SPOC : GPGPU programming through Stream Processing with OCaml

Mathias Bourgoin - Emmanuel Chailloux - Jean-Luc Lamotte

January 23rd, 2012
GPGPU Programming

Two main frameworks

- **Cuda**
- **OpenCL**

Different Languages

- To write kernels
  - **Assembly** (ptx, il, ...)
  - subsets of **C/C++**
- To manage kernels
  - **C/C++/Objective-C**
  - Fortran
  - Python
  - Scala
  - **Java**
  - **Scilab**
  - ...
# Who needs GPGPU

## Two kinds of programmers

<table>
<thead>
<tr>
<th>HPC / Scientific</th>
<th>General Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Known hardware</td>
<td>Unknown hardware/multiplatform</td>
</tr>
<tr>
<td>Heavy optimisation</td>
<td>Light optimisation</td>
</tr>
<tr>
<td>Problem driven</td>
<td>User driven</td>
</tr>
</tbody>
</table>

## Main Difficulties

<table>
<thead>
<tr>
<th>From the managing program</th>
<th>From the kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory transfers</td>
<td>Highly parallel</td>
</tr>
<tr>
<td>Multiple Devices management</td>
<td>Different levels of parallelism</td>
</tr>
<tr>
<td>Many different kinds of Devices</td>
<td>Different kinds of memory (global, local, . . . )</td>
</tr>
</tbody>
</table>

## What for?

- Data parallelism
- Distributed computation
- Hopefully high speed-ups
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**

<table>
<thead>
<tr>
<th>GPGPU frameworks are</th>
<th>Ocaml is</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highly Parallel</td>
<td>Mainly Sequential</td>
</tr>
<tr>
<td>Architecture Sensitive</td>
<td>Multi-platform/architecture</td>
</tr>
<tr>
<td>Very Low-Level</td>
<td>Very High-Level</td>
</tr>
</tbody>
</table>

**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
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

Solution
SPOC: Abstracting frameworks

Our choice

- **Dynamic linking.**
- The Cuda implementation uses the Cuda Driver API instead of the Runtime Library (lower level API, does not need the cudart library which is only provided with the Cuda SDK).

Compilation doesn’t need any specific hardware (no need of a Cuda/OpenCL compatible Device) or SDK.

Allows

- development **for multiple architectures from a single system**;
- executables to use **any OpenCL/Cuda Devices conjointly**;
- distribution of a **single executable for multiple architectures**.
Automatic Transfers

Vectors automatically move from CPU to Devices
- When a CPU function uses a vector, SPOC moves it to the CPU RAM
- When a kernel uses a vector, SPOC moves it to the Device Global Memory
- Unused vectors do not move
- SPOC allows users to explicitly force transfers

OCaml memory manager

Vectors are managed by the OCaml memory manager
- **Automatic allocation(s)**
- The GC **automatically frees** vectors (on the CPU as well as on Devices)
- Allocation failure during a transfer triggers a collection
A Little Example

Example

```ocaml
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 = vector_add v3 v1 v2 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 dev.(0) (block,grid) k;
  for i = 0 to Vector.length v3 - 1 do
    Printf.printf "res[\%d] = \%f; " i v3.[<i>]
  done;
```
A Little Example

Example

```ocaml
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 = vector_add v3 v1 v2 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 dev.(0) (block,grid) k;
  for i = 0 to Vector.length v3 − 1 do
    Printf.printf "res[\%d] = \%f; " i v3.[<i>]
  done;
```
A Little Example

Example

```ocaml
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 = vector_add v3 v1 v2 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 dev.(0) (block,grid) k;
    for i = 0 to Vector.length v3 − 1 do
        Printf.printf "res[\%d] = \%f; " i v3.[<i>]
    done;
```
A Little Example

Example

```ocaml
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 = vector_add v3 v1 v2 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 dev.(0) (block,grid) k;
    for i = 0 to Vector.length v3 − 1 do
        Printf.printf "res[\%d] = \%f; " i v3.[<i>]
    done;
```
Example

```ocaml
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 = vector_add v3 v1 v2 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 dev.(0) (block,grid) k;
  for i = 0 to Vector.length v3 − 1 do
    Printf.printf "res[\%d] = \%.f; " i v3.[<i>]
  done;
```
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 = vector_add v3 v1 v2 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 dev.(0) (block,grid) k;
  for i = 0 to Vector.length v3 − 1 do
    Printf.printf "res[\%d] = \%f; " i v3.[<i>]
done;
Kernels

Type-Safe Kernel Declaration

- Static arguments types checking (compilation time)
- Kernel.run compiles kernel from source (.ptx / .cl)

```
kernel vec_add : Vector.vfloat64 -> Vector.vfloat64 -> Vector.vfloat64 -> int -> unit = "kernels" "vec_add"

<table>
<thead>
<tr>
<th>Kernel launch</th>
<th>dev</th>
<th>compilation/execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel.run</td>
<td>dev</td>
<td>vec_add</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cuda</td>
</tr>
<tr>
<td></td>
<td></td>
<td>kernels.ptx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for i = 0 to Vector.length v3 do</td>
</tr>
<tr>
<td></td>
<td></td>
<td>printf &quot;res[%d] = %f;&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i (Mem.get v3 i)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>done;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OpenCL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>kernels.cl</td>
</tr>
</tbody>
</table>
```

M. Bourgoin - E. Chailloux - J-L. Lamotte (UPMC-LIP6)  SPOC : GPGPU programming through Stream Processing
Errors

Type-Checking: Error at compile-time

```
kernel vector_sum: Vector.vfloat64 -> unit = "my_file" "kernel_sum"
let v = Vector.create Vector.float32 1024 in
   Kernel.run device (block, grid) vector_sum v;
```

Type-Checking: Correct

```
kernel vector_sum: Vector.vfloat64 -> unit = "my_file" "kernel_sum"
let v = Vector.create Vector.float64 1024 in
   Kernel.run device (block, grid) vector_sum v;
```

Exceptions

SPOC raises OCaml exceptions when

- Kernel compilation/execution fails
- Not enough memory on devices
open Sopc
let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float32 n
let v2 = Vector.create Vector.float32 n
let v3 = Vector.create Vector.float32 n

let k = vector_add v3 v1 v2

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 dev.(0) (block,grid) k;
    for i = 0 to Vector.length v3 do
        Printf.printf "res[\%d] = \%f; " i (Mem.get v3 i)
    done;

open Sopc
let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float32 n
let v2 = Vector.create Vector.float32 n
let v3 = Vector.create Vector.float32 n

let v1_1 = Mem.sub_vector v1 0 (n/2)
let v1_2 = Mem.sub_vector v1 (n/2) (n/2)
let v2_1 = Mem.sub_vector v2 0 (n/2)
let v2_2 = Mem.sub_vector v2 (n/2) (n/2)
let v3_1 = Mem.sub_vector v3 0 (n/2)
let v3_2 = Mem.sub_vector v3 (n/2) (n/2)

let k1 = vector_add v3_1 v1_1 v2_2 (n/2)
let k2 = vector_add v3_2 v1_2 v2_2 (n/2)

let block = {blockX = 1024; blockY = 1; blockZ = 1}
let grid = {gridX = (n+1024-1)/(1024/2); gridY = 1; gridZ = 1}

let main () =
    random_fill(v1);
    random_fill(v2);
    Kernel.run dev.(0) (block,grid) k1;
    Kernel.run dev.(1) (block,grid) k2;
    Mem.to_cpu v3_1 ();
    Mem.to_cpu v3_2 ();
    Devices.flush dev.(0);
    Devices.flush dev.(1);

    for i = 0 to Vector.length v3 do
        Printf.printf "res[\%d] = \%f; " i (Mem.get v3 i)
    done;
Multi-GPU

**Devices/Frameworks**
- SPOC allows to use **any Device** from both frameworks **indifferently**
- SPOC allows to use **any Device** from both frameworks **conjointly**
- Tested with **Cuda** used **conjointly** with **OpenCL**
- Tested with **Tesla C2070** used **conjointly** with **AMD 6950**

**Transfers**
- Automatic Transfers **from CPU to Device**
- Automatic Transfers **from Device to CPU**
- Automatic Transfers **from Device to Device**
Benchmarks - 1

Spoc easily speeds OCaml programs up

**Mandelbrot**

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

<table>
<thead>
<tr>
<th></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>GFLOPS SP</td>
<td>1 Core 4 Cores</td>
<td>OpenCL</td>
<td>Cuda</td>
<td>Cuda+OpenCL</td>
<td>OpenCL</td>
</tr>
<tr>
<td>1 Core</td>
<td>-</td>
<td>102.4</td>
<td>2250</td>
<td>1030</td>
<td>-</td>
</tr>
<tr>
<td>4 Cores</td>
<td>102.4</td>
<td></td>
<td>2250</td>
<td>1030</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>892s 307s</td>
<td>SPOC</td>
<td>12.84s</td>
<td>10.99s</td>
<td>6.56s</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
Matrix Multiply SP

- optimized kernel
- Nvidia → Cublas sgemm

\[
\{A\} \times \{B\} = \{C\}
\]

sgemm
OCaml+Spoc runtime+GC overhead

Matrix Multiply SP
- optimized kernel
- Nvidia → Cublas sgemm

\[
\begin{align*}
\begin{bmatrix}
A_1 & B_1 \\
C_1 & D_1 \\
\end{bmatrix} 
\times 
\begin{bmatrix}
A_2 & B_2 \\
C_2 & D_2 \\
\end{bmatrix} 
= 
\begin{bmatrix}
A_1A_2 + B_1C_2 & A_1B_2 + B_1D_2 \\
C_1A_2 + D_1C_2 & C_1B_2 + D_1D_2 \\
\end{bmatrix}
\end{align*}
\]
Benchmarks - 2

OCaml+Spoc runtime+GC overhead

Matrix Multiply SP
- optimized kernel
- Nvidia → Cublas sgemm

<table>
<thead>
<tr>
<th></th>
<th>Matrix Multiply 1</th>
<th>Matrix Multiply 2</th>
<th>Matrix Multiply 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix size</td>
<td>21000</td>
<td>21000</td>
<td>25000</td>
</tr>
<tr>
<td>Maximum memory needed</td>
<td>5.2GB</td>
<td>5.2GB</td>
<td>7.5GB</td>
</tr>
<tr>
<td>GFLOPS¹</td>
<td>156</td>
<td>156</td>
<td>139</td>
</tr>
</tbody>
</table>

M. Bourgoin - E. Chailloux - J-L. Lamotte (UPMC-LIP6)  SPOC : GPGPU programming through Stream Processing  8 juin 2011  23 / 26
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 Toplevel allows to test kernels
  - OCaml can be used to quickly express new algorithms
  - Still possible to use C externals...
Current and Future Work

Abstractions
Skeletons and Composition: Tomorrow 4:30pm OpenGPU workshop

DSL
Embedded language to express kernel

Real World Use Case
- 2DRMP: Dimensional R-matrix propagation (Computer Physics Communications)
- Simulates electron scattering from H-like atoms and ions at intermediate energies
- Multi-Architecture: MultiCore, GPGPU, Clusters, GPU Clusters
- Translate from Fortran + Cuda to OCaml+SPOC + Cuda/OpenCL
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