Le langage OCaml et la programmation des GPU

GPU programming with OCaml

Mathias Bourgoin - Emmanuel Chailloux - Jean-Luc Lamotte

Le projet OpenGPU : un an plus tard
Ecole Polytechnique - 8 juin 2011
Outline

1 GPGPU Programming
2 OCaml
3 Motivations
4 GPGPU programming with OCaml
   - SPOC Overview
   - A Little Example
   - Tools
   - MultiGPU
5 Benchmarks
6 Conclusion
   - OCaml meets GPGPU
   - Future Work
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

- **HPC / Scientific**
  - Known Hardware
  - Heavy Optimisation
  - Problem Driven

- **General Purpose**
  - Unknown Hardware/Multiplatform
  - Light Optimisation
  - User Driven

Main Difficulties

- **From the managing program**
  - Memory transfers
  - Multiple Devices management
  - Many different kind of Devices

- **From the kernel**
  - Highly parallel
  - Different levels of parallelism
  - Different kinds of memory (global, local, . . .)

What for?

- Data parallelism
- Distributed Computation
- Hopefully High Speed-Ups
OCaml

- High-Level language
  - 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 - Usage

Domains

- Education
- Research
- Industry

Software Examples

- The Coq Proof Assistant
- LexiFi’s Modeling Language for Finance
- FFTW
- Scade Suite

The Caml Consortium:
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
Main Objectives

Goals

- Allow complete 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** (*Stream Programming OCaml*)
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.
SPOC: Abstracting Transfers

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** when created
- The garbage collector **automatically frees** vectors (on the CPU or on Devices)
- Allocation failure during a transfer triggers a collection
1 open Spoc
2 let dev = Devices.init ()
3 let n = 1_000_000
4 let v1 = Vector.create Vector.float32 n
5 let v2 = Vector.create Vector.float32 n
6 let v3 = Vector.create Vector.float32 n
7 let k = vector_add v3 v1 v2 n
8 let block = {blockX = 1024; blockY = 1; blockZ = 1}
9 let grid = {gridX = (n+1024-1)/1024; gridY = 1; gridZ = 1}
10 let main () =
11  random_fill(v1);
12  random_fill(v2);
13  Kernel.run dev.(0) (block,grid) k;
14  for i = 0 to Vector.length v3 do
15    Printf.printf "res[%d] = %f; "
16    i (Mem.get v3 i)
17  done;
1 open Spoc
2 let dev = Devices.init()
3 let n = 1,000,000
4 let v1 = Vector.create Vector.float32 n
5 let v2 = Vector.create Vector.float32 n
6 let v3 = Vector.create Vector.float32 n let k = vector_add v3 v1 v2 n
7 let block = {blockX = 1024; blockY = 1; blockZ = 1}
8 let grid = {gridX = (n+1024-1)/1024; gridY = 1; gridZ = 1}
9 let main() =
10 random_fill(v1);
11 random_fill(v2);
12 Kernel.run dev.(0) (block,grid) k;
13 for i = 0 to Vector.length v3 do
14 Printf.printf "res[%d] = %f; "
15 i (Mem.get v3 i)
16 done;
A Little Example

```
1 open Spoc
2 let dev = Devices.init ()
3 let n = 1_000_000
4 let v1 = Vector.create Vector.float32 n
5 let v2 = Vector.create Vector.float32 n
6 let v3 = Vector.create Vector.float32 n
7 let k = vector_add v3 v1 v2 n
8 let block = {blockX = 1024; blockY = 1; blockZ = 1}
9 let grid = {gridX = (n+1024-1)/1024; gridY = 1; gridZ = 1}
10 let main () =
11    random_fill(v1);
12    random_fill(v2);
13    Kernel.run dev.(0) (block,grid) k;
14    for i = 0 to Vector.length v3 do
15        Printf.printf "res[%d] = %f; "
16        i (Mem.get v3 i)
17    done;
```
open Spoc
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 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 do
        Printf.printf "res[%d] = %f; "
        i (Mem.get v3 i)
    done;
A Little Example

```ocaml
open Spoc
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 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 do
    Printf.printf "res[%d] = %f; ";
    i (Mem.get v3 i)
  done;
```
A Little Example

1 open Spoc
2 let dev = Devices.init ()
3 let n = 1.000_000
4 let v1 = Vector.create Vector.float32 n
5 let v2 = Vector.create Vector.float32 n
6 let v3 = Vector.create Vector.float32 n
7 let k = vector.add v3 v1 v2 n
8 let block = {blockX = 1024; blockY = 1; blockZ = 1}
9 let grid = {gridX = (n+1024-1)/1024; gridY = 1; gridZ = 1}
10 let main () =
11   random.fill(v1);
12   random.fill(v2);
13   Kernel.run dev.(0) (block,grid) k;
14   for i = 0 to Vector.length v3 do
15     Printf.printf "res[\%d] = \%f; "
16     i (Mem.get v3 i)
17   done;
A Little Example

```ocaml
1 open Spoc
2 let dev = Devices.init ()
3 let n = 1_000_000
4 let v1 = Vector.create Vector.float32 n
5 let v2 = Vector.create Vector.float32 n
6 let v3 = Vector.create Vector.float32 n
7 let k = vector.add v3 v1 v2 n
8 let block = {blockX = 1024; blockY = 1; blockZ = 1}
9 let grid = {gridX = (n+1024-1)/1024; gridY = 1; gridZ = 1}
10 let main () =
11 random.fill(v1);
12 random.fill(v2);
13 Kernel.run dev.(0) (block,(grid) k:
14   for i = 0 to Vector.length v3 do
15     Printf.printf "res[%d] = %f; ");
16       i (Mem.get v3 i)
17 done;
```
Kernels

Type-Safe Kernel Declaration

**kernel** `vec_add : Vector.vfloat64 → Vector.vfloat64 → Vector.vfloat64 → unit = "source_file" "kernel_name"

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

```
kernels.cl
__kernel void vec_add (...){
...
}
Kernel.run dev vec_add
Cuda
OpenCL
dev
kernel vec_add :  ... i = 0 to Vector.length v3 do
  printf "res[%d] = %f;"
i (Mem.get v3 i)
done;
kernel call compilation/execution
```
Errors

**Type-Checking: Error at compile-time**

1. `kernel` `vector_sum`: `Vector.float64 → unit = "my_file" "kernel_sum"
2. `let v = Vector.create Vector.float32 1024 in`
3. `Kernel.run device (block, grid) vector_sum v;`

**Type-Checking: Correct**

1. `kernel` `vector_sum`: `Vector.float64 → unit = "my_file" "kernel_sum"
2. `let v = Vector.create Vector.float64 1024 in`
3. `Kernel.run device (block, grid) vector_sum v;`

**Exceptions**

SPOC raises OCaml exceptions when
- Kernel compilation/execution fails
- Not enough memory on devices
Tools

Development tools : OCaml Interactive Toplevel

```
# $ ./spoclevel_cublas
   Objective Caml version 3.12.0

   Camlp4 Parsing version 3.12.0

# open Spoc
Random.self_init();
let dev = (Devices.init()).(0) in
let a = Vector.create Vector.float32 10240 in
let res = ref 0. in
for i = 0 to 10239 do
  let tmp = Random.float 32. in
  Mem.set a i tmp;
  res := !res +. tmp;
done;
let gpu_res = Cublas.run dev (Cublas.cublasSasum 10240 a 1) in
(!res, gpu_res)
;;
- : float * float = (163561.160761084117, 163561.15625)
#```
Tools

Development tools

- IDE (OCaide plugin for Eclipse)
  http://www.algo-prog.info/ocaide/
- Cublas (v1)
- Optimized vector iterators
MultiGPU?

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 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 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 6970

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>OCaml Corei7 960</th>
<th>C i7 960</th>
<th>Spoc - CUDA NVIDIA C2070</th>
<th>Spoc - OpenCL AMD 6970</th>
<th>Cuda + OpenCL C2070 + 6970</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>913s</td>
<td>741s</td>
<td>6.49s</td>
<td>9.70s</td>
<td>4.74s</td>
</tr>
<tr>
<td>speedups</td>
<td>x1</td>
<td>x1.2</td>
<td>x140.7</td>
<td>x94.2</td>
<td>x192.6</td>
</tr>
</tbody>
</table>

opencl kernel not vectorized
OCaml+Spoc actually usable as a kernel composition language

Matrix Multiply SP

- 2 optimized kernels
  - Nvidia → Cublas sgemm
  - AMD → kernel from AMD OpenCL SDK

<table>
<thead>
<tr>
<th></th>
<th>Average Sequential</th>
<th>Cuda NVIDIA C2070</th>
<th>OpenCL AMD 6970</th>
<th>Cuda + OpenCL NVIDIA C2070 + AMD 6970</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>-</td>
<td>485 GFlops</td>
<td>330 GFlops</td>
<td>657 GFlops</td>
</tr>
<tr>
<td>OCaml</td>
<td>-</td>
<td>457 GFlops</td>
<td>322 GFlops</td>
<td>678 GFlops</td>
</tr>
<tr>
<td>Overhead</td>
<td>25.5%</td>
<td>5.7%</td>
<td>2.4%</td>
<td>-3.2%</td>
</tr>
</tbody>
</table>
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...
Conclusion - Future Work

Real world use case: PROP

- 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
- Test on Bull GPU Cluster

Objectives

- Use OCaml+S poc to simplify parallelism and transfers
- Verify that OCaml and SPOC enhance development speed, safety and maintainability while keeping high performances
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