Experiments with SPOC

Mathias Bourgoin - Emmanuel Chailloux - Jean-Luc Lamotte

January 24th, 2012
SPOC: Stream Processing with OCaml

- 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, functional, 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
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
Spoc easily speeds OCaml programs up

Mandelbrot

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

<table>
<thead>
<tr>
<th>GFLOPS SP</th>
<th>Intel i7</th>
<th>AMD 6950</th>
<th>Tesla C2070</th>
<th>C2070+6950</th>
<th>C2070+6950</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 Core</td>
<td>4 Cores</td>
<td>OpenCL</td>
<td>Cuda</td>
<td>Cuda+OpenCL</td>
</tr>
<tr>
<td>GFLOPS SP</td>
<td>-</td>
<td>102.4</td>
<td>2250</td>
<td>1030</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>892s</td>
<td>307s</td>
<td>SPOC</td>
<td>12.84s</td>
<td>10.99s</td>
</tr>
<tr>
<td>Speedup</td>
<td>-</td>
<td>1</td>
<td>23.91</td>
<td>27.93</td>
<td>46.80</td>
</tr>
</tbody>
</table>

opencl kernel not vectorized
Benchmarks - 2

OCaml+Spoc runtime+GC overhead

Matrix Multiply SP

- optimized kernel
- Nvidia → Cublas sgemm

<table>
<thead>
<tr>
<th></th>
<th>Matrix Multiply</th>
<th>Matrix Multiply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix size</td>
<td>21000</td>
<td>25000</td>
</tr>
<tr>
<td>Maximum memory needed</td>
<td>5.2GB</td>
<td>7.5GB</td>
</tr>
<tr>
<td>GFLOPS</td>
<td>156</td>
<td>139</td>
</tr>
</tbody>
</table>
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
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 → device → vector → vector
Skeletons

- **map**: $\text{kernel} \rightarrow \text{env} \rightarrow \text{vector} \rightarrow \text{skeleton}$
- **reduce**: $\text{kernel} \rightarrow \text{env} \rightarrow \text{vector} \rightarrow \text{skeleton}$

Composition

- **pipe**: $\text{skeleton} \rightarrow \text{skeleton} \rightarrow \text{skeleton}$
- **par**: $\text{skeleton} \rightarrow \text{skeleton} \rightarrow \text{skeleton}$

Skeletons

- **run**: $\text{skeleton} \rightarrow \text{device} \rightarrow \text{vector} \rightarrow \text{vector}$
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
```
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

- 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 $\rightarrow$ device list $\rightarrow$ vector $\rightarrow$ vector

Benefits

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

Mandelbrot

<table>
<thead>
<tr>
<th></th>
<th>Tesla C2070</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spoc</td>
<td>10.99s</td>
</tr>
<tr>
<td>Map Skeleton</td>
<td>10.99s</td>
</tr>
</tbody>
</table>

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

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

```plaintext
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

<table>
<thead>
<tr>
<th></th>
<th>Ocaml (1 thread)</th>
<th>Spoc</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Iteration</td>
<td>1654.29s</td>
<td>382.77s</td>
<td>x4.32</td>
</tr>
<tr>
<td>Game1</td>
<td>244.24s</td>
<td>33.34s</td>
<td>x7.32</td>
</tr>
<tr>
<td>Game2</td>
<td>244.24s</td>
<td>4.88s</td>
<td>x50.05</td>
</tr>
</tbody>
</table>

### Current Limitation

- Reduce currently sequential
Conclusion

- Spoc allows GPGPU programming with OCaml
- Skeletons help expressing algorithms
- 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

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