the original job order, the second row is the job reordering array, and the third row shows the resulting job order (warp size 4).

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 pseudo code in Algorithm 1 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.

4.3 Columnar Object Layout

In traditional programming languages and virtual machines, objects are typically represented row-wise, i.e., every object is a contiguous chunk of data in the memory. However, it is common practice in GPU programming to work with multiple arrays of structure fields instead of one array of structures.

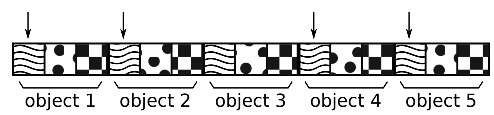
Memory Coalescing Global memory is one of the main bottlenecks of GPUs. One approach is to aim for memory access patterns where memory that is accessed in parallel by a number of threads is spatially local. Such memory accesses can be coalesced, i.e., the GPU can process such accesses in a single request,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

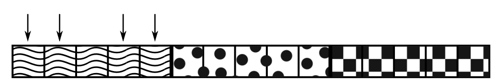
Algorithm 1 Job Reordering

procedure REORDERINGARRAY(base, W)
    $types \leftarrow$ Hash.new   *default value:* Array.new
    for all $(el, idx) \in base$ **do**
        $types[el.\text{class}].\text{add}(idx)$
    end for
    $result \leftarrow \text{Array.new}(\sum_{arr \in types.values} \lceil |arr|/W \rceil * W)$
    $next \leftarrow 0$
    for all $arr \in types.\text{values}$ **do**
        for all $idx \in arr$ **do**
            $result[next] = idx$
            $next \leftarrow next + 1$
        end for
        $next \leftarrow \lceil next/W \rceil * W$
    end for
    return $result$
end procedure

thread accesses an instance variable, then all other threads within the same warp access the same instance variable (or block because of branch 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).



(a) Row-based Layout (uncoalesced)



(b) Column-based Layout (maybe coalesced)

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 objects as columns. For the moment, we assume that all source code is statically typed and the types of all expressions and variables were 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 their classes 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 objects via their IDs).
4. For every object o with 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 base type of the base array that is passed to the kernel, is `int` and contains object IDs if it contains non-primitive objects.

In our implementation, the object tracer is combined with the column generator. A pseudo code implementation which was also used for benchmarks is shown in Section A.

Source Code Transformation Since objects are now represented as fields of arrays, Ikra must generate different source code for reading 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 their IDs around. Passing type information is not necessary, because we assume that we know the type of every expression and variable[4].

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 slot.

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.

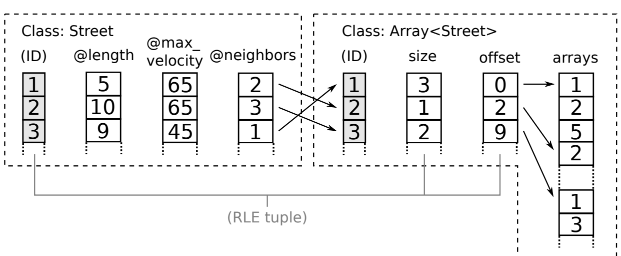


Figure 6: Example: Array Representation for `@neighbors` ($n : m$ relationship). (`Street.ID`, `Array.offset`, `Array.size`) is an RLE tuple [2].

From an implementation point of view, an array is an ordinary object with an offset and a size attribute. The offset attribute points into a single large array containing the contents of all arrays. This layout might change in future versions of Ikra (see Section 7.2).

4.4 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.

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

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 tag*. A class tag 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. 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 (agent tags and agent IDs) are passed to the kernel function. The device function representing the block invokes the correct instance method.

```
__global__ void kernel(int *jobs, int ↩
    *agent_tag, int *agent_id, int weather)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  block(agent_tag[jobs[id]], agent_id[jobs[id]], ↩
      weather);
}

__device__ void block(int agent_tag, int ↩
    agent_id, int weather)
{
  for (int i = 0; i <= ticks; i++)
  {
    switch (agent_tag)
    {
      case TAG_Car:
        method_Car_move(agent_id, weather);
        break;
      case TAG_Pedestrian:
        method_Pedestrian_move(agent_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 {`Car`, `Pedestrian`} instead of `Agent`. 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 `Agent`. If a method is only implemented in a superclass (e.g., `Agent`), but not in the receiver's class itself (e.g., `Car`), the generated code can dispatch to that method directly; that case can be detected at translation time. In addition to the object ID of the `self` (receiver) object, its class tag 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 `Agent`. Subclasses and superclass share the same IDs. If an object is an instance of a different subclass, the corresponding column values are *null* values [13]. For example, if object 15 is a `Pedestrian`, then $A_{\texttt{Car,@max\_velocity}}[15]$ is null.

4.5 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.

4.6 Object Tracer

The object tracer is responsible for generating a list of objects that must be transferred to the GPU before kernel invocation. It starts with a set of root objects: 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 [10].

5. Microbenchmarks

To evaluate our optimizations, we conducted a series of microbenchmarks using the traffic simulation example[5]. We ran benchmarks on the Tsubame supercomputer[6] on a *thin* compute node with two Intel Xeon X5670 CPUs (2.93 GHz × 6 cores each), 54 GB RAM, and an nVidia Tesla K20Xm GPU, running Linux `3.0.76-0.11-default x86_64`, CUDA 7.0.27, and Ruby 1.9.3p448.

We simulated 1,000,000 iterations of a random street network with 500 streets and random vertex out-degrees between 1 and 10, 4096 cars, and 16384 pedestrians. All benchmark running times are average values of 5 runs using the same random scenario.

Kernel Execution Figure 7 shows the kernel running time of the traffic simulation in various configurations. Figure 7a shows the running time with a columnar object layout, which is around 30% faster than the row-based layout in Figure 7b.

The objects involved in this example are quite small. Agents are represented by 32-byte structs (three instance variables and class tag) in row-based layout, or three 4-byte columns (class tag is passed as an argument) in a columnar layout, respectively. The GPU's L1 cache is 48 KB, with a cache line size of 128 bytes. To analyze the effect of caching, we ran the row-based benchmark with artificially enlarged object sizes (10/5 additional instance variables for agents/streets, resulting in an object size of 72 bytes/32 bytes, respectively; Figure 7c). This configuration is interesting because the example code accesses all instance variables of an agent subsequently, diminishing the advantage of a column-based layout, because the entire object (and three subsequent objects) can be held in cache (prefetching). A columnar object layout is around 60% faster compared to this configuration.

[5] `git@github.com:matthias-springer/array2016-paper.git`

[6] `http://tsubame.gsic.titech.ac.jp/`

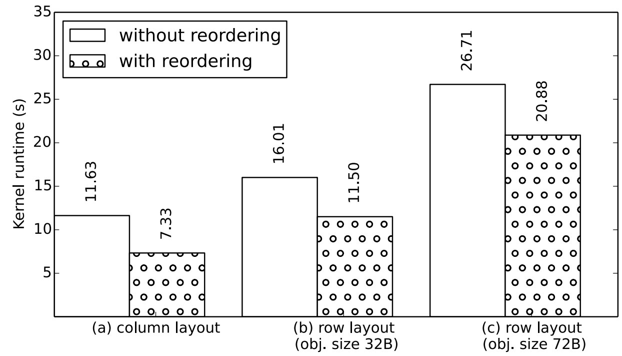


Figure 7: Kernel Running Time for Traffic Simulation (CUDA)

Job Reordering Figure 8 shows the running time for generating the job reordering array (warp size 32). This is currently done in the Ruby interpreter but could be moved to the GPU side in future versions. The running time increases linearly with the number of elements in the base array. Changing the number of types (classes) has only a small effect on the running time. We assume that this number is much smaller than the number of elements. For the traffic simulation example, the running time for generating the job reordering array is neglectable.

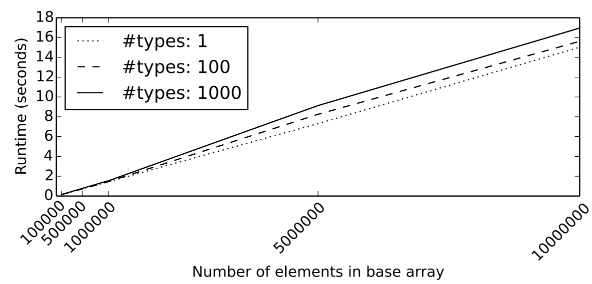


Figure 8: Running Time for Generating Job Reordering Array (Ruby)

Tracing Object and Generating Columns Figure 9 shows the running time for tracing objects and generating columns (Algorithm 2) for the traffic simulation example. For the number of agents/streets that were used in the kernel benchmarks, the running time is neglectable. Moreover, in future plans for Ikra we want to perform this step only once and reuse data that was already processed and moved to the GPU earlier (see Section 7.1).

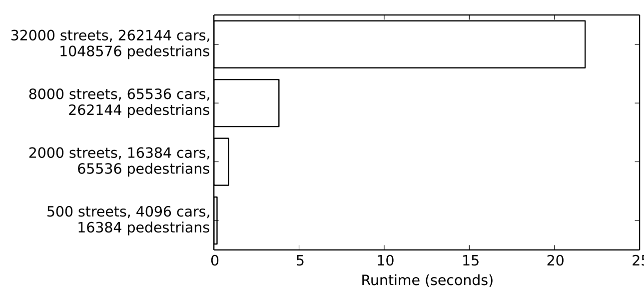


Figure 9: Running Time for Tracing Objects and Generating Columns (Ruby)

The running time for transferring columns to the GPU and generating the CUDA code is not included in these benchmarks.

6. Related Work

Columnar data layouts are known to be superior compared to row-based data layouts for certain kinds of database queries (e.g., OLAP queries) [16], and especially for GPU-powered databases [4]. In fact, one of the benefits of column stores for CPU-based database systems is *prefetching*, which is similar to coalescing on GPUs, but without the parallel aspect. Columnar data layouts have also been evaluated for object-oriented programming languages. Mattis et al. have implemented a columnar object layout in Pypy to increase the performance of analytical queries [13]. Ikra essentially uses the same columnar object layout, but extended to polymorphic types.

A number of different techniques exist for avoiding branch divergence, e.g., detecting and delaying divergent branches at runtime in order to execute them at a later time, or factoring out instructions that are common to two (divergent) branches [7]. A different approach is to reorder jobs, either with a reordering array (which is what Ikra does) or by physically changing the order of the jobs in the base array. Both techniques can be combined to increase memory coalescing [22] (physically reordering data, then using a reordering array to restore the original semantics), but detailed knowledge about memory access patterns is required. Previous work has also investigated how the overhead of these transformations can be hidden using a CPU-GPU pipelining scheme [21], if done at runtime in order to react to changes.

Ishizaki et al. presented a framework for executing lambda expression used with the parallel streams API in Java 8 programs on nVidia GPUs [9]. Their approach is to generate LLVM intermediate code (IR) from Java bytecode, which is in our opinion superior to Ikra's approach of performing a Ruby-to-CUDA source-code-to-source-code transformation from an engineering point of view. Future versions of Ikra might generate LLVM IR code from YARV bytecode [18]. Further optimizations of their Java 8 compiler include a check for array aliasing (which is what Ikra's object tracer does implicitly) and utilizing a read-only cache. Their implementation supports virtual method calls using direct devirtualization if the receiver type can be uniquely determined at compile time, or guarded devirtualization, executing an iteration on the GPU if the guard fails; in either case, the GPU code must only be able to handle monomorphic method calls. Ikra's code generator applies direct devirtualization, i.e., it does not generate type dispatch statements if the receiver type can be inferred unambiguously. Otherwise, it generates CUDA code that dispatches to the correct method based on class tags.

Firepile is a Scala-to-CUDA compiler [14]. It supports Scala classes and generates a struct definition per class. The first field has as type the struct type for the superclass and is needed for inherited instance variables. The struct for `Object` contains a class tag used for method dispatch. Ikra passes and stores class tags together with object IDs. If class tags were stored in one column shared by all objects, then all objects (and types) would have to follow the same numbering scheme, which would lead to sparse columns and a waste of global memory. The same problem occurs already in a less severe form in the light of subclassing (see Section 4.4, *null* values).

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 objects 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 approach replaces instance variable accessors 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 or must be swapped from/to the main memory. 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 program 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's capabilities to modify data inside a parallel section are still limited, nevertheless sufficient for use cases like agent-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 expensive.

As another example, it is currently not possible to add or remove elements from an array[7]. 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 agents, as opposed to iterating of over all agents, where every agent is moved for a series of simulation ticks (see listing).

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

The following code snippet is more intuitive, but would allocate one thread per tick instead of one thread per agent[8]. However, the mechanism described in this paper takes advantage of allocating threads based on the agents' types.

```
(1..ticks).peach do
  agents.each do |agent|
    agent.move(weather)
  end
end
```

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

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

[7] Changing an element is allowed.

[8] Wrapping `peach` inside an `each` block leads to unnecessary data transfers.

Algorithm 2 Object Tracing and Column Generation

```
procedure TRACEANDCOLUMNIZE(base, lexical, IV)
    Q ← Queue.new(base ∪ lexical)
    H ← Hash.new                          default value: Hash.new
    M ← Hash.new                                 default value: 0
    while |Q| > 0 do
        next ← Q.pop()
        c ← next.class
        if ¬c.isPrimitive ∧ next ∉ H[c].keys then
            H[next] ← M[c]
            M[c] ← M[c] + 1
            if c == Array then
                for all e ∈ next do
                    Q.push(e)
                end for
            else
                for all v ∈ IV[c] do
                    Q.push(next[v])
                end for
            end if
        end if
    end while
    for all c ∈ H.keys do
        for all v ∈ IV[c] do
            A_{c,v} ← Array.new(M[c])        (Array case omitted)
        end for
        for all (obj, id) ∈ H[c] do
            for all v ∈ IV[c] do
                if obj[v].class.isPrimitive then
                    A_{c,v}[id] ← obj[v]
                else if obj[v].class == Array then
                    A_{c,size}[id] ← |obj|
                    A_{c,ofs}[id] ← A_{c,ofs}[id−1]+A_{c,size}[id−1]
                    (copy over array into slots [ofs; ofs + size[)
                else
                    A_{c,v}[id] ← H[obj[v].class][obj[v]]
                end if
            end for
        end for
    end for
end procedure
```

optimizations in the area of column databases [1; 11] 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 [15; 17].

8. Summary

We presented Ikra, a dynamic Ruby-to-CUDA compiler for array-based parallel operations. Ikra allows programmers to write source code in an object-oriented way and applies optimizations to reduce branch divergence and to increase memory coalescing. Type information is used to reorder objects in the base array, making sure that only objects of the same type are executed in a warp. A columnar object layout is beneficial for memory coalescing, because threads inside a warp are executed in a SIMD manner. Future work will focus on additional performance optimizations and take into account a broader set of examples and benchmarks.

A. Appendix: Object Tracer

In the following pseudo code for tracing objects and generating columns, $base$ and $lexical$ are object tracer roots, and IV is a map of read/written instance variables per class (type inference).

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] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon 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.

[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] Edward Corwin and Antonette 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] Tianyi David Han and Tarek S. Abdelrahman. Reducing branch divergence in GPU programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1–3:8, New York, NY, USA, 2011. ACM.

[8] Dirk Helbing. *Social Self-Organization: Agent-Based Simulations and Experiments to Study Emergent Social Behavior*, chapter Agent-Based Modeling, pages 25–70. Springer Berlin Heidelberg, 2012.

[9] Kazuaki Ishizaki, Akihiro Hayashi, Gita Koblents, and Vivek Sarkar. Compiling and optimizing Java 8 programs for GPU execution. In *24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2015.

[10] Glenn Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[11] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. Fast in-memory SQL analytics on relationships between entities. *CoRR*, abs/1602.00033, 2016.

[12] Hidehiko Masuhara and Yusuke Nishiguchi. A data-parallel extension to ruby for GPGPU: Toward a framework for implementing domain-specific optimizations. In *Proceedings of the 9th ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, RAM-SE '12, pages 3–6, New York, NY, USA, 2012. ACM.

[13] 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.

[14] Nathaniel Nystrom, Derek White, and Kishen Das. Firepile: Runtime compilation for GPUs in Scala. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, GPCE '11, pages 107–116, New York, NY, USA, 2011. ACM.

[15] 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.

[16] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 1–2, New York, NY, USA, 2009. ACM.

[17] Piotr Przymus and Krzysztof Kaczmarski. *On the Move to Meaningful Internet Systems 2012 Workshops: OTM Academy, Industry Case Studies Program, EI2N, INBAST, META4eS, OnToContent, ORM, SeDeS, SINCOM, and SOMOCO 2012.Proceedings*, chapter Improving Efficiency of Data Intensive Applications on GPU Using Lightweight Compression, pages 3–12. Springer Berlin Heidelberg, 2012.

[18] Koichi Sasada. YARV: Yet Another RubyVM: Innovating the ruby interpreter. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 158–159, New York, NY, USA, 2005. ACM.

[19] Mohamed Wahib and Naoya Maruyama. Scalable kernel fusion for memory-bound GPU applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 191–202, Piscataway, NJ, USA, 2014. IEEE Press.

[20] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 107–118, Washington, DC, USA, 2012. IEEE Computer Society.

[21] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng Shen. Streamlining GPU applications on the fly: Thread divergence elimination through runtime thread-data remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 115–126, New York, NY, USA, 2010. ACM.

[22] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 369–380, New York, NY, USA, 2011. ACM.