Experiments with SPOC

Mathias Bourgoin - Emmanuel Chailloux - Jean-Luc Lamotte

January 24th, 2012

SPOC: Stream Processing with OCaml

SPOC

- An OCaml Library
- Managing Cuda/OpenCL kernels
- Managing transfers between Host and Guests automatically

OCaml

- High-Level language
  - **Efficient** Sequential Computations
  - **Statically Typed**
  - **Type inference**
  - **Multiparadigm** (imperative, object, functionnal, modular)
  - Compile to **Bytecode**/**native Code**
  - Memory Manager (very efficient **Garbage Collector**)
  - Interactive **Toplevel** (to learn, test and debug)
  - **Interoperability with C**
- Portable
  - System : Windows - Unix (OS-X, Linux. . .)
  - Architecture : x86, x86-64, PowerPC, ARM. . .

OCaml

- High-Level language
  - **Efficient** Sequential Computations
  - **Statically Typed**
  - **Type inference**
  - **Multiparadigm** (imperative, object, functionnal, modular)
  - Compile to **Bytecode**/**native Code**
  - Memory Manager (very efficient **Garbage Collector**)
  - Interactive **Toplevel** (to learn, test and debug)
  - **Interoperability with C**
- Portable
  - System : Windows - Unix (OS-X, Linux. . .)
  - Architecture : x86, x86-64, PowerPC, ARM. . .

Motivations

OCaml and GPGPU complement each other

GPGPU frameworks are

- Highly Parallel
- Architecture Sensitive
- Very Low-Level

Ocaml is

- Mainly Sequential
- Multi-platform/architecture
- Very High-Level

Idea

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

SPOC Overview

OCaml meets GPGPU

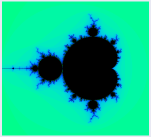
- OCaml developers can now use GPGPU programming
- SPOC allows to easily develop efficient GPGPU applications
  - Abstracted frameworks (Cuda/Opencl)
  - Automatic transfers
  - Kernel type safety
  - Efficient memory manager
- Can also be used as a tool for non OCaml developers
  - OCaml can be used to quickly express new algorithms
  - Still possible to use C externals. . .

Benchmarks - 1

Spoc easily speeds OCaml programs up

Mandelbrot

- Naive implementation
- Non optimized kernels
- Graphic display handled by CPU



Mandelbrot

	Intel i7			AMD 6950	Tesla C2070	C2070+6950	C2070+6950
	1 Core	4 Cores		OpenCL	Cuda	Cuda+OpenCL	OpenCL
GFLOPS SP	-	102.4		2250	1030	-	-
C	892s	307s	SPOC	12.84s	10.99s	6.56s	6.66s
Speedup	-	**1**		23,91	27,93	46,80	46,10

opencl kernel not vectorized

Benchmarks - 2

OCaml+Spoc runtime+GC overhead

Matrix Multiply SP

- optimized kernel
  - Nvidia → Cublas sgemm

Matrix Multiply SP

	Matrix Multiply	Matrix Multiply
Matrix size	21000	25000
Maximum memory needed	5.2GB	7.5GB
GFLOPS	156	139

Main Objectives

Goals

- Allow use of Cuda/OpenCL frameworks with OCaml
- Abstract these two frameworks
- Abstract memory
- Abstract memory transfers
- Use OCaml type-checking to ensure kernels type safety
- Propose Abstractions for GPGPU programming

Main Objectives

Goals

- Allow use of Cuda/OpenCL frameworks with OCaml
- Abstract these two frameworks
- Abstract memory
- Abstract memory transfers
- Use OCaml type-checking to ensure kernels type safety
- Propose Abstractions for GPGPU programming

Kernel Composition

Composition

Compose multiple kernels to express algorithms

Benefits

- Ease programming
- Allow new automatic optimizations

Problem

Spoc allows only to use external kernels.
To be composable, kernels must have an input/output

Skeletons

Problem

Spoc allows only to use external kernels.
To be composable, kernels must have an input/output

Work in progress

Describe Skeletons as :

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

Compose skeletons

Skeletons

$run : skeleton \rightarrow device \rightarrow vector \rightarrow vector$

Skeletons

Skeletons

- $map : kernel \rightarrow env \rightarrow vector \rightarrow skeleton$
- $reduce : kernel \rightarrow env \rightarrow vector \rightarrow skeleton$

Composition

- $pipe : skeleton \rightarrow skeleton \rightarrow skeleton$
- $par : skeleton \rightarrow skeleton \rightarrow skeleton$

Skeletons

$run : skeleton \rightarrow device \rightarrow vector \rightarrow vector$

Example

Power Iteration

```
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
```

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

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

Example

Power Iteration

```
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
```

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

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

Example

Power Iteration

```
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
```

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

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

Benefits

Benefits

- Explicitely describe relation between kernels/data
- Automatic blocks/grids mapping on GPUs
- Optimize data location (GPUs/CPU)
- Optimize automatic transfers

More Composition

Parallel Skeleton

par_run : *skeleton* → *device list* → *vector* → *vector*

Benefits

- Automatic blocks/grids mapping on GPUs
- Optimize data location (GPUs/CPU)
- Optimize automatic transfers

Previous Benchmarks

Mandelbrot

	Tesla C2070
Spoc	10.99s
Map Skeleton	10.99s

Results

Skeletons keep performance

Examples

To test skeleton compositions we used small kernels which do only basic tasks

Power Iteration

Two skeleton compositions

Game of Life

Each computed generation becomes the input of next computation
Two versions:

- Game1 : Each generation computed is brought back to CPU memory
- Game2 : Only the final generation is brought back

Game of Life

Game of Life

Each computed generation becomes the input of next computation
Two versions:

- Game1 : Each generation computed is brought back to CPU memory
- Game2 : Only the final generation is brought back

```
for i = 1 to last_generation do
 let current_generation = run (map (game_of_life)) last_generation in
 draw current_generation;
 last_generation <- current_generation;
done
```

```
let rec compose i c =
 if i = 1 then c
 else  compose (i-1) (pipe c (map game_of_life)) in
let final_generation =
  run (compose generations_count (map game_of_life))) first_generation in
draw final_generation;
```

Examples

Benchmarks

	Ocaml (1 thread)	Spoc	speedup
Power Iteration	1654.29s	382.77s	x4.32
Game1	244.24s	33.34s	x7.32
Game2	244.24s	4.88s	x50.05

Current Limitation

- Reduce currently sequential

Conclusion

Conclusion

- Spoc allows GPGPU programming with OCaml
- Skeletons help expressing algortihms
- Skeletons help automatic optimization
- Work in progress
- Already show promising results

Future Work

Embedded Language

- Describe full GPGGU kernel
- Automatic kernel generation from vector skeletons
- Describe kernels with
  - input
  - output
  - global environment

Composition

- Modify current skeletons with embedded language
- More skeletons

Thanks

Emmanuel Chailloux
Jean-Luc Lamotte

SPOC sources : `http://www.algo-prog.info/spoc/`
Spoc is compatible with x86_64: Unix (Linux, Mac OS X), Windows

For more information
mathias.bourgoin@lip6.fr