GPGPU programming with OCaml
SPOC and Sarek

Mathias Bourgoin - Emmanuel Chailloux - Jean-Luc Lamotte

May 17th, 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 Kernel Composition
   - Parallel Skeletons
   - Example

5 Benchmarks
   - Toy Examples
   - Composition
   - Real-world example

6 Using SPOC with Multicore CPUs?

7 Conclusion & Future Work
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>
GPGPU Usage

- High Performance Computing: better ratio GFLOPS/$ and GFLOPS/W
- Multimedia
- Video Games
- Anywhere there are a lot of data to compute on...
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
  - ...
Virtual Hardware

- Several Multiprocessors
  - Each contains several Stream Processors (Cores)
  - SIMT (Single Instruction, Multiple Threads)

Stream Processing

Given a set of data (a **stream**), a series of operations (**kernel** functions) are applied to each element in the stream.
Different layers of parallelism/memory

- Threads: computation units
- Blocks: grouping threads
- Grid: grouping blocks
- Local Memory: local to a thread
- Shared Memory: shared inside a block
- Global Memory: shared by the whole grid
Grid

Block 0

Shared Memory

Registers

Thread (0,0)

Local Mem

Registers

Thread (1,0)

Local Mem

Block 1

Shared Memory

Registers

Thread (0,1)

Local Mem

Registers

Thread (1,1)

Local Mem

Global Memory

Specific Memory

Memory

Virtual Compute Unit
Stream Processing

Different layers of parallelism/memory

- **Threads**: computation units
- **Blocks**: grouping threads
- **Grid**: grouping blocks
- **Local Memory**: local to a thread
- **Shared Memory**: shared inside a block
- **Global Memory**: shared by the whole grid

GPU is a guest!

Don’t forget the host memory and **DMA copy** between host and guest

<table>
<thead>
<tr>
<th></th>
<th>X86-CPU i7-3770K</th>
<th>Laptop GPU GTX 680M</th>
<th>Desktop GPU GTX 680</th>
<th>Desktop GPU 7970HD</th>
<th>HPC GPU K20X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem. Bandwidth</td>
<td>25.6GB/s</td>
<td>115.2 GB/s</td>
<td>192.2GB/s</td>
<td>264GB/s</td>
<td>250GB/s</td>
</tr>
</tbody>
</table>

Max PCI-Express 3.0 Bandwidth 16GB/s
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];
}
```
// create OpenCL device & context
cl_context hContext;
hContext = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU,
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);
// create & compile program
cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, 1,
sProgramSource, 0, 0);
clBuildProgram(hProgram, 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,
0, 0, 0);
hDeviceMemB = clCreateBuffer(hContext,
CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
0, 0, 0);
hDeviceMemC = clCreateBuffer(hContext,
CL_MEM_WRITE_ONLY,
0, 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);
// copy results from device back to host
clEnqueueReadBuffer(hContext, hDeviceMemC, CL_TRUE, 0,
&cnDimension, 0, 0, 0);
clReleaseMemObj(hDeviceMemA);
clReleaseMemObj(hDeviceMemB);
clReleaseMemObj(hDeviceMemC);
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, 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...
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 Kernel Composition
   - Parallel Skeletons
   - Example

5 Benchmarks
   - Toy Examples
   - Composition
   - Real-world example

6 Using SPOC with Multicore CPUs?

7 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

M. Bourgoin et al., HLPGPU 2012
M. Bourgoin et al., PPL, 2012
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 (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;
```

M. Bourgoin - E. Chailloux - J-L. Lamothe (UPMC-LIP6) - GPGPU programming with OCaml
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;
```

M. Bourgoin - E. Chailloux - J-L. Lamotte (UPMC-LIP6)  
GPGPU programming with OCaml  
May 17th, 2013 18 / 49
```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;
```
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. Kernel Composition
   - Parallel Skeletons
   - Example
5. Benchmarks
   - Toy Examples
   - Composition
   - Real-world example
6. Using SPOC with Multicore CPUs?
7. 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

A DSL for OCaml: Sarek
- Easy to express
- Easy transformation from OCaml
- Safer

Interoperability with Cuda/OpenCL kernels
- Higher optimisations
- Compatible with current libraries
- Less safe
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"
```

<table>
<thead>
<tr>
<th><strong>kernel launch</strong></th>
<th><strong>dev</strong></th>
<th><strong>compilation/execution</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel.run dev vec_add</td>
<td><img src="CUDA.png" alt="" /></td>
<td><img src="kernels.ptx" alt="" /></td>
</tr>
</tbody>
</table>
| ![](OpenCL.png) | ![](kernels.cl) | for i = 0 to Vector.length v3 do
  printf "res[%d] = %f;"
  i (Mem.get v3 i)
done; |

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

GPGPU programming with OCaml

May 17th, 2013
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
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>]
```

OpenCL 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];
}
```

M. Bourgoin et al., HLPP 2013
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>]
```

Sarek offers

- ML-like syntax
- Type inference
- Static type checking
- Static compilation to OCaml code
- Dynamic compilation to Cuda and OpenCL
Sarek Static Compilation

\[
\text{kern } a \rightarrow \text{let idx = Std.global_thread_id () in } a.[< \text{idx}>] \leftarrow 0
\]

\[
\text{Bind( (Id 0), (ModuleAccess((Std), (global_thread_id)), (VecSet(VecAcc...))))}
\]

**OCaml Code**

\[
\text{fun a } -> \text{let idx = Std.global_thread_id () in } a.[< \text{idx}>] \leftarrow 0\]

**Kir**

Kern
Params
VecVar 0
VecVar 1
...

**spoc_kernel**

class spoc_class1
method run = ...
method compile = ...
end
new spoc_class1

**Typing**
let \textit{my\_kernel} = \textit{kern} ... \rightarrow ... \\
...;; \\
\texttt{Kirc.gen my\_kernel;} \\
\texttt{Kirc.run my\_kernel (dev) (block,grid);}

- Compile to Cuda C source file
- \texttt{nvcc -O3 -ptx...}
- OpenCL C99
- Compile to Cuda ptx assembly
- Cuda run \textit{my\_kernel} \texttt{dev (block,grid)}
- OpenCL C99
- Cuda ptx assembly
- Return to OCaml code execution

GPGPU programming with OCaml

M. Bourgoin - E. Chailloux - J-L. Lamoé (UPMC-LIP6)

May 17th, 2013
open Sopc

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>]

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 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;
    Kirc.gen vec_add;
    Kirc.run vec_add (v1, v2, v3, n) (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. Kernel Composition
   - Parallel Skeletons
   - Example
5. Benchmarks
   - Toy Examples
   - Composition
   - Real-world example
6. Using SPOC with Multicore CPUs?
7. Conclusion & Future Work
## Kernel Composition

### Composition

Compose multiple kernels to express algorithms.

### Benefits

- Ease programming
- Allow new automatic optimizations

### Problems

To be composable, kernels must have an input/output.
Parallel Skeletons Using External Kernels

Using External kernels

Describe Skeletons as:

- an external kernel
- an execution environment
- an input
- an output

Two running functions:

- run: runs on one device
- par_run: tries running on a list devices

M. Bourgoin et al., OpenGPU Workshop, 2012
Two kinds of Skeletons

- map : kernel → env → vector → skeleton
- reduce : kernel → env → vector → skeleton

Skeleton Composition

- pipe : skeleton → skeleton → skeleton
- par : skeleton → skeleton → skeleton

Running Function

run : skeleton → device → vector → vector
Using Sarek

Skeletons are functions transforming Kir AST:

Example:

\[
\text{map } (\text{kern } a \rightarrow b) \Rightarrow \\
\text{Scalar computations } (\langle a \rightarrow b \rangle) \text{ are transformed into vector ones } (\langle a, c \rangle \text{ vector } \rightarrow \langle b, d \rangle \text{ vector}).
\]

Currently

Sarek skeletons generates spoc_kernels compatible with external kernel skeletons

\[
\text{val map : } (\text{a } \rightarrow \text{ b}) \text{ kirc_kernel } \rightarrow \text{ spoc_kernel } \rightarrow \\
\text{spoc_kernel } * (\langle \text{a}, \text{ c} \rangle \text{ vector } \rightarrow \langle \text{b}, \text{ d} \rangle \text{ vector}) \text{ kirc_kernel}
\]
Example

Power Iteration

SPOC

```ocaml
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
```

Skeletons

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

Composition

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

Power Iteration

SPOC

```ocaml
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
```

Skeletons

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

Composition

```ocaml
while (iter<IterMax) && (max_n > eps) do
  let m = pipe (map ( *x0)) (reduce max) A in
  max_n <- pipe
    (pipe
      (map ( / m)
        (map (abs x0[i])))
      (reduce max) u;
    x0 <- x; iter <- iter + 1;
done
```
### Power Iteration

**SPOC**

```ocaml
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
```

**Skeletons**

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

**Composition**

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

- Explicitly describe relation between kernels/data
- Automatic blocks/grids mapping on GPUs
- Optimize data location (GPUs/CPU)
- Optimize automatic transfers
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 Kernel Composition
   • Parallel Skeletons
   • Example

5 Benchmarks
   • Toy Examples
   • Composition
   • Real-world example

6 Using SPOC with Multicore CPUs?

7 Conclusion & Future Work
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\text{ext}</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\text{Sarek}</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>
<td></td>
</tr>
<tr>
<td>Matmult\text{ext}</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\text{Sarek}</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>
<td></td>
</tr>
</tbody>
</table>

Using Sarek offers very high performance for data parallel programs
Using external kernels allows to achieve higher optimizations
### Power Iteration: Using SPOC

<table>
<thead>
<tr>
<th>Device</th>
<th>Theoretical GFLOPS(DP)</th>
<th>Framework</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-3770</td>
<td>32 (1 core)</td>
<td>OCaml (1 Thread)</td>
<td>637</td>
<td>×1</td>
</tr>
<tr>
<td>Tesla C2070</td>
<td>515</td>
<td>SPOC (Cuda)</td>
<td>150</td>
<td>×4.24</td>
</tr>
<tr>
<td>Radeon HD 6950</td>
<td>562.5</td>
<td>SPOC (OpenCL)</td>
<td>101</td>
<td>×6.31</td>
</tr>
</tbody>
</table>

Using SPOC already improves performance
## Power Iteration: Using Skeletons

<table>
<thead>
<tr>
<th>Devices</th>
<th>Framework</th>
<th>Time (s)</th>
<th>Speedups</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-3770</td>
<td>OCaml</td>
<td>637</td>
<td>OCaml: -</td>
</tr>
<tr>
<td>Tesla C2070</td>
<td>Cuda</td>
<td>135</td>
<td>OCaml: $\times 4.72$</td>
</tr>
<tr>
<td>Radeon HD 6950</td>
<td>OpenCL</td>
<td>81</td>
<td>OCaml: $\times 7.86$</td>
</tr>
</tbody>
</table>

Skeletions help increase performance via automatic grid/bloc mapping.
Results : Using Composition

Power Iteration : Composing Skeletons

<table>
<thead>
<tr>
<th>Composition</th>
<th>Framework</th>
<th>Time (s)</th>
<th>Speedups</th>
</tr>
</thead>
<tbody>
<tr>
<td>Devices</td>
<td>OCaml</td>
<td>637</td>
<td>-</td>
</tr>
<tr>
<td>i7-3770</td>
<td>OCaml</td>
<td>637</td>
<td>-</td>
</tr>
<tr>
<td>Tesla C2070</td>
<td>Cuda</td>
<td>133</td>
<td>$\times 4.79$</td>
</tr>
<tr>
<td>Radeon HD 6950</td>
<td>OpenCL</td>
<td>74</td>
<td>$\times 8.61$</td>
</tr>
</tbody>
</table>

Pipe composition allows transfer overlapping by computation, increasing performance even further.
Real-world Example

**PROP**

- Included in the 2DRMP\(^a\)\(^b\) suite
- Simulates \(e^-\) scattering in H-like ions at intermediates energies
- PROP Propagates a \(R\)-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

\(^a\)NS Scott, MP Scott, PG Burke, T. Stitt, V. Faro-Maza, C. Denis, and A. Maniopoulou.

\(^b\)2DRMP : A suite of two-dimensional R-matrix propagation codes. Computer Physics Communications, 2009

\(^b\)HPC prize for Machine Utilization, awarded by the UK Research Councils’ HEC Strategy Committee, 2006
First modification (*Caps-Entreprise*)

- Matrix multiplication ported to Cuda using the HMPP Compile
- Propagation equation modified to handle bigger matrices
- Lower transfers/computation ratio but still many computations made by the CPU

Second modification (*LIP6*)

- Reduce transfer by performing all propagation computation on the GPGPU
- Overlaps transfers with computations over different sections of the $R$-matrix
- Uses Cublas and Magma library to perform computations
- Uses a C glue to bind Fortran code with Cuda
Our approach

- Translation of the computing part only (leaving I/O and initialisation to Fortran)
- Binding of a subset of the Cublas and Magma Library for OCaml (using SPOC)
- C glue between Fortran and OCaml

Result

- A program mixing Fortran, C, OCaml using Sarek
- Dramatic reduction of the code size
- No more transfers!!
<table>
<thead>
<tr>
<th>Running Device</th>
<th>Running Time</th>
<th>Speedup / Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CPU 1</td>
</tr>
<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 Kernel Composition
   - Parallel Skeletons
   - Example

5 Benchmarks
   - Toy Examples
   - Composition
   - Real-world example

6 Using SPOC with Multicore CPUs?

7 Conclusion & Future Work
Using SPOC with Multicore CPUs?

Why?

OCaml cannot run parallel threads...

Multiple “solutions” have been considered:

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

\(^a\)M. Bourgoin, P. Wang \textit{et al.}, IFL 2009
Benchmarks using SPOC on Multicore CPUs

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. Kernel Composition
   - Parallel Skeletons
   - Example
5. Benchmarks
   - Toy Examples
   - Composition
   - Real-world example
6. Using SPOC with Multicore CPUs?
7. 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

Skeletons and Composition
- Ease programming
- Allow automatic optimization
Results

- Great performance
- Great for both GPU and CPU
- Nice playground for further abstractions
Future Work

**SPOC**
- CPU optimization: no more copy
- Test on new architectures: ARM SOC, Xeon Phi

**Sarek**
- Custom types, Function declarations, Recursion, Exceptions, ...
- Allow shared memory access
- Optimize code generation
- Auto tuning for different GPGPU architectures

**Skeletons and Composition**
- Use Sarek to provide more skeletons
- e.g., Allow kernel splitting/merging
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