GPGPU Composition with OCaml

Array 2014

Mathias Bourgoin - Emmanuel Chailloux

June 13, 2014

Motivations

OCaml and GPGPU frameworks are very different

GPGPU frameworks are

- Highly Parallel
- Architecture Sensitive
- Very Low-Level
- Complex to program
- Very efficient

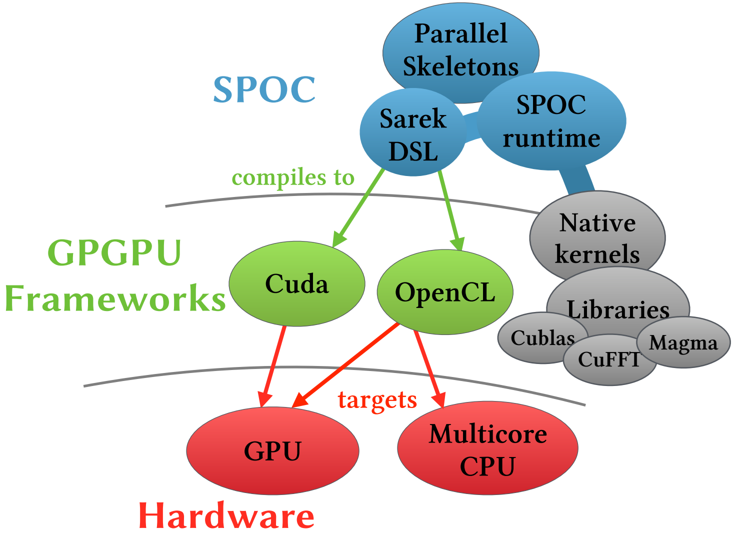
OCaml is

- Mainly Sequential
- Multi-platform/architecture
- Very High-Level
- Easy to program
- Cannot benefit from parallel architectures

Idea

- Provide GPGPU programming in their favorite language to OCaml developers.
- Use OCaml to develop high level abstractions for GPGPU.
- Make GPGPU programming safer and easier.

Overview


Parallel Skeletons
SPOC
Sarek DSL
SPOC runtime
compiles to
GPGPU Frameworks
Cuda
OpenCL
Native kernels
Libraries
Cublas
CuFFT
Magma
targets
GPU
Multicore CPU
Hardware

Host Side Solution

Stream Processing with OCaml

Features

- Targets Cuda/OpenCL frameworks with OCaml
- Unify these two frameworks
- Abstract memory transfers
- “Lazy” on demand transfers

GPGPU kernels

What we want

- Simple to express
- Predictable performance
- Easily extensible
- Current high performance libraries
- Optimisable
- Safer

Two Solutions

Interoperability with Cuda/OpenCL kernels

- Higher optimisations
- Compatible with current libraries
- Less safe

A DSL for OCaml : Sarek

- Easy to express
- Easy transformation from OCaml
- Safer

Sarek

Sarek Vector Addition

```
let vec_add = kern a b c n ->
  let open Std in
  let idx = global_thread_id in
  if idx < n then
    c.[<idx>] <- a.[<idx>] + b.[<idx>]
```

Sarek features

- Monomorphic
- Imperative
- Specific GPGPU globals
- Portable
- ML-like syntax

- Type inference
- Static type checking
- Static compilation to OCaml code
- Dynamic compilation to Cuda and OpenCL
- Exposes its internal representation to the host

Simple example : vector addition

SPOC & Sarek

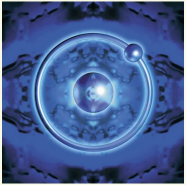
```
open Spoc
let vec_add = kern a b c n ->
  let open Std in
  let idx = global_thread_id in
  if idx < n then
    c.[<idx>] <- a.[<idx>] + b.[<idx>]

let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float64 n
let v2 = Vector.create Vector.float64 n
let v3 = Vector.create Vector.float64 n

let block = {blockX = 1024; blockY = 1; blockZ = 1}
let grid={gridX=(n+1024-1)/1024; gridY=1; gridZ=1}

let main () =
  random_fill v1;
  random_fill v2;
  Kirc.gen vec_add;
  Kirc.run vec_add  (v1, v2, v3, n) (block,grid) dev.(0);
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[%d] = %f; " i v3.[<i>]
  done;
```

- OCaml
- No explicit transfers
- Type inference
- Static typing
- Portable
- Heterogeneous

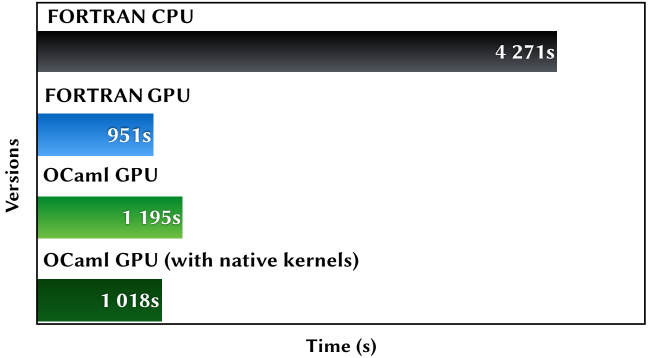
Sequential composition : real-world example

PROP

- Included in the 2DRMP suite
- HPC prize for Machine Utilization, awarded by the UK Research Councils' HEC Strategy Committee, 2006
- Simulates e^- scattering in H-like ions at intermediates energies
- PROP Propagates a $\mathcal{R}$-matrix in a two-electrons space
- Computations mainly implies matrix multiplications
- Computed matrices grow during computation
- Programmed in Fortran
- Compatible with sequential architectures, HPC clusters, super-computers



Results: PROP



SPOC+Sarek achieves 80% of hand-tuned Fortran performance.
SPOC+external kernels is on par with Fortran (93%)

Type-safe	30% code reduction
Memory manager + GC	No more transfers

Functional composition

Why?

- Helps describe complex algorithms
- Composition constructs can provide better optimizations

Problem

Kernels are procedures
Difficult to identify input/output automatically (with external native kernels)

Two solutions

- Analyze and transform Sarek internal representation
- Provide skeletons to associate kernels with their inputs/outputs

Solution 1 : Sarek transformations

Using Sarek

Transformations are OCaml functions transforming Sarek AST :
Example:

```
map (kern a -> b)
```

Scalar computations ($'a \rightarrow 'b$) are transformed into vector ones ($'a\ vector \rightarrow 'b\ vector$).

Vector addition

```
let v1 = Vector.create Vector.float64 10_000
and v2 = Vector.create Vector.float64 10_000 in
let v3 = map2 (kern a b -> a + b) v1 v2

val map2 :
  ('a -> 'b -> 'c) sarek_kernel ->
  ?dev:Spoc.Devices.device ->
  'a Spoc.Vector.vector ->
  'b Spoc.Vector.vector -> 'c Spoc.Vector.vector
```

Solution 1 : Sarek transformations

```
sort (kern a b -> a - b) vec1
val sort : ('a -> 'a -> int) sarek_kernel -> 'a vector -> unit
```

Injection into sort kernel

```
let bitonic_sort = kern v j k ->
 let open Std in
 let i = thread_idx_x +
        block_dim_x*block_idx_x in
 let ixj = Math.xor i j in
 let mutable temp = 0. in
 if ixj >= i then (
    if (Math.logical_and i k) = 0  then (
    if v.[< i >] - v.[< ixj >] > 0 then
     (temp := v.[<ixj>];
      v.[<ixj>] <- v.[<i>];
      v.[<i>] <- temp))
     else  if v.[< i >] - v.[< ixj >] <= 0 then
       (temp := v.[<ixj>];
        v.[<ixj>] <- v.[<i>];
        v.[<i>] <- temp);)
```

```
while !k <= size do
  j := !k lsr 1;
  while !j > 0 do
    run bitonic_sort
        (vec1,!j,!k)
        device;
    j := !j lsr 1;
  done;
  k := !k lsl 1 ;
done;
```

Host composition

Solution 2 : Parallel skeletons

A skeleton combines

- a kernel
- an execution environment
- an input
- an output

Two running functions:

- *run* : runs on one device
- *par_run* : tries running on a list of devices

- Explicitly describes relations between kernels/data
- Provides automatic optimizations

Examples

Skeleton

```
(* 'a : environment, 'b : input, 'c : output *)
val MAP : 'a external_kernel -> 'b vector -> 'c vector -> ('a,'b,'c) skeleton
val run : ('a,'b,'c) skeleton -> 'a -> 'c vector
```

- Automatic grid/block mapping on GPU
- Automatic parallelization on multiple GPUs

Composition

```
val PIPE : ('a,'b,'c) skeleton -> ('d,'c,'e) skeleton -> ('f,'b,'e) skeleton
```

- Automatic overlapping of transfers by computations

Conclusion

Our solution

- High-level multiparadigm programming language : **OCaml**
- Runtime library with vectors and lazy transfers : **SPOC**
- Easily extensible dedicated DSL : **Sarek**
- Kernel transformations combining kernels and host composition
- Functional composition and automatic optimizations *via* parallel skeletons

Portable approach

Requires a high level programming language with :

- Multiple paradigms
- Customizable garbage collector
- C interoperability

Future work

Improve composability

- Use Sarek to provide deeper transformations
- Build more skeletons
- Add a cost model to Sarek
- Target highly heterogeneous systems

High performance web-client programming

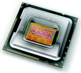
- Using js_of_ocaml
- Targeting WebCL
- Demands to translate SPOC's low-level C code to javascript
- Helps develop rich multimedia applications with intensive computations
- Eases accessibility : perfect playground for GPGPU/HPC courses

Thanks

Emmanuel Chailloux
Jean-Luc Lamotte

Open-Source distribution : http://www.algo-prog.info/spoc/
Or install it via OPAM, the OCaml Package Manager
SPOC is compatible with x86_64: Unix (Linux, Mac OS X), Windows

For more information
mathias.bourgoin@lip6.fr



CPU RAM

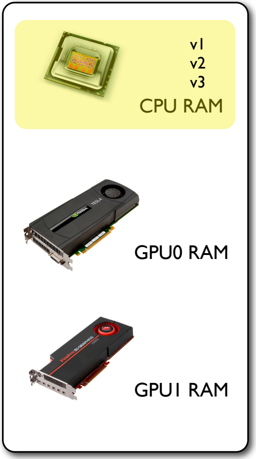
GPU0 RAM

GPU1 RAM

Example

```
let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float64 n
let v2 = Vector.create Vector.float64 n
let v3 = Vector.create Vector.float64 n

let k = vec_add (v1, v2, v3, n)
let block = {blockX = 1024; blockY = 1; blockZ = 1}
let grid={gridX=(n+1024-1)/1024; gridY=1; gridZ=1}

let main () =
  random_fill v1;
  random_fill v2;
  Kernel.run k (block,grid) dev.(0);
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[%d] = %f; " i v3.[<i>]
  done;
```

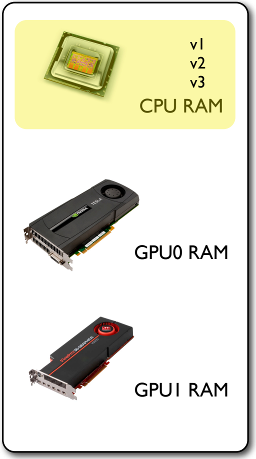
A Little Example



Example

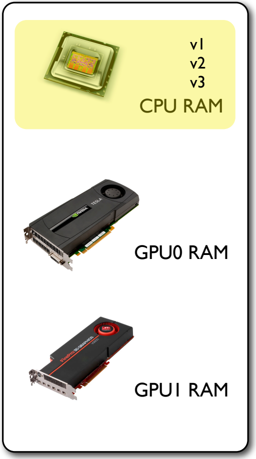
```
let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float64 n
let v2 = Vector.create Vector.float64 n
let v3 = Vector.create Vector.float64 n

let k = vec_add (v1, v2, v3, n)
let block = {blockX = 1024; blockY = 1; blockZ = 1}
let grid={gridX=(n+1024-1)/1024; gridY=1; gridZ=1}

let main () =
  random_fill v1;
  random_fill v2;
  Kernel.run k (block,grid) dev.(0);
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[%d] = %f; " i v3.[<i>]
  done;
```

A Little Example



Example

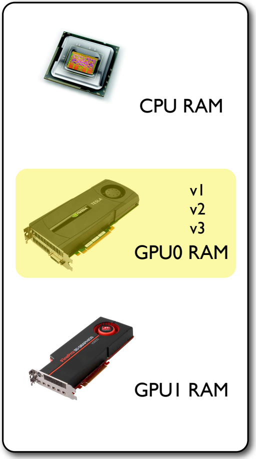
```
let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float64 n
let v2 = Vector.create Vector.float64 n
let v3 = Vector.create Vector.float64 n

let k = vec_add (v1, v2, v3, n)
let block = {blockX = 1024; blockY = 1; blockZ = 1}
let grid={gridX=(n+1024-1)/1024; gridY=1; gridZ=1}

let main () =
  random_fill v1;
  random_fill v2;
  Kernel.run k (block,grid) dev.(0);
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[%d] = %f; " i v3.[<i>]
  done;
```



Example

```
let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float64 n
let v2 = Vector.create Vector.float64 n
let v3 = Vector.create Vector.float64 n

let k = vec_add (v1, v2, v3, n)
let block = {blockX = 1024; blockY = 1; blockZ = 1}
let grid={gridX=(n+1024-1)/1024; gridY=1; gridZ=1}

let main () =
  random_fill v1;
  random_fill v2;
  Kernel.run k (block,grid) dev.(0);
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[%d] = %f; " i v3.[<i>]
  done;
```

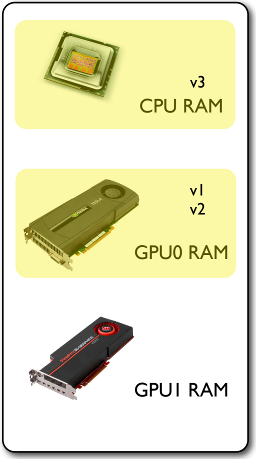
A Little Example



Example

```
let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float64 n
let v2 = Vector.create Vector.float64 n
let v3 = Vector.create Vector.float64 n

let k = vec_add (v1, v2, v3, n)
let block = {blockX = 1024; blockY = 1; blockZ = 1}
let grid={gridX=(n+1024-1)/1024; gridY=1; gridZ=1}

let main () =
  random_fill v1;
  random_fill v2;
  Kernel.run k (block,grid) dev.(0);
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[%d] = %f; " i v3.[<i>]
  done;
```

A Little Example



Example

```
let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float64 n
let v2 = Vector.create Vector.float64 n
let v3 = Vector.create Vector.float64 n

let k = vec_add (v1, v2, v3, n)
let block = {blockX = 1024; blockY = 1; blockZ = 1}
let grid={gridX=(n+1024-1)/1024; gridY=1; gridZ=1}

let main () =
  random_fill v1;
  random_fill v2;
  Kernel.run k (block,grid) dev.(0);
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[%d] = %f; " i v3.[<i>]
  done;
```

Example

Power iteration

SPOC

```
while (iter<IterMax)&&(max_n>eps) do
  let x=A*x0 in
  let m = max(x)in
  let x=u/m in
  let n = abs(x - x0) in
  max_n <- max(n);
  x0<-x;iter<-iter+1;
done
```

Skeletons

```
while (iter<IterMax)&&(max_n>eps) do
  let x= map ( * x0) A in
  let m = reduce (max) x in
  let x= map ( / m) u in
  let n = map (abs) x-x0 in
  max_n <- reduce max n;
  x0<-x;iter<-iter+1;
done
```

Composition

```
while (iter<IterMax)&&(max_n > eps) do
  let m= pipe (map ( *x0)) (reduce max) A in
  max_n <- pipe
           (pipe
            (map ( / m)
            (map  (abs(- x0)))))
           (reduce max) u;
  x0<-x;iter<-iter+1;
done
```

Example

Power Iteration

SPOC

```
while (iter<IterMax)&&(max_n>eps) do
  let x=A*x0 in
  let m = max(x)in
  let x=u/m in
  let n = abs(x - x0) in
  max_n <- max(n);
  x0<-x;iter<-iter+1;
done
```

Skeletons

```
while (iter<IterMax)&&(max_n>eps) do
  let x= map ( * x0) A in
  let m = reduce (max) x in
  let x= map ( / m) u in
  let n = map (abs) x-x0 in
  max_n <- reduce max n;
  x0<-x;iter<-iter+1;
done
```

Composition

```
while (iter<IterMax)&&(max_n > eps) do
  let m= pipe (map ( *x0)) (reduce max) A in
  max_n <- pipe
           (pipe
            (map ( / m)
            (map  (abs(- x0)))))
           (reduce max) u;
  x0<-x;iter<-iter+1;
done
```

Example

Power Iteration

SPOC

```
while (iter<IterMax)&&(max_n>eps) do
  let x=A*x0 in
  let m = max(x)in
  let x=u/m in
  let n = abs(x − x0) in
  max_n <− max(n);
  x0<−x;iter<−iter+1;
done
```

Skeletons

```
while (iter<IterMax)&&(max_n>eps) do
  let x= map ( * x0) A in
  let m = reduce (max) x in
  let x= map ( / m) u in
  let n = map (abs) x−x0 in
  max_n <− reduce max n;
  x0<−x;iter<−iter+1;
done
```

Composition

```
while (iter<IterMax)&&(max_n > eps) do
  let m= pipe (map ( *x0)) (reduce max) A in
  max_n <− pipe
             (pipe
              (map ( / m)
              (map  (abs(− x0)))))
             (reduce max) u;
  x0<−x;iter<−iter+1;
done
```

Composition benchmarks