GPGPU programming with OCaml
SPOC and Sarek

Mathias Bourgoin - Emmanuel Chailloux - Jean-Luc Lamotte

July 2nd, 2013
Outline

1. Introduction

2. GPGPU programming with OCaml
   - SPOC Overview
   - A Little Example

3. Expressing kernels
   - Interoperability with Cuda/OpenCL
   - A DSL for OCaml: Sarek

4. Benchmarks
   - Toy Examples
   - Real-world example

5. Using SPOC with Multicore CPUs?

6. Conclusion & Future Work
GPGPU?

Classic Dedicated GPU Hardware

- Several Multiprocessors
- Dedicated Memory
- Connected to a host through a PCI-Express slot
- Data are transferred between the GPU and the Host using DMA

Current Hardware

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td># cores</td>
<td>4–16</td>
<td>300–2000</td>
</tr>
<tr>
<td>Max Memory</td>
<td>32GB</td>
<td>6GB</td>
</tr>
<tr>
<td>GFLOPS SP</td>
<td>200</td>
<td>1000–4000</td>
</tr>
<tr>
<td>GFLOPS DP</td>
<td>100</td>
<td>100–1000</td>
</tr>
</tbody>
</table>

M. Bourgoin - E. Chailloux - J-L. Lamotte (UPMC-LIP6)
Vector Addition

```c
__kernel void vec_add(__global const double * c, __global const double * a, __global double * b, int N)
{
    int nIndex = get_global_id(0);
    if (nIndex >= N)
        return;
    c[nIndex] = a[nIndex] + b[nIndex];
}
```
GPGPU Programming In Practice
A Small Example: GPGPU Host Program in C

```c
// create OpenCL device & context
c_context hContext;
hContext = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, 0, 0, 0);

// query all devices available to the context
size_t nContextDescriptorSize;
clGetContextInfo(hContext, CL_CONTEXT_DEVICES, 0, 0, &nContextDescriptorSize);
cl_device_id *aDevices = malloc(nContextDescriptorSize);
clGetContextInfo(hContext, CL_CONTEXT_DEVICES, nContextDescriptorSize, aDevices, 0);

// create a command queue for first device the context reported
cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext, aDevices[0], 0, 0);

// create & compile program
cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, 1, &sProgramSource, 0, 0);
clBuildProgram(hProgram, 0, 0, 0, 0, 0);

// create kernel
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, "vec_add", 0);

// allocate device memory
cl_mem hDeviceMemA, hDeviceMemB, hDeviceMemC;
hDeviceMemA = clCreateBuffer(hContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, cnDimension * sizeof(cl_double), pA, 0);
hDeviceMemB = clCreateBuffer(hContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, cnDimension * sizeof(cl_double), pA, 0);
hDeviceMemC = clCreateBuffer(hContext, CL_MEM_WRITE_ONLY, cnDimension * sizeof(cl_double), 0, 0);

// setup parameter values
clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void *)&hDeviceMemA);
clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void *)&hDeviceMemB);
clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void *)&hDeviceMemC);

// execute kernel
clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0, &cnDimension, 0, 0, 0, 0);

// copy results from device back to host
clEnqueueReadBuffer(hContext, hDeviceMemC, CL_TRUE, 0, cnDimension * sizeof(cl_double), pC, 0, 0, 0);
clReleaseMemObj(hDeviceMemA);
clReleaseMemObj(hDeviceMemB);
clReleaseMemObj(hDeviceMemC);
```

M. Bourgoin - E. Chailloux - J-L. Lamotte (UPMC-LIP6)
GPGPU programming with OCaml
May 17th, 2013 5 / 24
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
Overview

1 Introduction

2 GPGPU programming with OCaml
   - SPOC Overview
   - A Little Example

3 Expressing kernels
   - Interoperability with Cuda/OpenCL
   - A DSL for OCaml: Sarek

4 Benchmarks
   - Toy Examples
   - Real-world example

5 Using SPOC with Multicore CPUs?

6 Conclusion & Future Work
Main Objectives

Goals

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

Host side solution
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(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 (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

```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 (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

```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 (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

```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 (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;
```
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 (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

```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 (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;
```
Overview

1. Introduction

2. GPGPU programming with OCaml
   - SPOC Overview
   - A Little Example

3. Expressing kernels
   - Interoperability with Cuda/OpenCL
   - A DSL for OCaml: Sarek

4. Benchmarks
   - Toy Examples
   - Real-world example

5. Using SPOC with Multicore CPUs?

6. Conclusion & Future Work
How to express 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
External 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"

Kernel.run dev vec_add
Cuda
for i = 0 to Vector.length v3 do
  printf "res[%d] = %f;" i (Mem.get v3 i)
done;
OpenCL
```

```
.kernels.ptx
.entry vec_add(...){
...
}

.kernels.cl
__kernel void vec_add (...){
...
}
```
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>]
```

OpenCL Vector Addition

```
__kernel void vec_add(__global const double * c, __global const double * a, __global double * b, int N)
{
  int nIndex = get_global_id(0);
  if (nIndex >= N)
    return;
  c[nIndex] = a[nIndex] + b[nIndex];
}
```
Sarek Vector Addition

```ocaml
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
Overview

1. Introduction

2. GPGPU programming with OCaml
   - SPOC Overview
   - A Little Example

3. Expressing kernels
   - Interoperability with Cuda/OpenCL
   - A DSL for OCaml: Sarek

4. Benchmarks
   - Toy Examples
   - Real-world example

5. Using SPOC with Multicore CPUs?

6. Conclusion & Future Work
Benchmarks: Toy Examples

Mandelbrot

Computation handled through SPOC
Graphics handled through OCaml graphics (by the CPU)

Matmult

Naive matrix multiply
Over two $2000 \times 2000$ matrices

Using unoptimised kernels (non vectorized, no shared memory, etc)
### Results: Toy Examples

<table>
<thead>
<tr>
<th>Sample / Device</th>
<th>OCaml Sequential (s)</th>
<th>C2070 Cuda (s)</th>
<th>GTX 680 Cuda (s)</th>
<th>AMD6950 OpenCL (s)</th>
<th>i7-3770 (Intel OpenCL) (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mandelbrot&lt;sub&gt;ext&lt;/sub&gt;</td>
<td>474.5</td>
<td>5.9×80.4</td>
<td>4.0×118.6</td>
<td>4.9×96.8</td>
<td>6.0×79.1</td>
</tr>
<tr>
<td>Mandelbrot&lt;sub&gt;Sarek&lt;/sub&gt;</td>
<td>474.5</td>
<td>7.0×67.8</td>
<td>4.8×98.8</td>
<td>5.6×84.7</td>
<td>7.2×65.9</td>
</tr>
<tr>
<td>Matmult&lt;sub&gt;ext&lt;/sub&gt;</td>
<td>85.0</td>
<td>1.3×65.4</td>
<td>1.7×50.0</td>
<td>0.3×283.3</td>
<td>4.8×17.7</td>
</tr>
<tr>
<td>Matmult&lt;sub&gt;Sarek&lt;/sub&gt;</td>
<td>85.0</td>
<td>1.7×50.0</td>
<td>2.1×40.5</td>
<td>0.3×283.3</td>
<td>6.2×13.7</td>
</tr>
</tbody>
</table>

Using Sarek offers very high performance for data parallel programs.
Using external kernels allows to achieve higher optimizations.
PROP

- Included in the 2DRMP\textsuperscript{a\textit{b}} suite
- 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

\textsuperscript{a}NS Scott, MP Scott, PG Burke, T. Stitt, V. Faro-Maza, C. Denis, and A. Maniopoulou. 2DRMP: A suite of two-dimensional R-matrix propagation codes. Computer Physics Communications, 2009

\textsuperscript{b}HPC prize for Machine Utilization, awarded by the UK Research Councils’ HEC Strategy Committee, 2006
## Results: PROP

<table>
<thead>
<tr>
<th>Running Device</th>
<th>Running Time</th>
<th>Speedup / Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran CPU 1 core</td>
<td>4271.00s (71m11s)</td>
<td>1.00</td>
</tr>
<tr>
<td>Fortran CPU 4 core</td>
<td>2178.00s (36m18s)</td>
<td>1.96</td>
</tr>
<tr>
<td>Fortran GPU</td>
<td>951.00s (15m51s)</td>
<td>4.49</td>
</tr>
<tr>
<td>OCaml GPU</td>
<td>1018.00s (16m58s)</td>
<td>4.20</td>
</tr>
<tr>
<td>OCaml (+ Sarek) GPU</td>
<td>1195.00s (19m55s)</td>
<td>3.57</td>
</tr>
</tbody>
</table>

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
- Ready for the real world...
Overview

1. Introduction

2. GPGPU programming with OCaml
   - SPOC Overview
   - A Little Example

3. Expressing kernels
   - Interoperability with Cuda/OpenCL
   - A DSL for OCaml: Sarek

4. Benchmarks
   - Toy Examples
   - Real-world example

5. Using SPOC with Multicore CPUs?

6. Conclusion & Future Work
Using SPOC with Multicore CPUs?

Why?

OCaml cannot run parallel threads...
Multiple “solutions” have been considered:

- New runtime/GC $\Rightarrow$ OC4MC experiment?
- Automatic forking $\Rightarrow$ ParMap?
- Extension for distributed computing $\Rightarrow$ JoCaml?
- Probably many other solutions (new compiler?, parallel virtual machine?, etc)
### Comparison

- **ParMap**: data parallel, very similar to current OCaml map/fold
- **OC4MC**: Posix threads, compatible with current OCaml code
- **SPOC**: GPGPU kernels on CPU, mainly data parallel, needs OpenCL

### Benchmarks

<table>
<thead>
<tr>
<th></th>
<th>OCaml</th>
<th>ParMap</th>
<th>OC4MC</th>
<th>SPOC + Sarek</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>11s14</td>
<td>3s30</td>
<td>-</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>Matmul</td>
<td>85s</td>
<td>-</td>
<td>28s</td>
<td>6.2s</td>
</tr>
</tbody>
</table>

Running on a quad-core Intel Core-i7 3770@3.5GHz
Overview

1. Introduction

2. GPGPU programming with OCaml
   - SPOC Overview
   - A Little Example

3. Expressing kernels
   - Interoperability with Cuda/OpenCL
   - A DSL for OCaml: Sarek

4. Benchmarks
   - Toy Examples
   - Real-world example

5. Using SPOC with Multicore CPUs?

6. Conclusion & Future Work
## Conclusion

### SPOC: Stream Processing with OCaml
- OCaml library
- Unifies Cuda/OpenCL
- Offers automatic transfers
- Is compatible with current high performance libraries

### Sarek: Stream ARchitecture using Extensible Kernels
- OCaml-like syntax
- Type inference
- Easily extensible via OCaml code
Conclusion

Results

- Great performance
- Portability for free
- Great for both GPU and multicore CPU
- Nice playground for further abstractions

Who can benefit from it?

- OCaml programmers → better performance
- HPC programmers → simpler and safer than usual low-level tools
- Parallel libraries developers → efficient, portable, extensible
- Education - Industry - Research
Current and Future Work

Sarek

- Custom types, Function declarations, Recursion, Exceptions, ...
- Build parallel skeletons using SPOC and Sarek

Example

```ocaml
let v1 = Vector.create Vector.float64 10_000
and v2 = Vector.create Vector.float64 10_000
in
let vec3 = map2 (kern a b -> a + b) vec1 vec2
```
Or install it via [OPAM](http://opam.ocaml.org), the OCaml Package Manager
SPOC is compatible with x86_64: Unix (Linux, Mac OS X), Windows

For more information
[mathias.bourgoin@lip6.fr](mailto:mathias.bourgoin@lip6.fr)