SPOC: GPGPU programming with OCaml

Mathias Bourgoin - Emmanuel Chailloux - Jean-Luc Lamotte

November 17th, 2011

Outline

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***
  - . . .

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	Mobile GPU	Desktop GPU		HPC GPU
	i7-2700K	Geforce 580M	Geforce 580	Radeon 6970HD	Tesla C2070
# Cores	4	384	512	1536	448
Clock Speed	3.50GHz	1.24GHz	1.54GHz	880MHz	1.15 GHz
Max Memory Size	32GB	2GB	1.5GB	2GB	6GB
Memory Bandwidth	21GB/s	96GB/s	192.4GB/s	176GB/s	144GB/s
SP GFLOPS	217.6	952.3	1581.1	2700	1030.5
DP GFLOPS	108.8	59.11-119	168	683	515.2

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 parallism
  - Different kinds of memory (global, local,. . .)

What for?

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

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.

GPGPU Programming In Practice 1

Stream Processing

Different layers of parallelism

- Threads : computation units
- Blocks : grouping threads
- Grid : grouping blocks

Different layers of memory

- Local Memory : local to a thread
- Shared Memory : shared inside a block
- Global Memory : shared by the whole grid

Different layers of parallelism

- Threads : computation units
- Blocks : grouping threads
- Grid : grouping blocks

Different layers of memory

- 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

	CPU	Mobile GPU	Desktop GPU		HPC GPU
	i7-2700K	Geforce 580M	Geforce 580	Radeon 6970HD	Tesla C2070
Memory Bandwidth	21GB/s	96GB/s	192.4GB/s	176GB/s	144GB/s

Max PCI-Express 3.0 Bandwidth 16GB/s

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];
}
```

GPGPU Programming In Practice 2

A Small Example

```
// 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);
```

Cuda - Proprietary

- Only Compatible with NVIDIA Hardware

Cuda Compute Capabilities

Feature support (unlisted features are supported for all compute capabilities)	**Compute capability (version)**				
	1.0	1.1	1.2	1.3	2.x
Integer atomic functions operating on 32-bit words in global memory					
atomicExch() operating on 32-bit floating point values in global memory					
Integer atomic functions operating on 32-bit words in shared memory					
atomicExch() operating on 32-bit floating point values in shared memory					
Integer atomic functions operating on 64-bit words in global memory					
Double-precision floating-point operations					
Atomic functions operating on 64-bit integer values in shared memory					
Floating-point atomic addition operating on 32-bit words in global and shared memory					
...					

Cuda - Proprietary

- Only Compatible with NVIDIA Hardware
- With different “Compute Capabilities”

GPGPU Programming In Practice 3

Portability Issues

Cuda - Proprietary

- Only Compatible with NVIDIA Hardware
- With different “Compute Capabilities”

OpenCL - Standard

- GPUs (NVIDIA, AMD)
- CPUs (X64 (Intel, AMD), PPC (IBM), ARM (Samsung, Creative))
- Accelerators (Cell Blade IBM)
- Same code on every platform

GPGPU Programming In Practice 3

Portability Issues

Cuda - Proprietary

- Only Compatible with NVIDIA Hardware
- With different “Compute Capabilities”

OpenCL - Standard

- GPUs (NVIDIA, AMD)
- CPUs (X64 (Intel, AMD), PPC (IBM), ARM (Samsung, Creative))
- Accelerators (Cell Blade IBM)
- Same code on every platform **brings very different performance**
- Different OpenCL extensions
- Different Architecture implies different optimizations
- Eg: NVIDIA GPUs = Scalar, AMD GPUs = Vector

GPGPU Programming

Problems

- Low Level
- Multiple Layers (Parallelism/Memory)
- Portability
- GPU program depends on Host management to provide high performance

But...

Very efficient hardware ... for a low cost

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

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 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

What SPOC is Not

- An OCaml to Cuda/OpenCL compiler
- An OCaml VM running on GPUs
- A tool to write GPGPU computing kernels
- A Cuda/OpenCL framework binding
- A set of optimized fonctions for OCaml

What SPOC is

- An OCaml library
- Managing Cuda/OpenCL kernels
- Managing transfers between Host and Guests automatically
- A tool to manage data usable with GPU kernels

SPOC focuses on Host code!

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

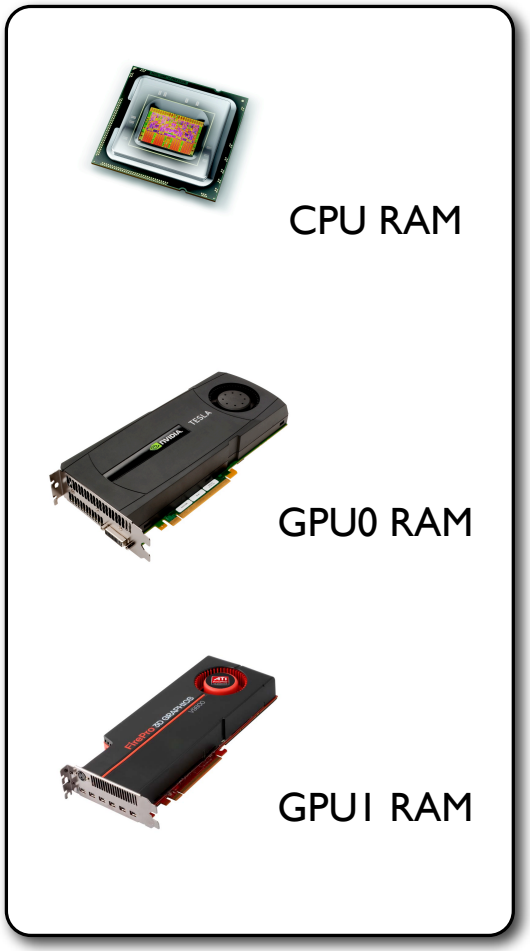
- When a CPU function uses a vector, SPOC moves it to the CPU RAM
- When a kernel uses a vector, SPOC moves it to the Device Global Memory
- Unused vectors do not move
- SPOC allows users to explicitely force transfers

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

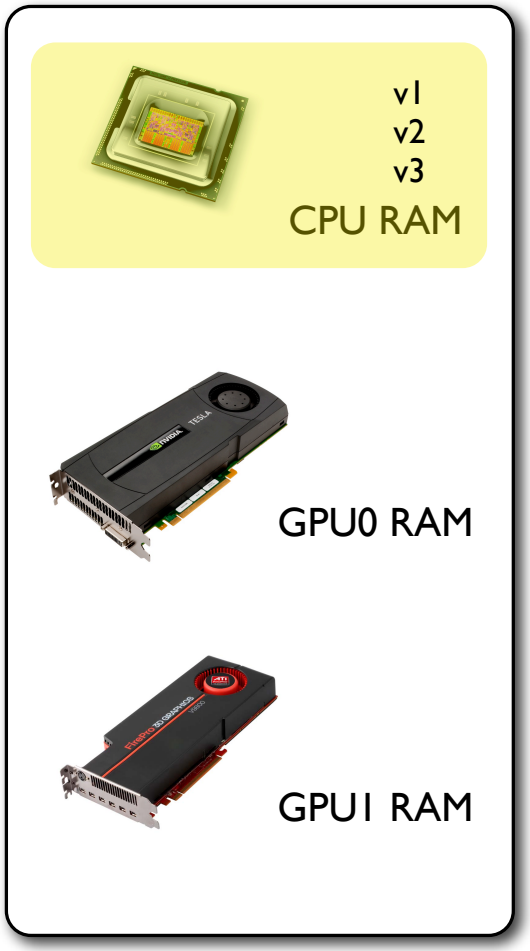
A Little Example



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 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;
```

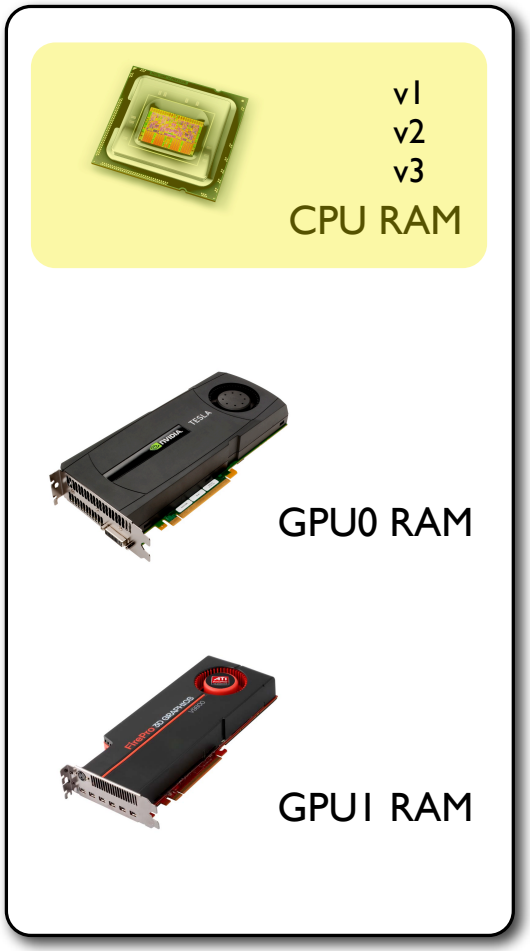
A Little Example



Example

```
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;
```

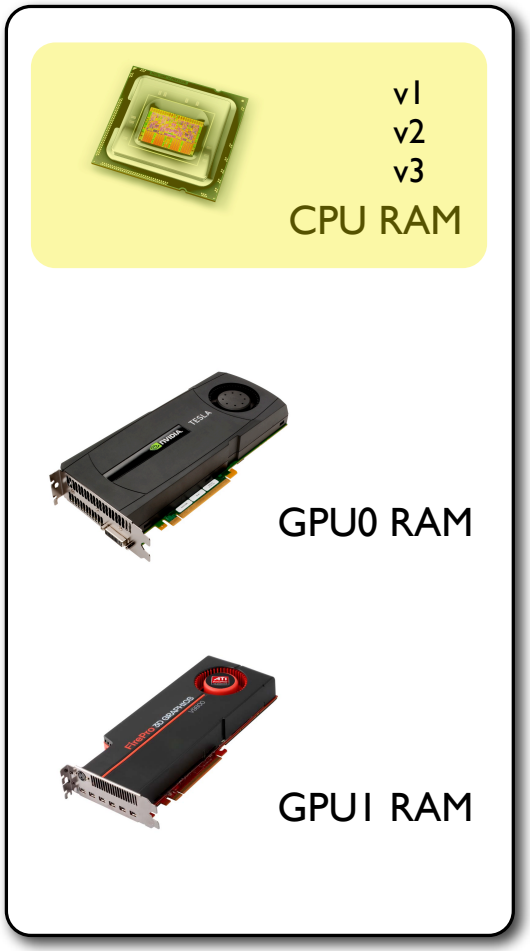
A Little Example



Example

```
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;
```

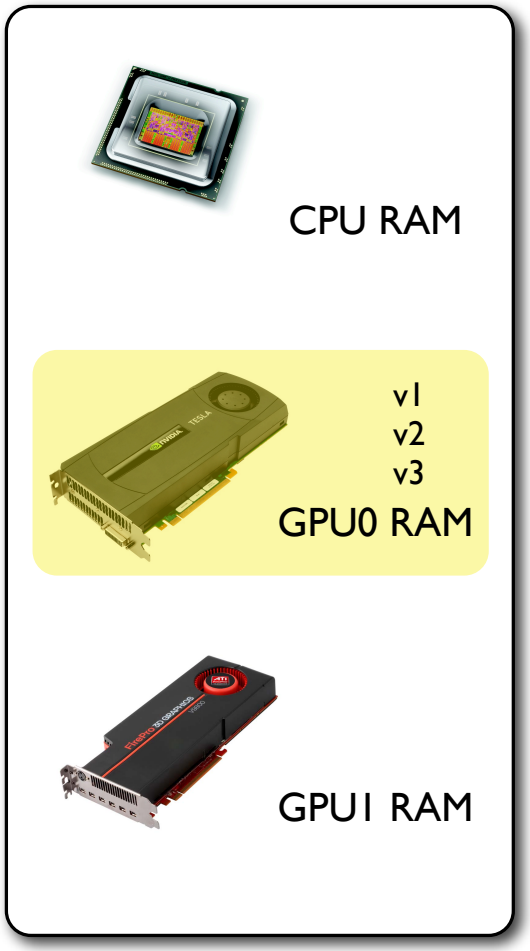
A Little Example



Example

```
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;
```

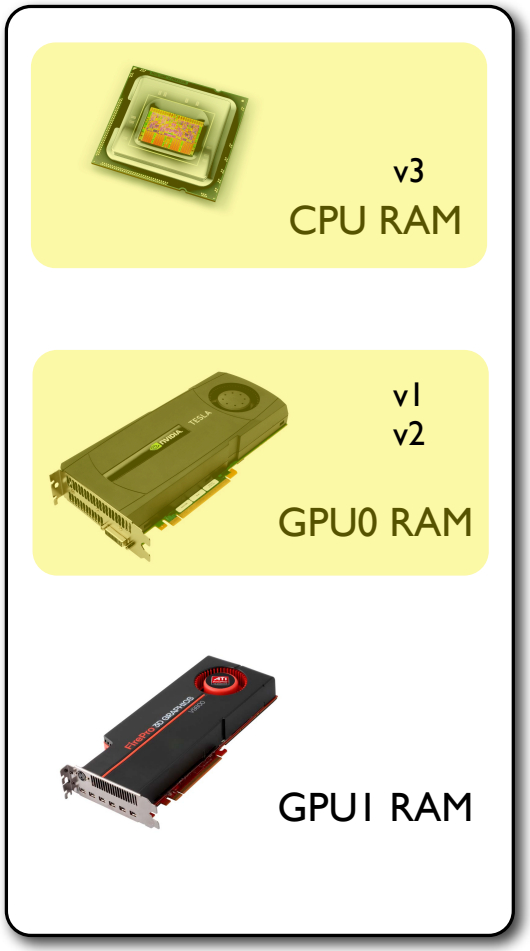
A Little Example



Example

```
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

```
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;
```

Remember

C Code

```
// 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);
```

Remember

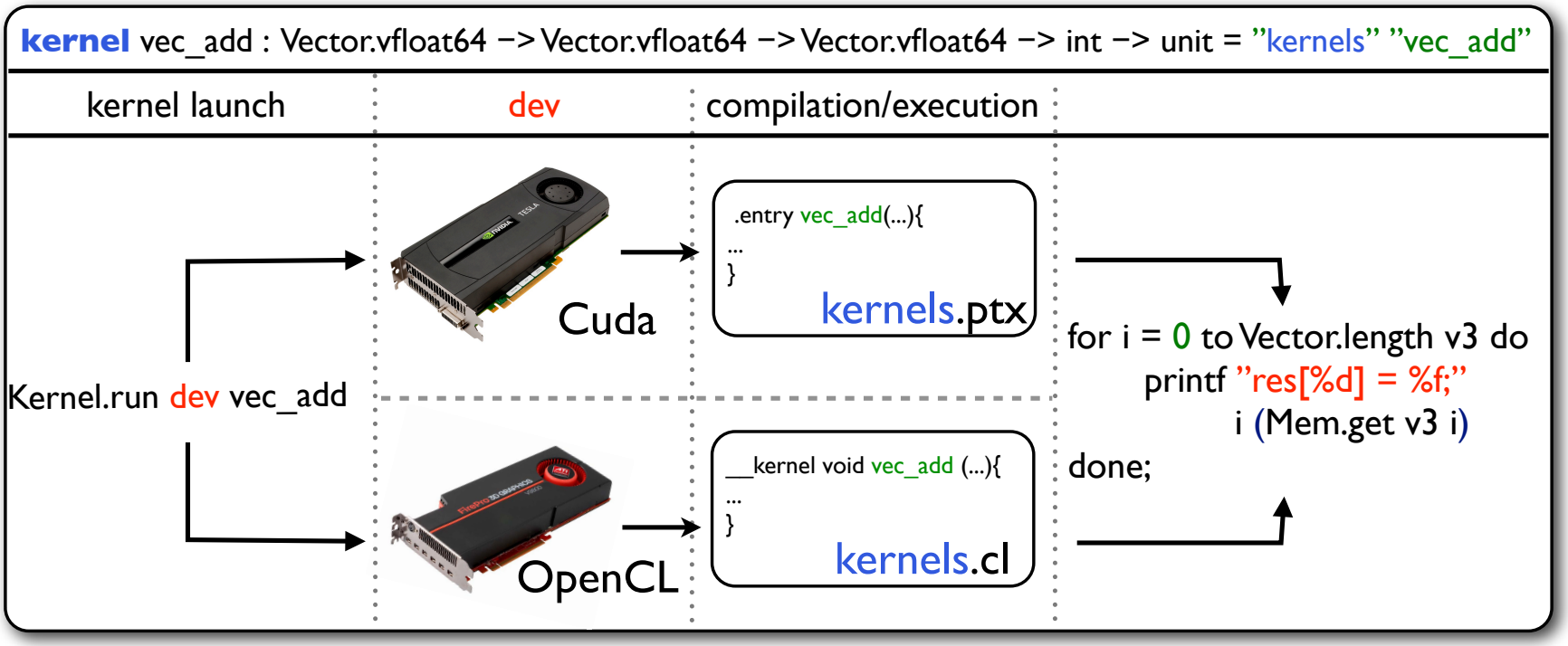
OCaml+SPOC Code

```
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)



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

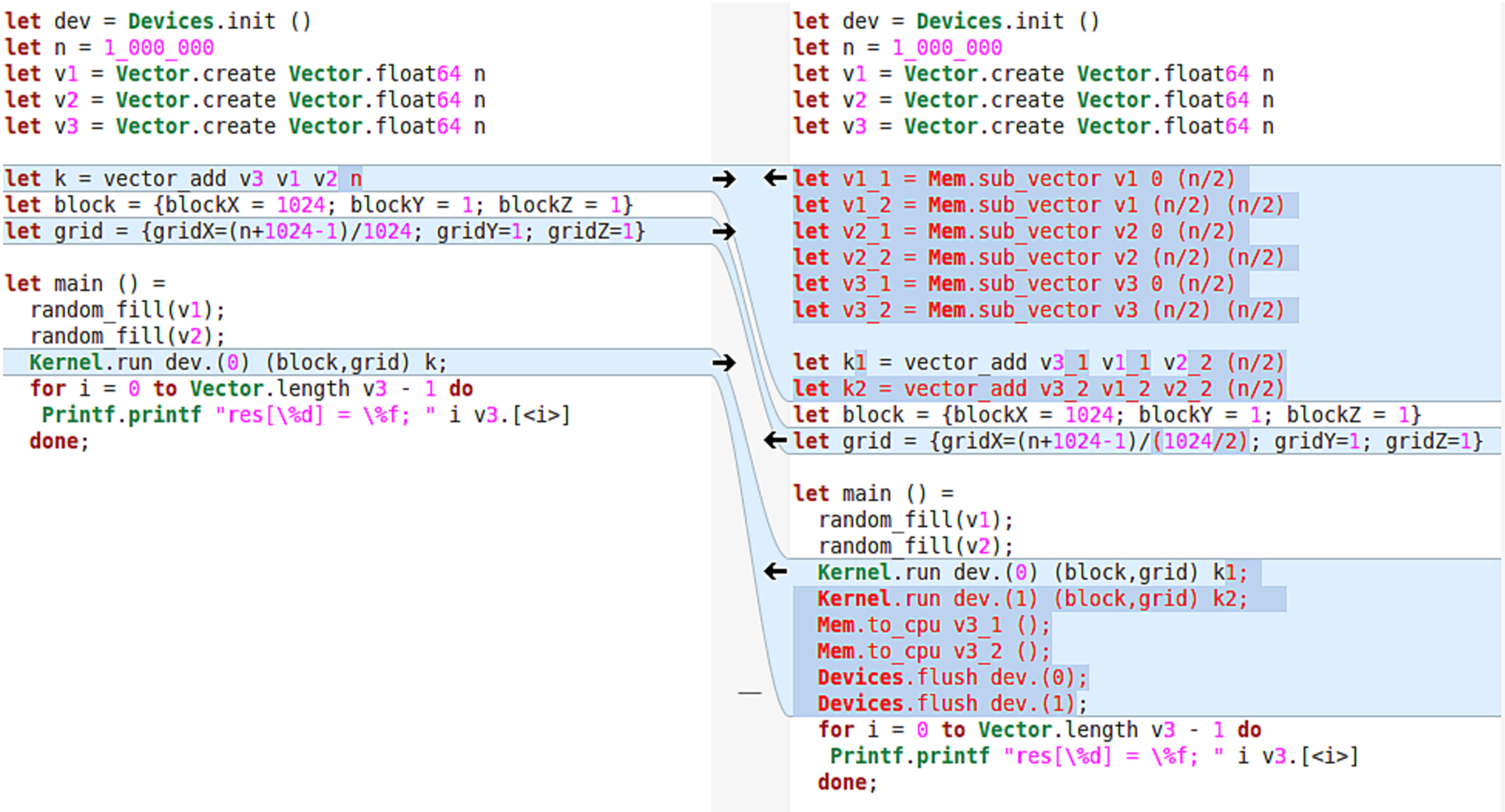
Tools

Development tools : OCaml Interactive Toplevel

mathias@vpaa: ~/workspace/Spoc — 80×21

```
#
$ ./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)
#
```

MultiGPU?


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 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 - 1 do
Printf.printf "res[\%d] = \%f; " i 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

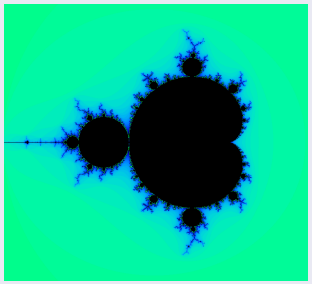
- Automatic Transfers **from CPU to Device**
- Automatic Transfers **from Device to CPU**
- Automatic Transfers **from Device to Device**

Benchmarks - 1

Spoc easily speeds OCaml programs up

Mandelbrot

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



Mandelbrot

	Intel i7	AMD 6950	NVIDIA Tesla C2070	C2070+6950	C2070+6950
	-	OpenCL	Cuda	Cuda+OpenCL	OpenCL+OpenCL
OCaml (+SPOC)	1252s	12.84s	10.99s	6.56s	6.66s
Speedup	1	97.50	113.92	190.85	187.98

opencl kernel not vectorized

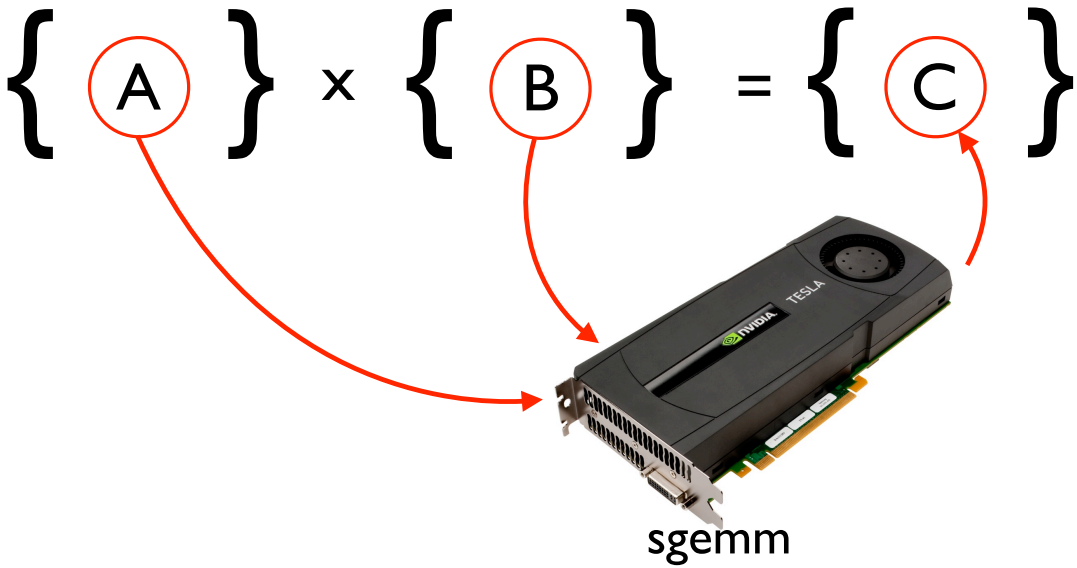
Benchmarks - 2

OCaml+Spoc runtime+GC overhead

Matrix Multiply SP

- optimized kernel
  - Nvidia $\rightarrow$ Cublas sgemm

Matrix Multiply SP



Benchmarks - 2

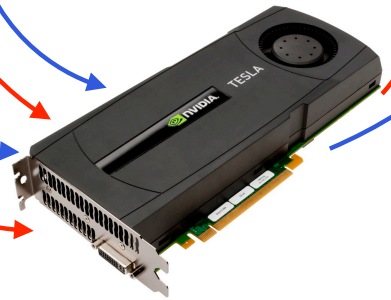
OCaml+Spoc runtime+GC overhead

Matrix Multiply SP

- optimized kernel
  - Nvidia $\rightarrow$ Cublas sgemm

Matrix Multiply SP

$$\begin{Bmatrix} A_1 & B_1 \\ C_1 & D_1 \end{Bmatrix} \times \begin{Bmatrix} A_2 & B_2 \\ C_2 & D_2 \end{Bmatrix} = \begin{Bmatrix} A_1A_2 + B_1C_2 & A_1B_2 + B_1D_2 \\ C_1A_2 + D_1C_2 & C_1B_2 + D_1D_2 \end{Bmatrix}$$



sgemm

Benchmarks - 2

OCaml+Spoc runtime+GC overhead

Matrix Multiply SP

- optimized kernel
  - Nvidia $\rightarrow$ Cublas sgemm

Matrix Multiply SP

	Matrix Multiply 1	Matrix Multiply 2	Matrix Multiply 2
Matrix size	21000	21000	25000
Maximum memory needed	5.2GB	5.2GB	7.5GB
GFLOPS	156	156	139

Conclusion - SPOC

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+Spoc 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