GPGPU programming with OCaml

SPOC and Sarek

Mathias Bourgoin - Emmanuel Chailloux - Jean-Luc Lamotte

July 2nd, 2013

Outline

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

	CPU	GPU
# cores	4–16	300–2000
Max Memory	32GB	6GB
GFLOPS SP	200	1000–4000
GFLOPS DP	100	100–1000

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];
}
```

```c
// 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, 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);
```

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

Overview

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

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

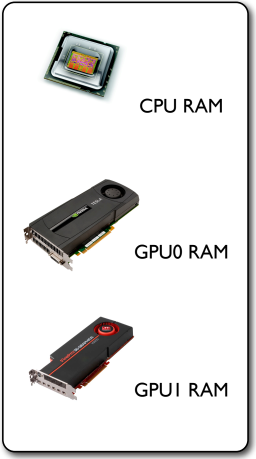
- When a CPU function uses a vector, SPOC moves it to the CPU RAM
- When a kernel uses a vector, SPOC moves it to the Device Global Memory
- Unused vectors do not move
- SPOC allows users to explicitly force transfers

OCaml memory manager

Vectors are managed by the OCaml memory manager

- **Automatic allocation(s)**
- The GC **automatically frees** vectors (on the CPU as well as on Devices)
- Allocation failure during a transfer triggers a collection

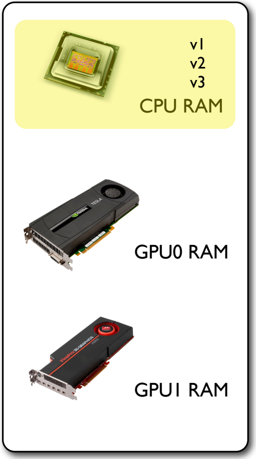
A Little Example



Example

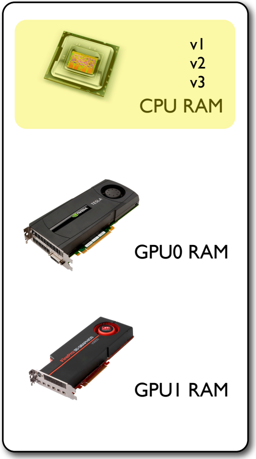
```
let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float64 n
let v2 = Vector.create Vector.float64 n
let v3 = Vector.create Vector.float64 n

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

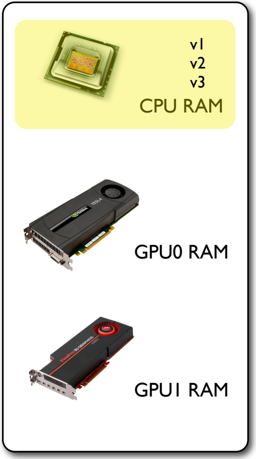
```
let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float64 n
let v2 = Vector.create Vector.float64 n
let v3 = Vector.create Vector.float64 n

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

```
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;
```

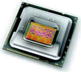
A Little Example



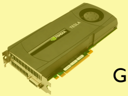
Example

```
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



CPU RAM



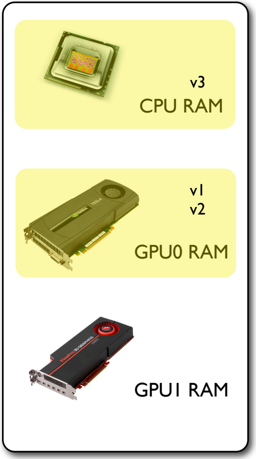
v1
v2
v3
GPU0 RAM

GPU1 RAM

Example

```
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

```
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

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

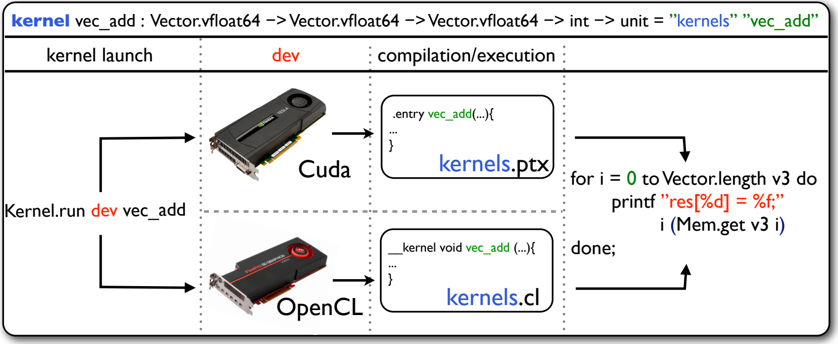
- Higher optimisations
- Compatible with current libraries
- Less safe

A DSL for OCaml : Sarek

- Easy to express
- Easy transformation from OCaml
- Safer

Type-Safe Kernel Declaration

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



Sarek : Stream ARchitecture using Extensible Kernels

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

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>]
```

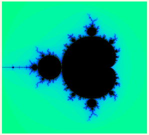
Sarek features

- 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

Benchmarks : Toy Examples

Mandelbrot



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

Matmult

Naive matrix multiply
Over two 2000×2000 matrices

Using unoptimised kernels (non vectorized, no shared memory, etc)

Results : Toy Examples

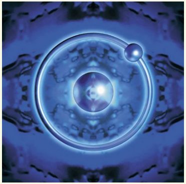
Sample / Device	OCaml Sequential (s)	C2070 Cuda (s)		GTX 680 Cuda (s)		AMD6950 OpenCL (s)		i7-3770 (Intel OpenCL) (s)	
Mandelbrot_{ext}	474.5	5.9	×80.4	4.0	×118.6	4.9	×96.8	6.0	×79.1
$\text{Mandelbrot}_{Sarek}$		7.0	×67.8	4.8	×98.8	5.6	×84.7	7.2	×65.9
Matmult_{ext}	85.0	1.3	×65.4	1.7	×50.0	0.3	×283.3	4.8	×17.7
Matmult_{Sarek}		1.7	×50.0	2.1	×40.5	0.3	×283.3	6.2	×13.7

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

Real-world Example

PROP

- Included in the 2DRMP[a][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



[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

[b] HPC prize for Machine Utilization, awarded by the UK Research Councils' HEC Strategy Committee, 2006

Results: PROP

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

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

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)

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

	OCaml	ParMap	OC4MC	SPOC + Sarek
Power	11s14	3s30	-	<1s
Matmul	85s	-	28s	6.2s

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

Overview

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 $\rightarrow$ better performance
- HPC programmers $\rightarrow$ simpler and safer than usual low-level tools
- Parallel libraries developers $\rightarrow$ efficient, portable, extensible
- Education - Industry - Research

Current and Future Work

Sarek

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

Example

```
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
```

Thanks

Emmanuel Chailloux
Jean-Luc Lamotte

open-source distribution : http://www.algo-prog.info/spoc/
Or install it via OPAM, the OCaml Package Manager
SPOC is compatible with x86_64: Unix (Linux, Mac OS X), Windows

For more information
mathias.bourgoin@lip6.fr