A Data-Parallel Extension to Ruby for GPGPU

Toward a Framework for Implementing Domain-Specific Optimizations

Hidehiko Masuhara
Graduate School of Arts and Sciences
University of Tokyo
JST/CREST
masuhara@acm.org

Yusuke Nishiguchi
Graduate School of Arts and Sciences
University of Tokyo
nishiguchi@graco.c.u-tokyo.ac.jp

ABSTRACT

We propose Ikra, a data-parallel extension to Ruby for general-purpose computing on graphical processing unit (GPGPU). Our approach is to provide a special array class with higher-order methods for describing computation on a GPU. With a static type inference system that identifies code fragments that shall be executed on a GPU and with a skeleton-based compiler that generates CUDA code, we aim at separating application logic and parallelization and optimizations. The paper presents the design of Ikra and an overview of its implementation along with preliminary performance evaluation.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*

General Terms

Languages

Keywords

Data-parallel programming, Ruby, general-purpose computing on graphical processing unit (GPGPU), type inference

1. INTRODUCTION

General-purpose computing on graphic processing unit (GPGPU) is one of the promising and cost-effective ways to achieve high-performance computing on various platforms, from game consoles to supercomputers. However, the state-of-the-art GPGPU programs have tightly-coupled descriptions of application logic and parallelization concerns due to low-level language abstractions provided by GPGPU languages/frameworks. In addition, fast paced evolution of hardware architectures makes the tight-coupling problem more serious as the programs need to employ different parallelization strategies and optimization techniques so as to achieve reasonable performance on a new hardware.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
RAM-SE'12, June 13, 2012, Beijing, China
Copyright 2012 ACM 978-1-4503-1277-6/12/06 ...$10.00.

To this tight coupling problem, we advocate an approach to providing a programming language with a high-level abstraction for GPGPU, where an application program is focused on its own logic. By applying different parallelization and optimization strategies, the programmer can obtain reasonable performance on different execution platforms.

Based on this idea, we are developing a data-parallel extension to Ruby, called Ikra[1], which serves as a language for describing application logic without much concerning about parallelization and optimizations. With our planned open framework for parallelization, the programmer would be able to quickly prototype application logics and explore different optimization strategies.

This paper focuses on the data-parallel abstraction in Ikra, while leaving the open parallelization framework future work. Even though we assume a GPU as an underlying hardware architecture, the discussion in the paper is mostly applicable to other massively parallel and heterogeneous architectures. Section 2 introduces the main challenges in the state-of-the-art of GPGPU programming. Section 3 briefly sketches the language design of Ikra through an example application program. Section 4 gives our current implementation strategy, followed by preliminary performance evaluation in section 5. After discussed related work in section 6, we conclude the paper in section 7.

2. CHALLENGES IN GPGPU PROGRAMMING

The state-of-the-art of GPGPU programming requires the programmer to be aware of low-level parallelization and optimization concerns when writing application logic. This is not only due to the mainstream programming languages for GPGPU, such as CUDA and OpenCL, but also due to the nature of heterogeneity in GPGPU[2].

GPGPU programming requires the programmer to choose right *kernels*—portions of an application program that will be parallely executed on a GPU—as a GPU serves as a single-instruction-multiple-data (SIMD) accelerator with hundreds of simple cores.

It is however not easy to determine the kernels because one GPU core is much lower than CPU's. Sometimes has to reform the control structure and data layout to extract such a portion.

[1]Ikra is a Russian word that means roe of the salmon, which is also known as "rubies of the sea" in Japan.
[2]Though we here assume the nVidia's Fermi architecture, most of the discussion here is valid with other GPU or heterogeneous hardware accelerators.

In addition, tightly limited resources on a GPU demands the programmer to precisely control the number and size of threads, the data allocations, and so forth.

Memory Management.

A GPU has a deep memory hierarchy separated from the CPU's main memory. For example, an nVidia's Fermi-based GPU has a large "global" memory accessible from all GPU cores (yet separated from CPU's address space), a small "shared" memory shared by a group of threads, and a private "local" memory per-thread.

A challenge with respect to the memory model is that the programmer has to manually control memory usage at each layer in the hierarchy. For example, instead of relying on the cache memory, the programmer has to manually fetch frequently accessed data into each core's shared memory. High latency and low-bandwidth between CPU and GPU memories also requires to overlap computation and memory transfer.

Choosing Kernels.

The programmer has to choose kernels, i.e., the portion of code to be execution on a GPU. Though this is a common challenge in parallel computing, it is particularly important in GPGPU as its granularity sensitive performance model. For example, when an application has doubly-nested loops, parallelizing either loop would result in different performance depending on the number of threads, and the size of a working set in a kernel, which is constrained by the amount of private memory. As it is difficult to predict the effect of this mutual dependency, the programmer often had to implement and compare different parallelization strategies.

Domain-Specific Optimizations.

Many optimization techniques for GPGPU are domain specific; i.e., specific to an application domain, or specific to a particular hardware architecture. As mentioned above, an ideal choice of a kernel is dependent on the size of a working set, which varies over different applications. In addition, GPU architectures are still evolving at a remarkable pace, which will change many architectural parameters such as local memory size.

3. IKRA: RUBY IMPLEMENTATION WITH DATA-PARALLEL ABSTRACTION

3.1 Design Goals

To tackle the above challenges, we propose Ikra, a data-parallel extension to Ruby for GPGPU. Its design goals can be summarized as the following three points.

Integration with Ruby: Ikra is an extension to Ruby; i.e., a sequential portion of a program runs as a standard Ruby program. This will help quick prototyping of a GPGPU application by exploiting various class libraries in Ruby, such as string manipulation, networking, and graphics.

Parallelization through a data-parallel array class: Ikra's primary abstraction for parallelization is a data-parallel array class, called `PArray`, whose `map` and `inject`[3] methods are executed on a GPU in parallel. Since those methods are mostly compatible[4] with the same methods of the standard array class in Ruby, the programmer can easily try different parallelization strategies by simply replacing an array class with `PArray`.

Separated optimization: Though it is beyond the scope of the paper and much is left for future work, Ikra will provide a framework for applying different optimization strategies separately from application logic. The Ikra's parallel abstractions are designed at a high-level so as to make it easy to apply optimization techniques.

```
require 'ikra'                    # provides PArray      1
require 'RMagick'                 # external library     2
                                                         3
LIMIT = 64; INF   = 10                                   4
                                                         5
# read parameters                                        6
res   = ARGV[0].to_f; r_min = ARGV[1].to_f               7
r_max = ARGV[2].to_f; i_min = ARGV[3].to_f               8
i_max = ARGV[4].to_f                                     9
r_size = ((r_max-r_min) / res).to_i                      10
i_size = ((i_max-i_min) / res).to_i                      11
                                                         12
def sq(x)      # a function called from the kernel       13
 x*x                                                     14
end                                                      15
                                                         16
m = PArray.new(r_size, i_size){ |r_idx, c_idx|           17
  # the kernel code executed on a GPU                    18
  cr = r_min + res*r_idx; ci = i_min + res*c_idx         19
  iter = 0; zr = 0.0; zi = 0.0                           20
  while(iter < LIMIT && sqrt(sq(zr)+sq(zi)) < INF)       21
    zr1 = zr*zr-zi*zi+cr; zi1 = zr*zi+zi*zr+ci           22
    zr = zr1; zi = zi1                                   23
    iter += 1                                            24
  end                                                    25
  iter                                                   26
}                                                        27
                                                         28
# visualization with an external library                 29
f = Image.new(r_size,i_size)                             30
m.map_with_index{ |row, col, iter|                       31
  f.store_pixels(row,col,1,1,                            32
    [Pixel.from_HSL([iter/64.0,1.0,0.5])])               33
}                                                        34
f.display                                                35
```

Figure 1: A Ruby program that computes Mandelbrot set with `PArray`.

3.2 Ikra by Example

Due to space limitations, we introduce key characteristics of Ikra by using a simple application program that computes a Mandelbrot set.

Integration with Ruby.

As shown in figure 1, the entire program can be seen as a Ruby program using the `PArray` class, whose interface is provided by requesting the `ikra` package (line 1). We also offer a pure Ruby implementation of `PArray` so that the programmer can test and debug behavior of a program in an environment without a GPU.

[3] A Ruby equivalent of the fold function in MLs and Haskell and the reduce function in Lisp.

[4] As long as there is no dependency among iterations.

The sequential part of the program can use any Ruby features including external libraries. In figure 1, the program except for the lines from 13 until 27 is sequential; hence it uses string manipulations and an image processing library for visualization.

The `PArray` class (appearing at line 17) serves as the data-parallel abstraction in Ikra. A `PArray` object is allocated on the GPU's memory. Operations on a `PArray` object are executed in parallel on a GPU when possible. However, its API is mostly compatible with the standard array class in Ruby.

`PArray` provides a few methods for parallel operations, including namely initialization (`new`), `map` and `inject`, which execute the body of their block argument in parallel. In other words, the kernels are written as block arguments to `PArray` methods in Ikra. The initialization of `PArray` at line 17 executes the body of its block argument (until line 27) in parallel, and initialize elements of `PArray` with the computed values.

The kernel code is written in a subset of Ruby, which does not require explicit type declarations. Supported data types are integers and floating point numbers.[5] Supported control structures are `if`, `for` and `while`, and non-recursive global functions. The block body can reference variables declared in its lexical scope; it however does not support destructive assignments to lexical variables.

When a program calls a parallel operation on a `PArray` with a block body that performs unsupported operations (e.g., using objects or calling native libraries), Ikra transfers the contents of the array to CPU's memory and sequentially performs the operation on a CPU. For example, the `map` operation[6] at line 31 is executed on a CPU as its body argument manipulates an `Image` object and the `Pixel` class. Some other operations, such as `length`, are merely executed on a CPU without data transfer.

4. IMPLEMENTATION

Our current implementation performs offline (i.e., ahead-of-time) type-inference and compilation, and generates Ruby and CUDA programs by using skeleton-based parallelization.

4.1 Offline Processing

Our implementation is based on an offline processing; i.e., generates GPU-executable code before starting an Ikra program. This is mainly for the sake of simplicity of implementation, where wecan rely the CUDA compiler as its back-end. Runtime code generation, i.e., generates GPU-executable code at the beginning of the kernel, would be able to support wider range of Ruby programs as we can access concrete runtime type information. We however left this for future work.

4.2 Type Inference

Given an Ikra program, the implementation first infers types of all expressions, variables and methods in the program. The inferred types are used (1) for determining the operations on `PArray` objects, (2) for judging if the body of a block argument to a `PArray` operation (e.g., `map` and `inject`) is executable on a GPU, and (3) for generating a CUDA kernel function from the body of the block argument.

We currently use an inference system that only supports a few built-in types (i.e., integer, float, Boolean, and array) and control structure (i.e., if, while, and top-level methods). Variables, expressions, and elements of an array are monomorphic, while top-level methods can be polymorphic by duplicating code per call site. An object in a class except for the built-in types is regarded as of "dynamic" type. Exceptionally, there are typing rules for special operations (e.g., `to_i`, which converts any object into an integer).

Such a simple inference system is sufficient for Ikra as we need type information only around the kernel code, which is eventually translated into functions in CUDA, which has no support for objects. When our type inference system analyzes the program in figure 1, the variables created at lines 4–11 have either integer or float types thanks to the special rules. The following method and expression at lines 13–27 are fully typed, and determined as kernel code. Most of the expressions in the bottom section (from line 29) are typed as dynamic, as they use externally defined classes, namely `Image` and `Pixel`.

4.3 Skeleton-Based Code Generation

After the type inference, our implementation generates (1) a CUDA kernel function with a wrapper from each block argument to parallel operations, namely `new`, `map` and `inject`, on a `PArray`, and (2) a Ruby program by replacing each of such calls to with an execution of the wrapper function.

Figure 2 shows the kernel and wrapper functions generated from the Ikra program in figure 1. The function annotated as "global" from line 3 is the kernel function that will be executed by each thread on a GPU. It takes the parameters to the method, a pointer to the result array, and values of lexical variables.

The body of the function is generated by filling holes in a skeleton function with the code fragments translated from the block body. The skeleton part is responsible to parallelization; i.e., computing indices from thread numbers, checking boundaries, iterating an inner loop (which does not appear in this example), synchronizing with other threads (which also does not appear), and storing results.

The wrapper function from line 17 receives arguments from Ruby, and merely invokes the kernel function after converting Ruby values into C's.

5. PRELIMINARY PERFORMANCE EVALUATION

We compared execution times of simple programs, namely computation of a Mandelbrot set (the same program in figure 1 without the visualization part) and summation of random numbers, written in CUDA, Ikra, C and Ruby. All executions are taken place on a 2.5 GHz Intel Core 2 Quad processor with an nVidia GeFore GTX 465 having 352 CUDA processor cores at 1215 MHz, running Debian 4.1. The implementations of CUDA, C and Ruby are nvcc 3.1v0.2.1221 with CUDA Driver 4.0; GCC 4.1.3 with `-O3` switch; and

[5]Currently, precision of those numbers is controlled by a parameter written in a separate configuration file of the compiler. A future version will allow to specify through directives embedded in a program.

[6]For an explanatory purpose, we used `map` operation, which can be replaced with `each` operation. However, the `each` operation is not a parallel operation in Ikra, because it is used for side-effecting data other than the array itself through destructive assignments.

```
__device__ float sq(float x){return x*x;}

__global__ void __kernel_0(int _max0, int _max1,
    float* _result, float res, int i_size, int inf
    , float r_min, int limit, float i_min, float
    cr, int r_size, float ci){
  float zr,zi,zr1,zi1;
  int r_idx = threadIdx.x + blockDim.x*blockIdx.x;
  int c_idx = threadIdx.y + blockDim.y*blockIdx.y;
  if(r_idx<_max1 && c_idx<_max0){
    cr=r_min+(res*r_idx); ci=i_min+(res*c_idx);
    iter = 0; zr = 0.0; zi = 0.0;
    while (iter<limit && sqrt(sq(zr)+sq(zi))<inf){
      zr1 = (((zr * zr) - (zi * zi)) + cr);
      zi1 = (((zr * zi) + (zi * zr)) + ci);
      zr = zr1; zi = zi1;
      iter = (iter + 1); }
    _result[r_idx * _max1 + c_idx] = iter; } }

VALUE _kernel_0_wrapper(VALUE self, VALUE gpu_ptrs
    , VALUE block_size, VALUE size, VALUE args){
  /* check runtime types of arguments */
  float* d_ptr_0 = (float*)FIX2INT(rb_ary_entry(
      gpu_ptrs, 0));
  int _max0 = FIX2INT(rb_ary_entry(size, 0));
  int _max1 = FIX2INT(rb_ary_entry(size, 1));
  float arg_0 = NUM2FLOAT(rb_ary_entry(args,0));
  //...extract and convert arguments...
  dim3 db, dg; // set dimensions
  __kernel_0<<<dg,db>>>(_max0, _max1, d_ptr_0,
      arg_0, arg_1, arg_2, arg_3, arg_4, arg_5,
      arg_6, arg_7, arg_8);
  return Qnil; }
```

Figure 2: CUDA code generated by Ikra.

CRuby 1.8.7-p334, respectively.

Figure 3 shows the execution times of the benchmark programs. The top graph compares execution times of two programs in four language implementations executed with and without a GPU. As can bee seen, the Ikra versions are as fast as or slightly slower than the CUDA versions, and significantly faster than the C and Ruby versions.

The bottom graph shows the times spent for memory transfer and computation by the GPU cores. It also shows that the Mandelbrot kernel in Ikra is slower than that in CUDA. We suspect that it is due to the redundant CUDA code generated by Ikra, though we are currently investigating detailed causes.

6. RELATED WORK

Accelerate[1] is a Haskell-like functional language for GPGPU. Similar to Ikra, it offers a special array-type as a data-parallel abstraction, relies on a type inference system for identifying kernel code, and employs skeleton-based code generation. Since Accelerate is based on a functional language rigorously typed, the Accelerate program has to explicitly convert between parallel and sequential arrays. Besides differences in the type systems of underlying languages, we presume that the implementation techniques in Accelerate and Ikra are mostly interchangeable.

Firepile[4] is a Scala framework for GPGPU. It takes a higher-order function in Scala, and dynamically generates an OpenCL kernel for parallel execution on a GPU. Unlike Ikra, Firepile offers as low-level abstractions as those offered by OpenCL or CUDA. Firepile generates GPU code at runtime. This is mainly because the kernel code in Firepile is supplied as a higher-order function, whose body cannot be easily determined in Scala. In Ruby, it is common to pro-

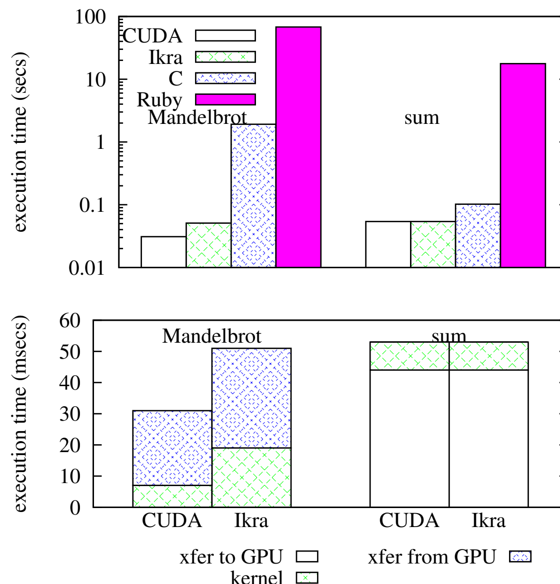

Figure 3: Preliminary benchmark results.

vide the block body in place of a method invocation, which makes it easier to identify the kernel code.

Another approach to provide parallel abstractions is to support GPGPU in a parallelizing compiler such as OpenMP[3]. Based on loop parallelism, we believe that the abstractions are at rather lower-level than data-parallel abstractions. However, matured compilation techniques developed for those compilers, such as the loop dependency analysis, would also be useful in our approach.

There are several systems for scripting languages, such as PyCUDA/PyOpenCL[2], that support GPGPU by embedding CUDA kernel code inside a program in a script language. Differently from those systems, Ikra allows the programmer to write kernel code in Ruby.

7. CONCLUSION

This paper presented the design of Ikra, a data-parallel extension to Ruby for supporting GPGPU and an overview of its implementation. Our prototype implementation showed that Ikra's type inference and compilation system offers comparable performance on a GPU to the hand-written CUDA code, though more work is needed to support applications beyond simple ones.

8. REFERENCES

[1] Chakravarty, M. M., et al.Accelerating Haskell array codes with multicore GPUs. In *DAMP*, pp. 3–14, 2011.

[2] Klöckner, et al.PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. submitted for publication, Nov. 2009.

[3] Lee, S., Min, S.-J., and Eigenmann, R. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP*, pp. 101–110, 2009.

[4] Nystrom, N., White, D., and Das, K. Firepile: run-time compilation for GPUs in Scala. In *GPCE*, pp. 107–116, 2011.