A Dynamically-typed Language for Prototyping High-Performance Data Parallel Programs

Hidehiko Masuhara and Tomoyuki Aotani (Tokyo Tech)

Ikra is

- an extended Ruby impl.
- w/data-parallel exec. (map & reduce over arrays)
- targeting GPGPU (& TSUBAME in future)

example: 2D diffusion + visualization

```
require 'sdl'  # user requested library

# array initialization
a = Array.new_(SIZE,SIZE){ |x,y|
  ...initialization... }
while true
  a = a.neighbor9.map{ |n|
   (n[-1,-1]+n[-1, 0]+n[-1, 1]+
    n[ 0,-1]+n[ 0, 0]+n[ 0, 1]+
    n[ 1,-1]+n[ 1, 0]+n[ 1, 1])/9
  }
  # visualize the array
  p = a.map{|v| colorHSB(v,1,0.5)}
  show(SIZE,SIZE,p.pack("I!*"))
end
```

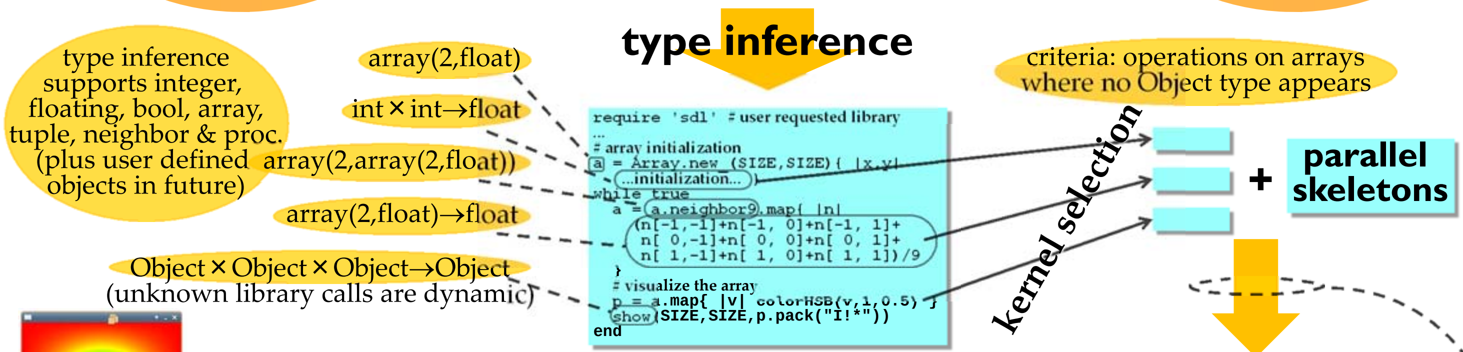
stencil code

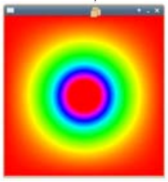
graphic library

Goal of Ikra

- quick prototyping of data-parallel programs
- integration with Ruby
  - ➢ straightforward APIs
  - ➢ minimal annotations
  - ➢ compatibility with existing libraries

type inference





visualize

sequential code (Ruby)

```
require('parray')  #parallel array class
require('kern')    #compiled kernel code
require('sdl')     #user requested library
...
a = (PArray.new_(SIZE,SIZE){ |ptr,s0,s1|
  Kern.init0(ptr,s0,s1)
}
while true do
  a = a.neighbor9().map(){
    |pin,z,pout,s0,s1|
    Kern.map1(pin,z,pout,s0,s1)
  }
  p = a.map(){ | pin,pout,s0,s1 |
    Kern.map2(pin,pout,s0,s1)
  }
  show(SIZE,SIZE,p.pack("I!*"))
end
```

Ruby + Ruby libs.

lib. call

native method call

stock Ruby VM (CPU)

kernel & wrapper (CUDA)

```
__device__
inline float body_map1(float n_1_1,float n_10,float n_11,float n0_1,
                float n00,float n01,float n1_1,float n10,float n11){
  return (n_1_1+n_10+n_11+n0_1+n...
        n1_1+n10+n11)/9;
}
__global__ void body_map1wrap(float* in,float z, float* out,
                          int s0,int s1){
  int offset=blockIdx.x*blockDim.x+threadIdx.x;
  if(offset<s0*s1){
    int i0=(offset%(s0*s1))/s1, i1=offset%s1;
   out[offset]=
    body_map1(aref(in,i0-1,i1-1,s0,s1,z),
              aref(in,i0-1,i1,  s0,s1,z),
                 ...rest 7 neighbors...);
} }
inline void kernel_map1(float* in,float z,float* out,
                     int s0,int s1){
  body_map1wrap<<<(s0*s1+NUM_THREADS-1)/NUM_TH...
            NUM_THREADS>>>(in,z,out,s0,s1)
}
VALUE wrapper_map1(VALUE self,VALUE in,VALUE z,VALUE out,
                 VALUE s0,VALUE s1){
  kernel_map1(get_pa(in,float),IKRA_NUM2FLOAT(z),
            get_pa(out,float),NUM2INT(s0),NUM2INT(s1));
  return Qnil;
}
```

kernel (from user prog.)

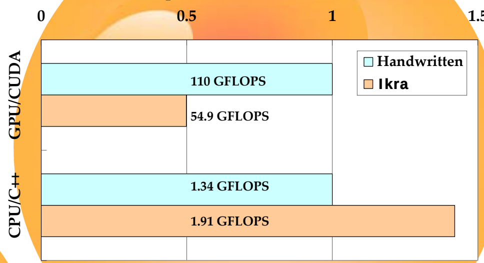
kernel (from skeleton)

kernel call

wrapper

GPU

Performance



Simple 3D diffusion over 256x256x256 grids; the handwritten version is distributed with Physis [Maruyama'11] / GPU: Tesla K20Xm (2688 cores), nvcc 6.0 –O3, single precision / CPU: Xeon X5670 2.93 GHz, g++ 4.3 –O3, Ruby 1.9.3p448

~50% of hand-written CUDA code due to different implementation of boundary checking

~ 1000x faster than Ruby interpreter

API

compatible w/Ruby's Array

`PArray.new(`$s_1,..,s_N$`)` `{ |`$i_1,..,i_N$`|` e `}` constructs N-dim. array initialized by e
a`.map{ |`v`|` e `}` constructs a new array from elements in a
a`.reduce(`z`){ |`x,y`|` e `}` reduces all elements in a by e
a_1`.zip(`$a_2,..,a_N$`)` virtually constructs an array of N-tuples
a`.neighbor`N`()` virtually constructs an array of N-neighbors

extended to support stencil computation

Optimizations (plan)

- separating boundary comp.
- spatial/temporal blocking
- loop fusion
- communication/computation overlapping

designing a modular framework