OCaml on Multicore CPUs with SPOC and FastFlow

Mathias Bourgoin

08.09.2016

Outline

Multicores and OCaml

OCaml cannot run parallel threads...

Multiple « solutions » have been considered :

- Extension for distributed computing ⇒ Caml/MPI (**1997**), BSML (**1998**), JoCaml (**2004**), MultiML (**2014-2015**) ?
- New runtime/GC ⇒ **OC4MC** (**2008**), multicore-ocaml (**2013**) ?
- Automatic/managed forking ⇒ ParMap (**2012**), Async.Parallel (**2014**) ?
- Probably many other solutions (new compiler ?, parallel virtual machine ?, etc)

...yet

« *As for the next major release, it will include a number of new features (including, if all goes well, support for multicore). It will be released some time around the end of this year.* »

– Damien Doligez for the OCaml development team, 2015.05.06

Another solution : GPGPU programming on CPUs

General Purpose computing with GPUs

- Use graphics processors to handle general putpose computations
- In theoery : general purpose = anything that is commonly handled by a CPU
- In practice : intensive computations (mostly data-parallel)

- Two frameworks : CUDA & OpenCL

GPGPU and CPUs ?

- OpenCL implementations exist for CPUs (Intel, AMD, ARM, IBM, ...)
- Benefits from multicore and (sometimes) vectorization
- Many limitations (as we will see later...)

GPGPU Programming

Two main frameworks

- **Cuda** (NVidia)
- **OpenCL** (Consortium OpenCL)

Different languages

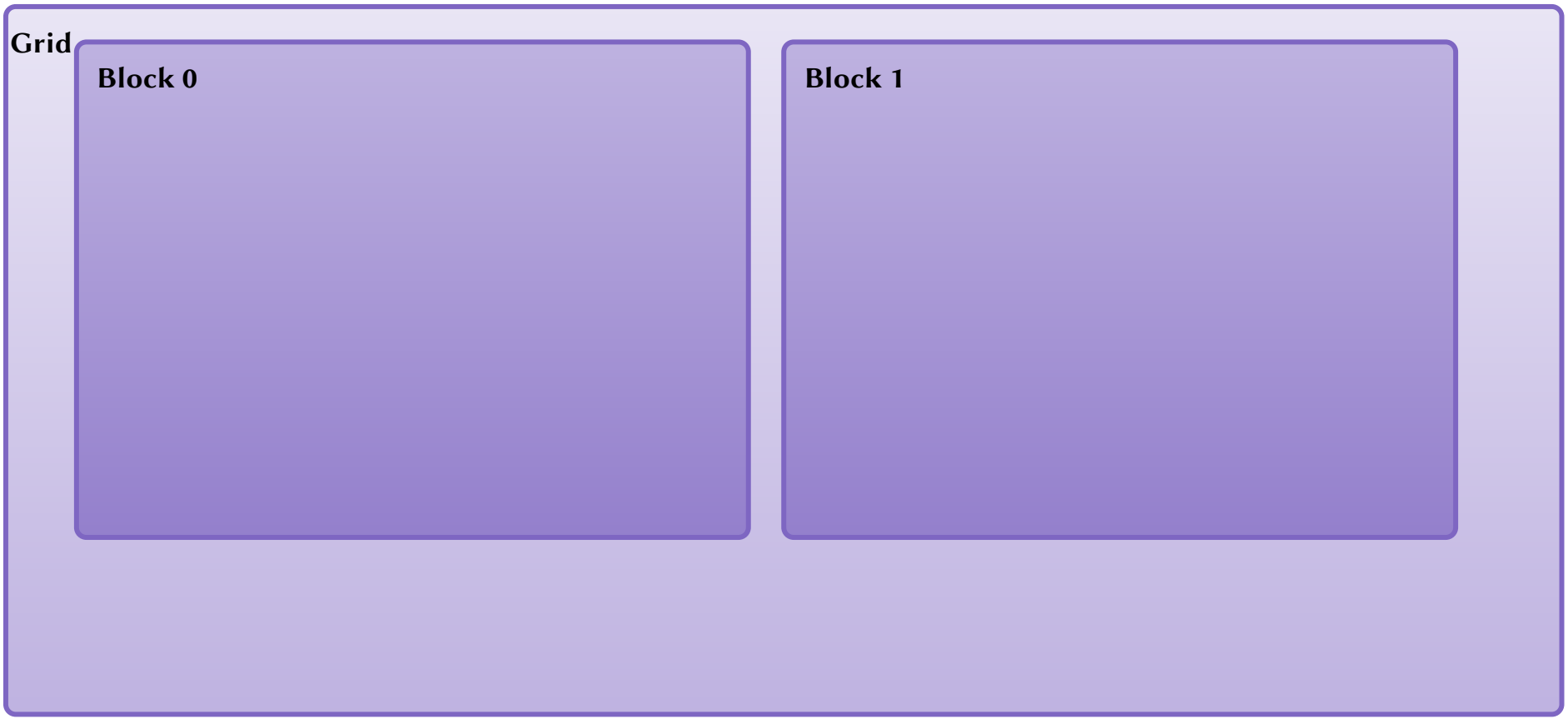
- To write kernels
  - **Assembly** (PTX, SPIR, IL,...)
  - Subsets of **C/C++**
- To manage kernels
  - *C/C++/Objective-C*
  - Bindings : Fortran, Python, Java, ...

Stream Processing

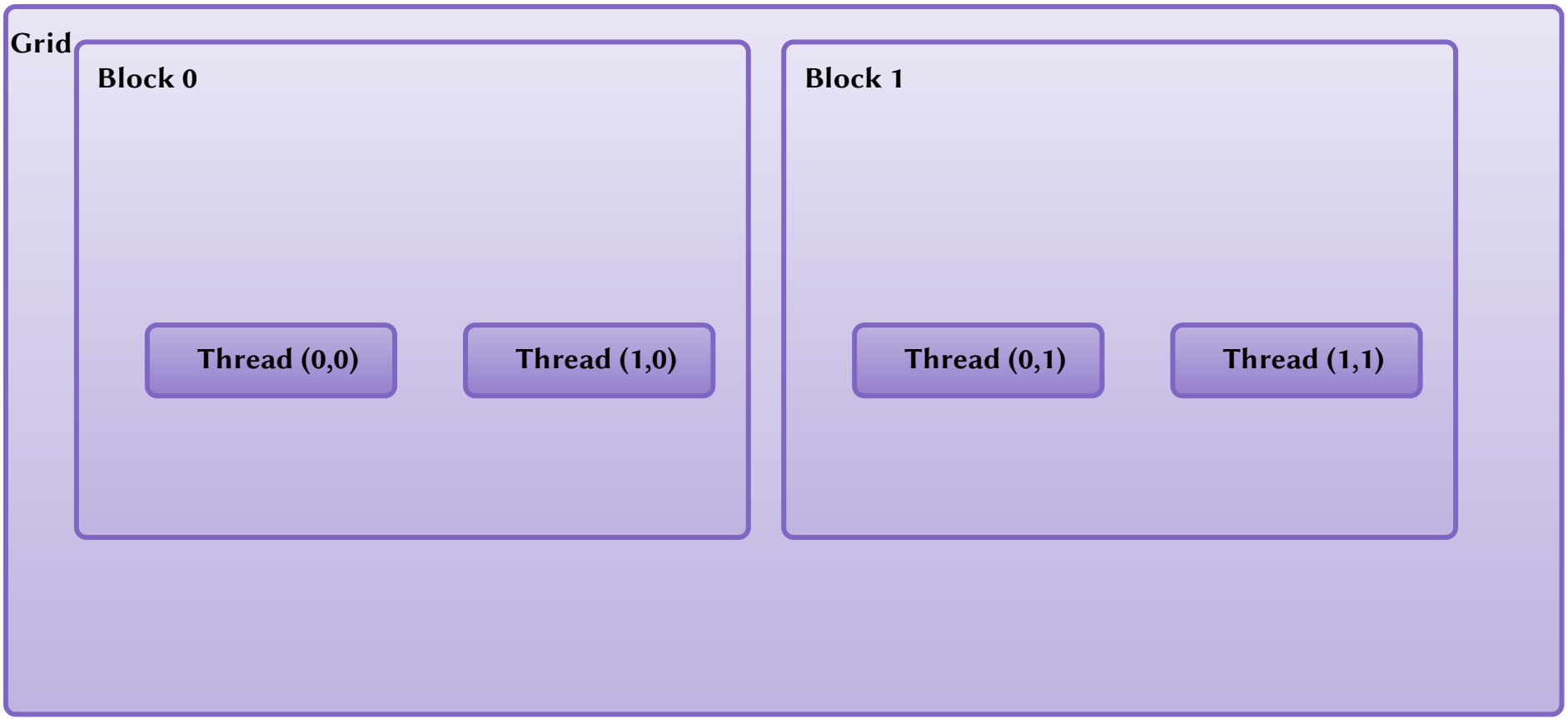
From a data set (stream), a series of computations (kernel) is applied to each element of the stream.

GPGPU programming in practice 1

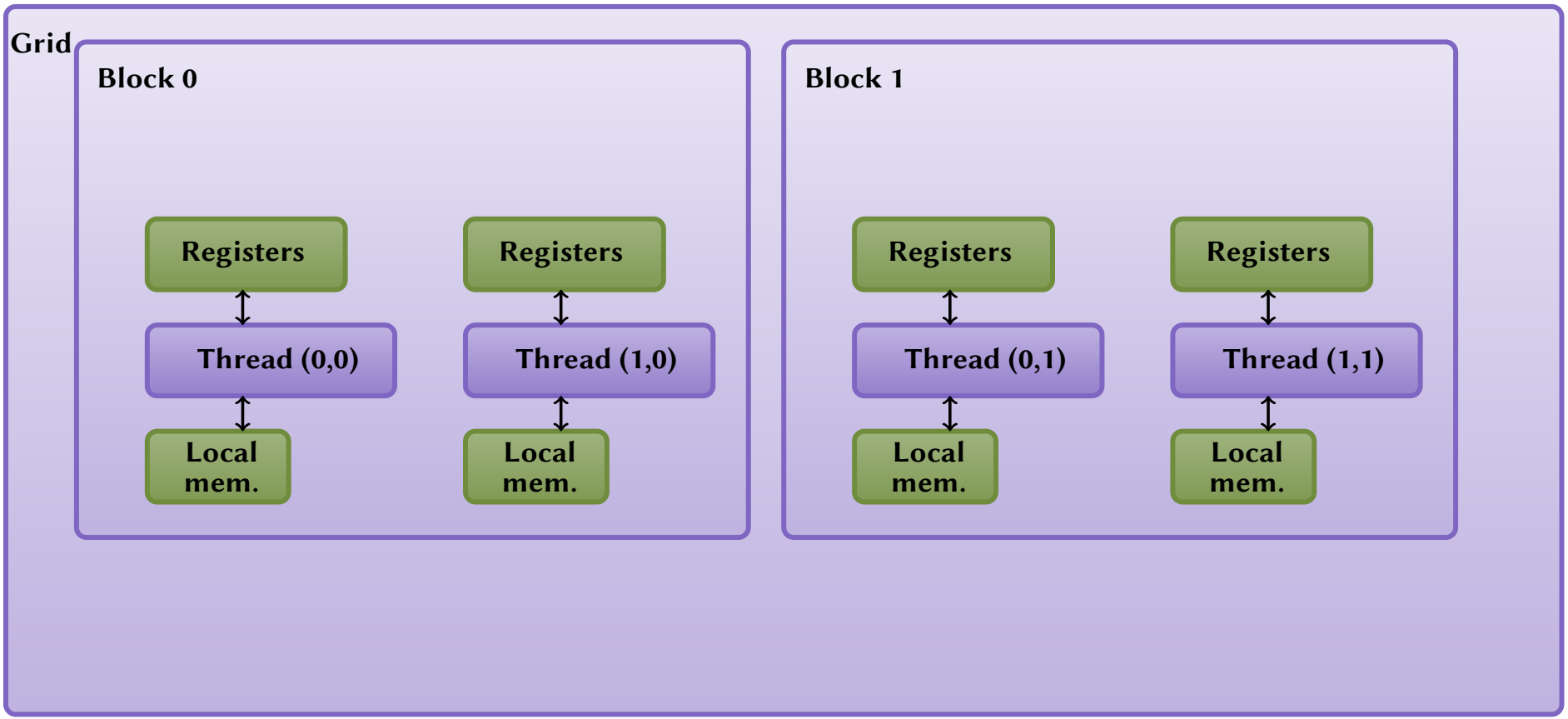
GPGPU programming in practice 1



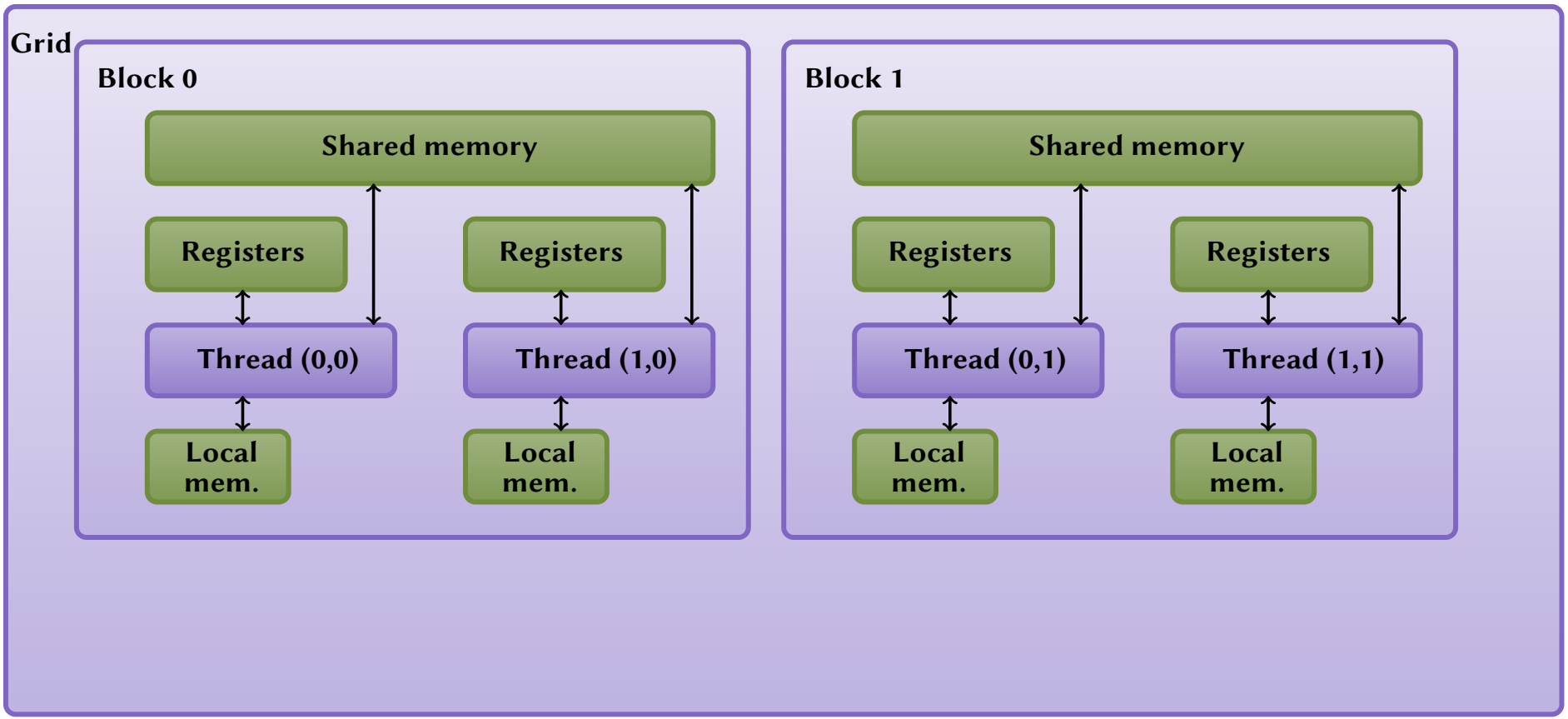
GPGPU programming in practice 1



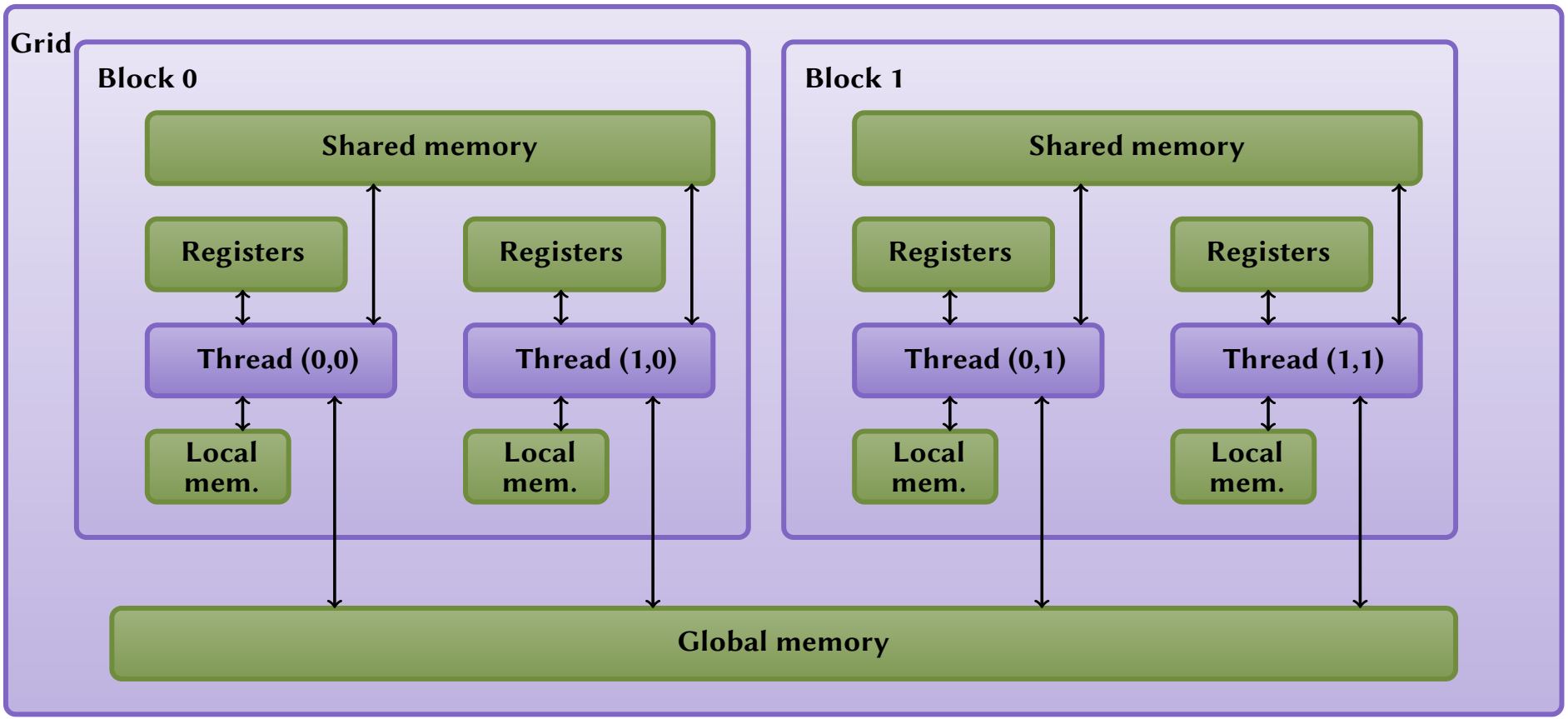
GPGPU programming in practice 1



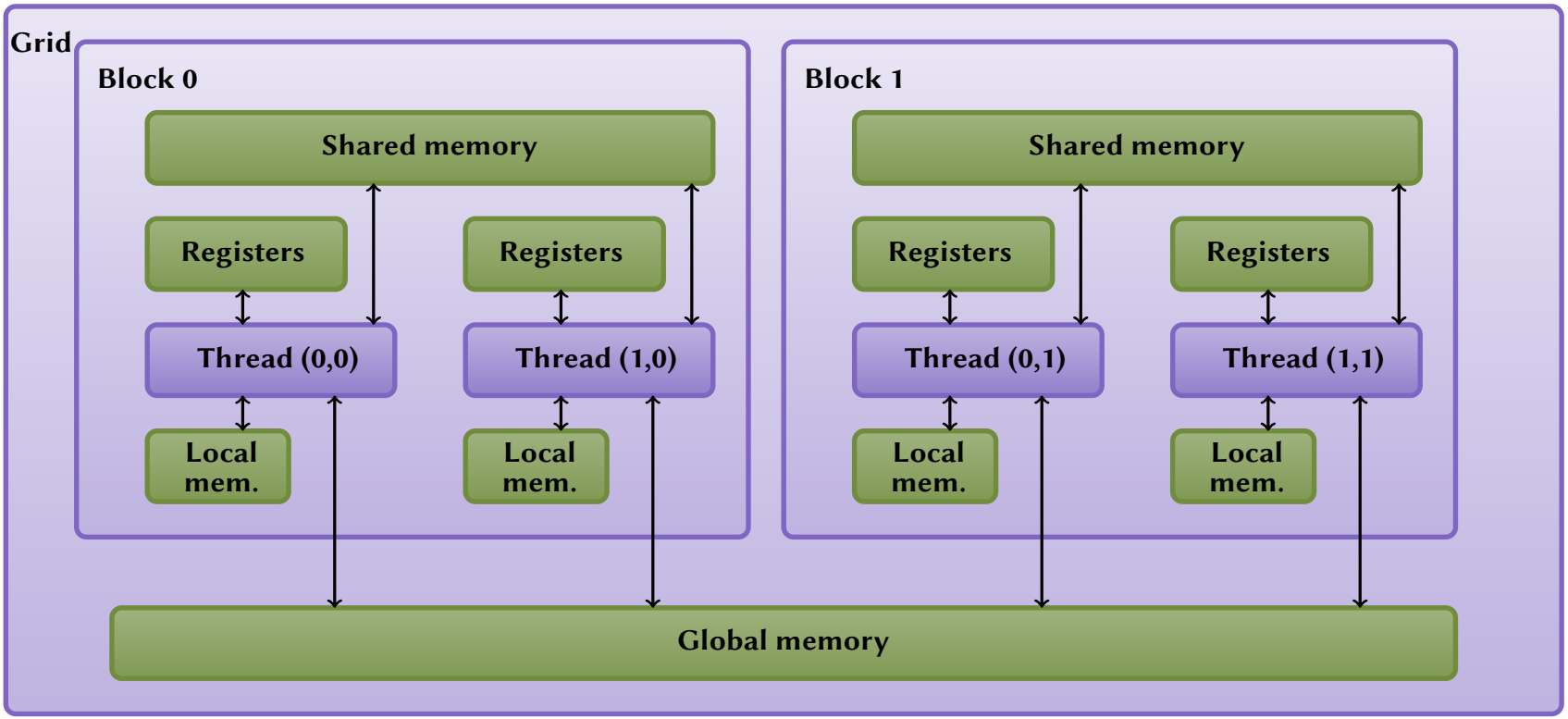
GPGPU programming in practice 1



GPGPU programming in practice 1



GPGPU programming in practice 1



Do not forget transfers between the host and its guests

	CPU-X86 Xeon E7 v3	GPU Mobile GTX 980M	GPU Gamer		GPU HPC Tesla K40
			GTX 980 Ti	Radeon R9 Fury X	
Memory bandwidth	102GB/s	160GB/s	336.5GB/s	512GB/s	288GB/s

PCI-Express 3.0 maximum bandwidth is 16GB/s

PCI-Express 4.0 (not before 2017) 32GB/s

GPGPU programming in practice 2

Kernel : small example using OpenCL

Vector addition

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

GPGPU programming in practice 2

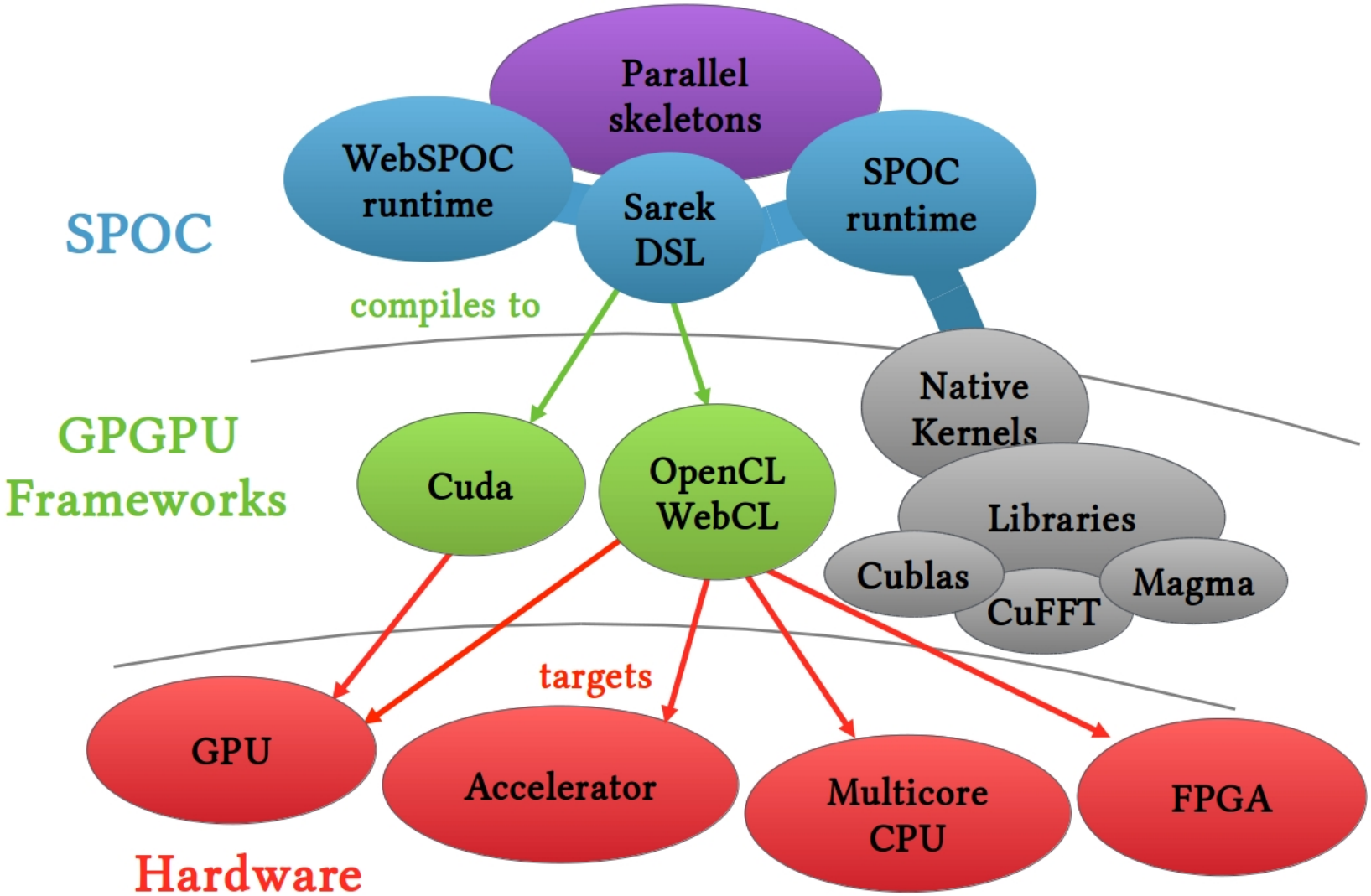
Host : small example using C

```
// 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);
```

GPGPU Programming with OCaml



SPOC overview

Abstract frameworks

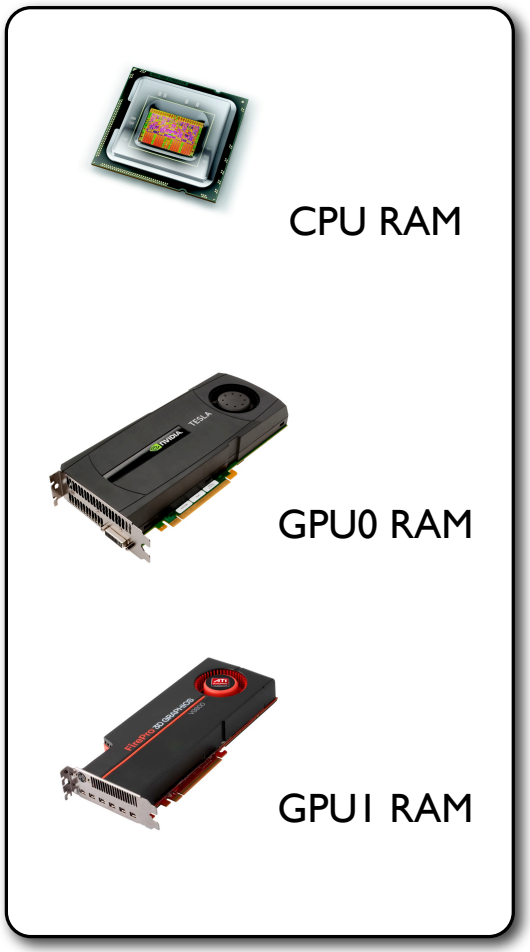
- Unify both APIs (Cuda/OpenCL), **dynamic linking**.
- Portable solution, multi-GPGPU, heterogeneous

Abstract transfers

Vectors move automatically between CPU and GPGPUs

- On-demand (lazy) transfers
- **Automatic allocation/dealloction** of the memory space used by vectors (on the host as well as on GPGPU devices)
- Failure during allocation on a GPGPU triggers a garbage collection

A small example



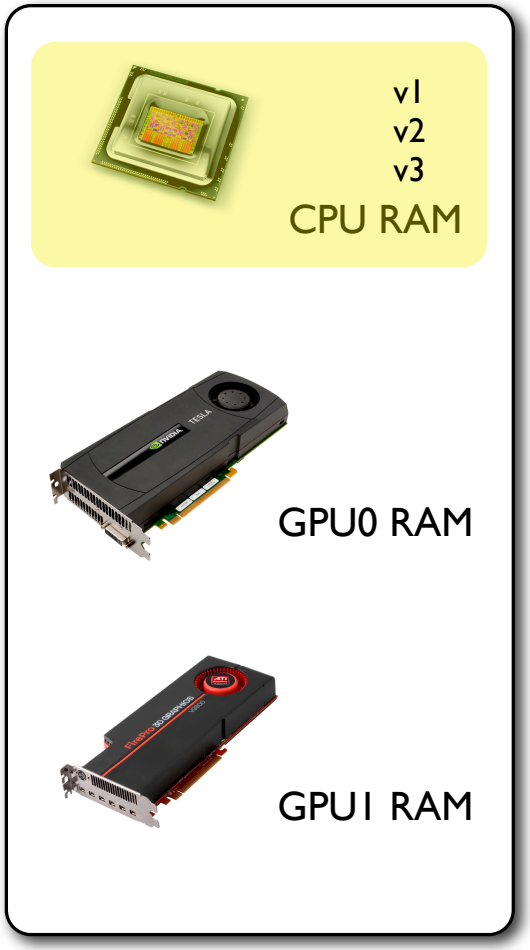
Example

```
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 = vec_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 small example



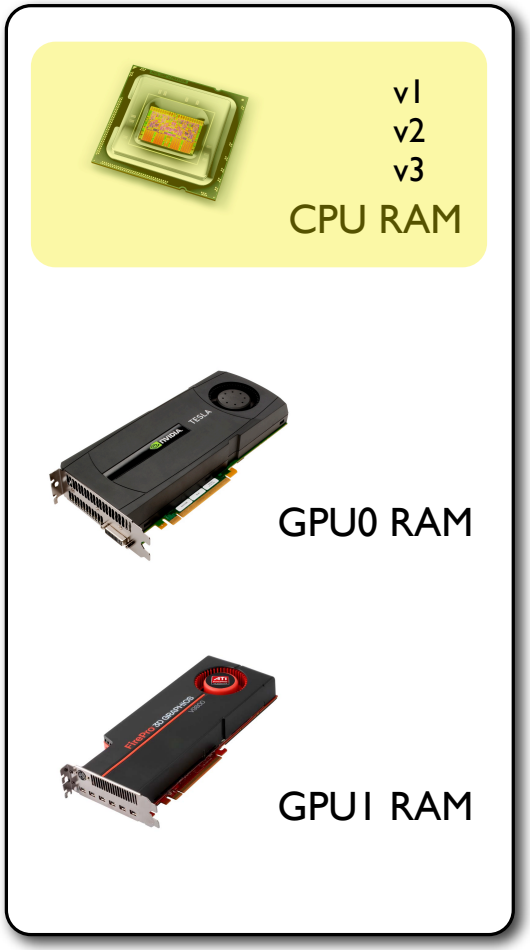
Example

```
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 = vec_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 small example



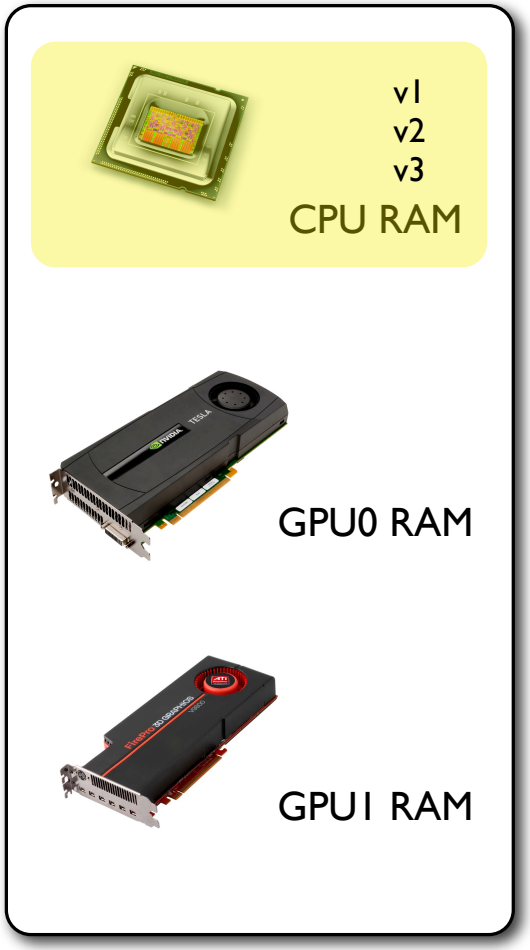
Example

```
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 = vec_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 small example



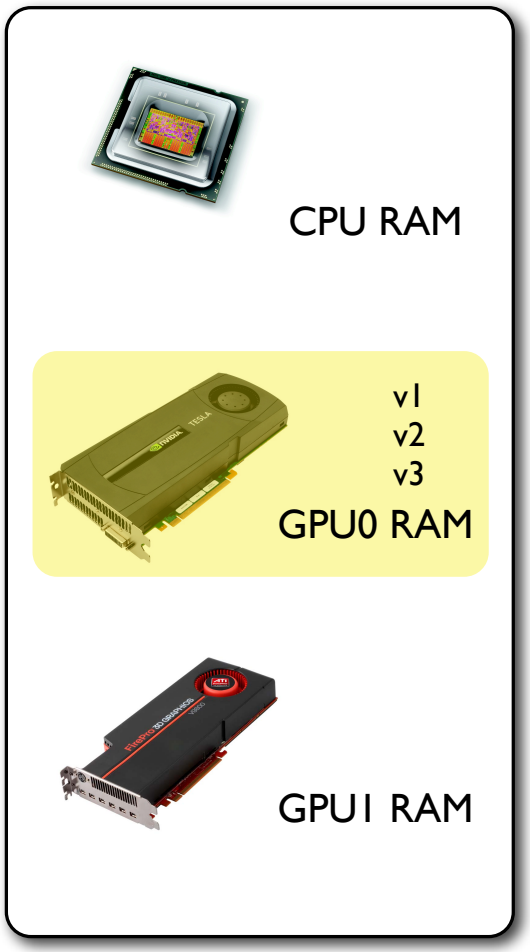
Example

```
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 = vec_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 small example



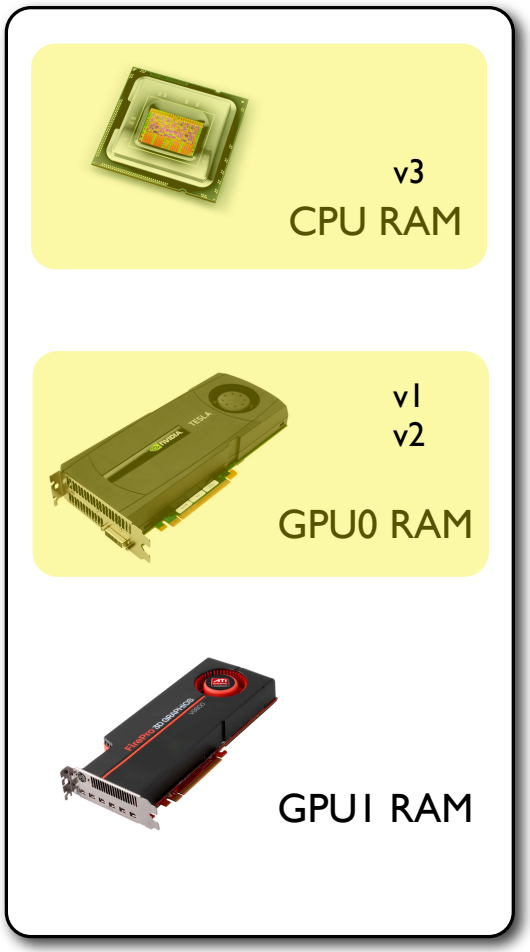
Example

```
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 = vec_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 small 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 = vec_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;
```

Sarek : Stream ARchitecture using Extensible Kernels

Vector addition with Sarek

```
let vec_add = kern a b c n ->
  let open Std in
  let open Math.Float64 in
  let idx = global_thread_id in
  if idx < n then
    c.[<idx>] <- add a.[<idx>] b.[<idx>]
```

Vector addition with OpenCL

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

Sarek

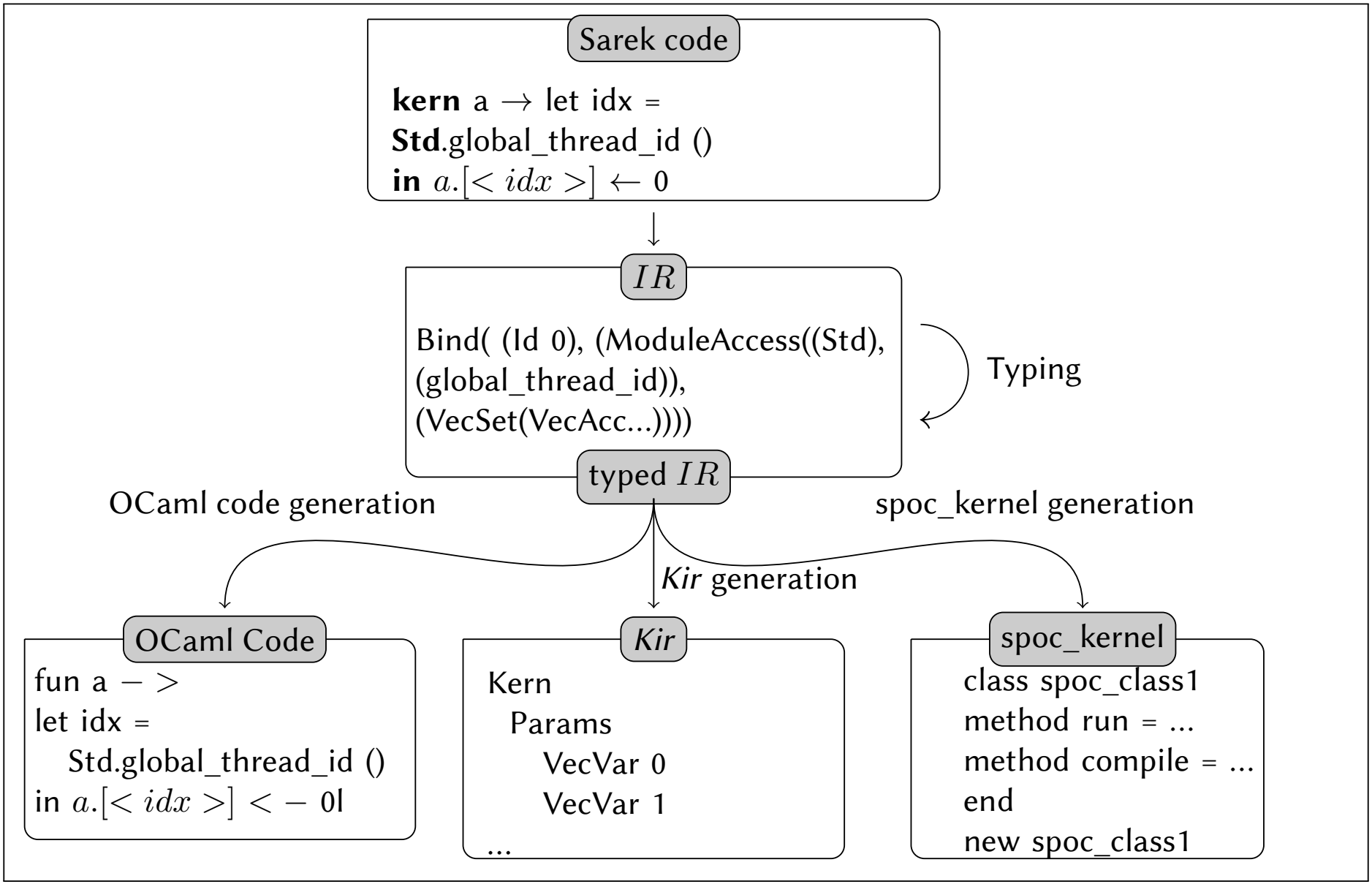
Vector addition with Sarek

```
let vec_add = kern a b c n ->
  let open Std in
  let open Math.Float64 in
  let idx = global_thread_id in
  if idx < n then
    c.[<idx>] <- add a.[<idx>] b.[<idx>]
```

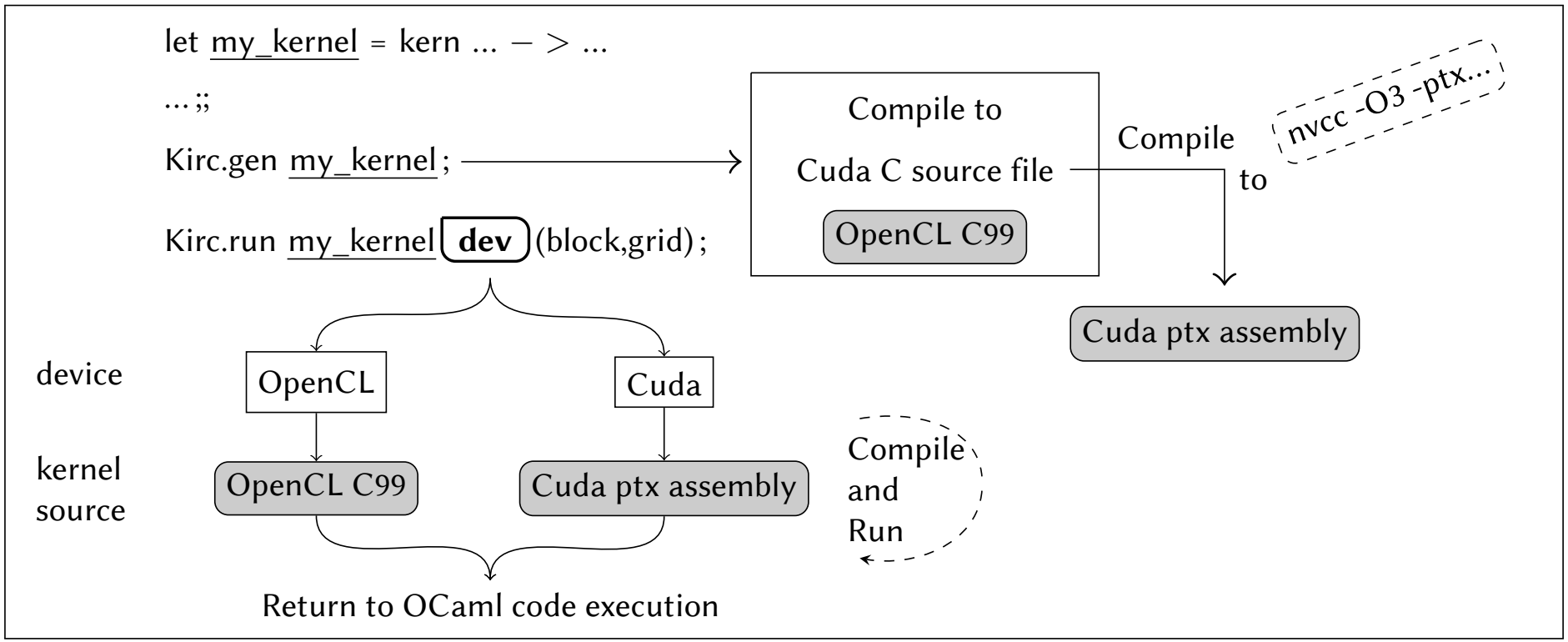
Sarek features

- ML-like syntax
- type inference
- static type checking
- **static** compilation to OCaml code
- **dynamic** compilation to Cuda/OpenCL

Sarek static compilation



Sarek dynamic compilation



Vectors addition

SPOC + Sarek

```
open Spoc
let vec_add = kern a b c n ->
  let open Std in
  let open Math.Float64 in
  let idx = global_thread_id in
  if idx < n then
    c.[<idx>] <- add 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;
```

- OCaml
- No explicit transfer
- Type inference
- Static type checking
- Portable
- Heterogeneous

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

Limitations

CPUs are used as GPUs

- Kernels are limited :
  - no pointers
  - no dynamic allocation
  - no recursion
  - ...
- Parallelism is limited to SIMT (Single Instruction Multiple Threads)
  - Branching limits performance...
- Complex memory model
- Hard to implement, optimize, debug
  (easier than with actual GPUs... thanks to `printf`)

Depends on OpenCL

- Different implementations = different features, bugs, performance
- What happens when no OpenCL implementation is installed ?

Solutions ?

Implement a fake GPU within SPOC

- In pure OCaml (no parallelism... yet)
- In OCaml + C

Use an existing library to manage parallelism

Many candidates :

- FastFlow
- StarPU
- Intel TBB
- ...

Solutions ?

Implement a fake GPU within SPOC

- In pure OCaml (no parallelism... yet)
- In OCaml + C

Use an existing library to manage parallelism

Many candidates :

- **FastFlow**
- StarPU
- Intel TBB
- ...

FastFlow

What/Why

Parallel programming framework

- Algorithmic skeleton approach
- C++ based
- Streaming ”process” networks
- Targeting heterogenous platforms composed of clusters of shared-memory platforms
  $\rightarrow$possibly equipped with computing accelerators
- Extremely efficient in inter-concurrent activity communication (in the range of 10nsecs)
- Better or comparable efficiency/performance w.r.t. ”standard” programming environments (OMP, Cilk, TBB)

FastFlow

Who

Core team

- Massimo Torquati – Parallel Programming Models Group, Computer Science Department, University of Pisa, Italy
- Marco Aldinucci – Parallel computing group, Computer Science Department, University of Torino, Italy
- **Marco Danelutto** – Parallel Programming Models Group, Computer Science Department, University of Pisa, Italy

+ Many contributors

`http://calvados.di.unipi.it/`

Skeleton (pattern ?) based approach

Abstract parallelism exploitation pattern by parametric code

- E.g. higher order function, code factories, C++ templates, ...
- Can be composed and nested as language constructs + offloading
- Stream and Data Parallel

Platform independent

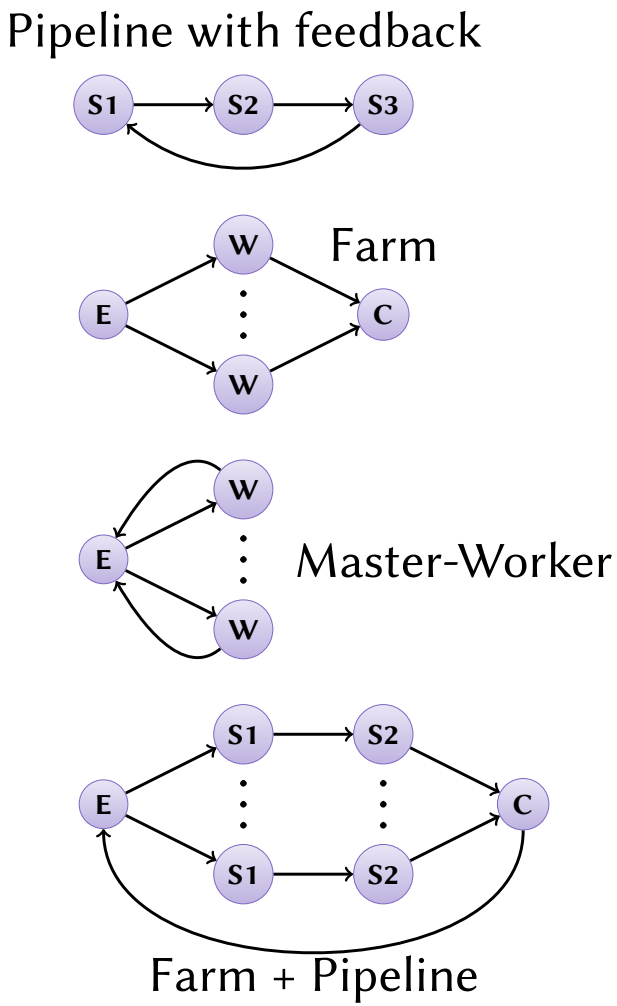
- Implementations on different multi/many-cores
- Support for hybrid architectures thanks to pattern compositionality

Provide state-of-the-art, parametric implementation of each parallelism exploitation skeleton

- With natural way of extending patterns, i.e. OO
- Functional (seq code) and tunable extra-functional (QoS) parameters

Stream Parallel Skeletons

- Pipeline
  arbitrary number of stages
- Farm
  Emitter-StringOfWorkers-Collector
  or Master/Worker
- Feedback channel
  feeds back (partial) results from last stage output to first stage input
- Free composition
  quite complex process graphs



Accelerator Interface

Standard streaming programs

- First stage generates the input stream
- Middle stage compute results
- Last stage outputs results

Accelerator mode

- Create skeleton program
- Offload tasks to be computed when needed
- Get results asynchronously
- Uses ”spare” cores

```
20 // FastFlow accelerated code
21 #define N 1024
22 long A[N][N],B[N][N],C[N][N];
23 int main() {
24   // < init A,B,C>
25
26   ff::ff_farm<> farm(true /* accel */);
27   std::vector<ff::ff_node *> w;
28   for(int i=0;i<PAR_DEGREE;++i)
29     w.push_back(new Worker);
30   farm.add_workers(w);
31   farm.run_then_freeze();
32
33   for (int i=0;i<N;i++) {
34     for(int j=0;j<N;++j) {
35       task_t * task = new task_t(i,j);
36       farm.offload(task);
37     }
38   }
39   farm.offload((void *)ff::FF_EOS);
40   farm.wait(); // Here join
41 }
42
43 // Includes
44 struct task_t {
45   task_t(int i,int j):i(i),j(j) {}
46   int i; int j;};
47
48 class Worker: public ff::ff_node {
49 public: // Offload target service
50   void * svc(void *task) {
51     task_t * t = (task_t *)task;
52     int _C=0;
53     for(int k=0;k<N;++k)
54       _C += A[t->i][k]*B[k][t->j];
55     C[t->i][t->j] = _C;
56     delete t;
57     return GO_ON;
58     }
59 };
```

OCaml-Fastflow

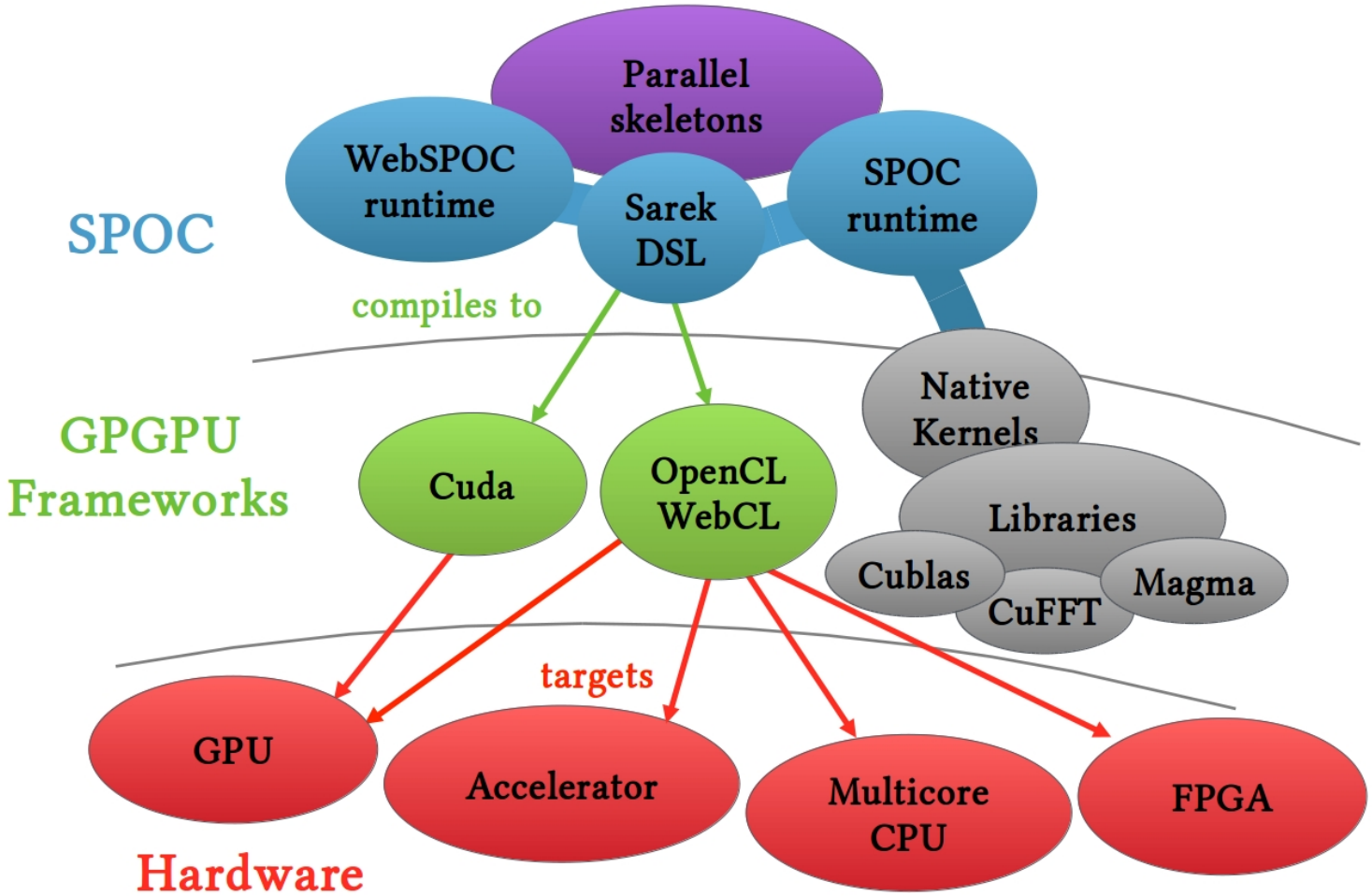
What/Why

- OCaml binding to the fastflow framework
- Based on SPOC and Sarek
- To solve the constraints imposed by GPGPU programming on CPU
  - Independant from OpenCL drivers
  - Classic memory model
  - Access to CPU libraries
  - Performance independant of branching divergence
- Benefit from OCaml to define/compose computations

Implementation based on Sarek

Idea

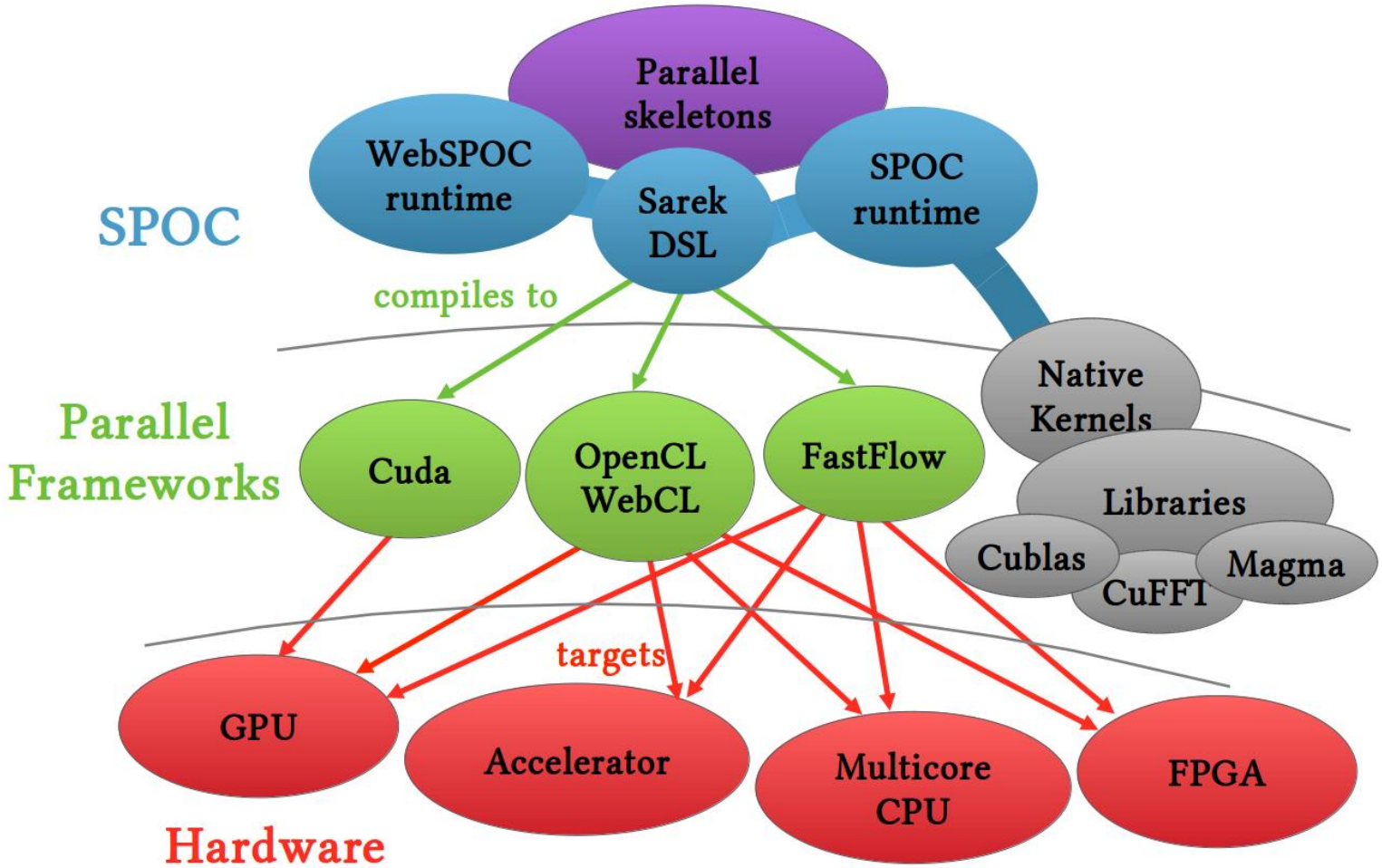
Sarek kernels are translated to Cuda/OpenCL C-like functions



Implementation based on Sarek

Idea

Sarek kernels are translated to Cuda/OpenCL C-like functions
$\rightarrow$ Let's add a new target !



First implementation

New modules

1. Minimal OCaml library dynamically loading C functions
2. New Sarek target : C

At compile-time

1. Generate Kernel Internal representation (KIR) from Sarek(as with GPGPU kernels)

At run-time

1. Compile KIR to minimal C++ library embedding our function into using FastFlow objects
2. Generate C interface to the library
3. Compile everything as a shared library
4. Use the C library to dynamically load and run Fastflow calls

First implementation

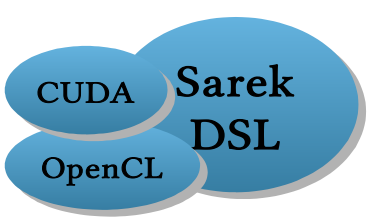
Problem

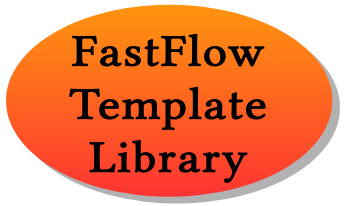
- FastFlow is a template library
- Compiling a (small) library that depends on FastFlow takes too much time...

Solution

- Precompile C++
- Only generate tasks as C functions
- Dynamic loading from C++

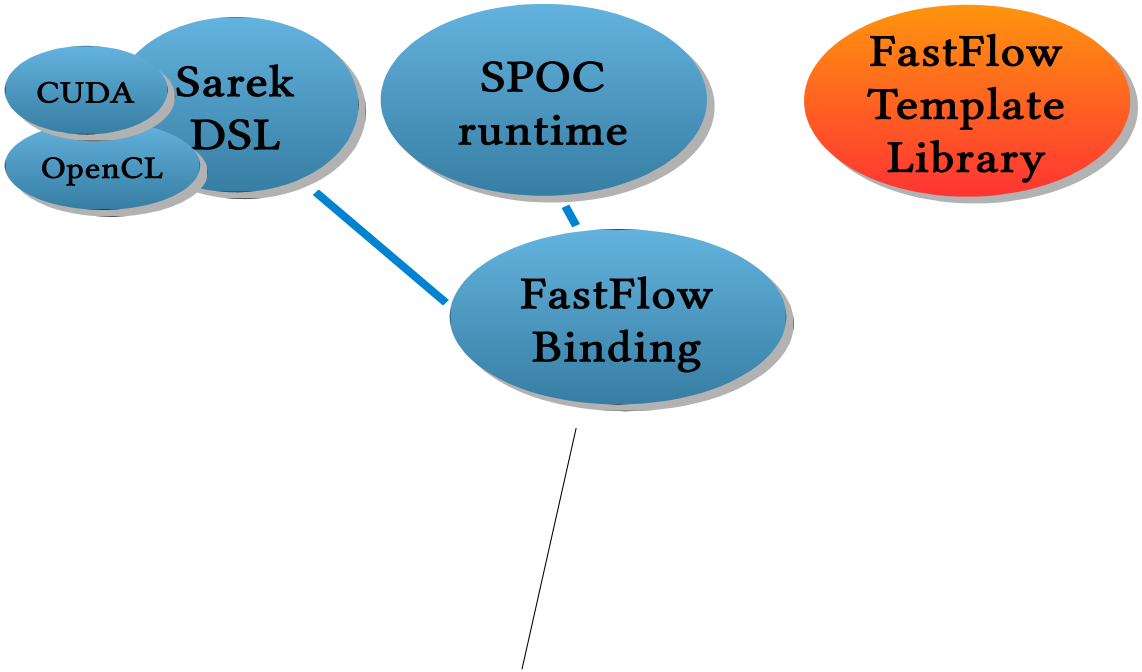
Second implementation





OCaml C++

Second implementation - New Components



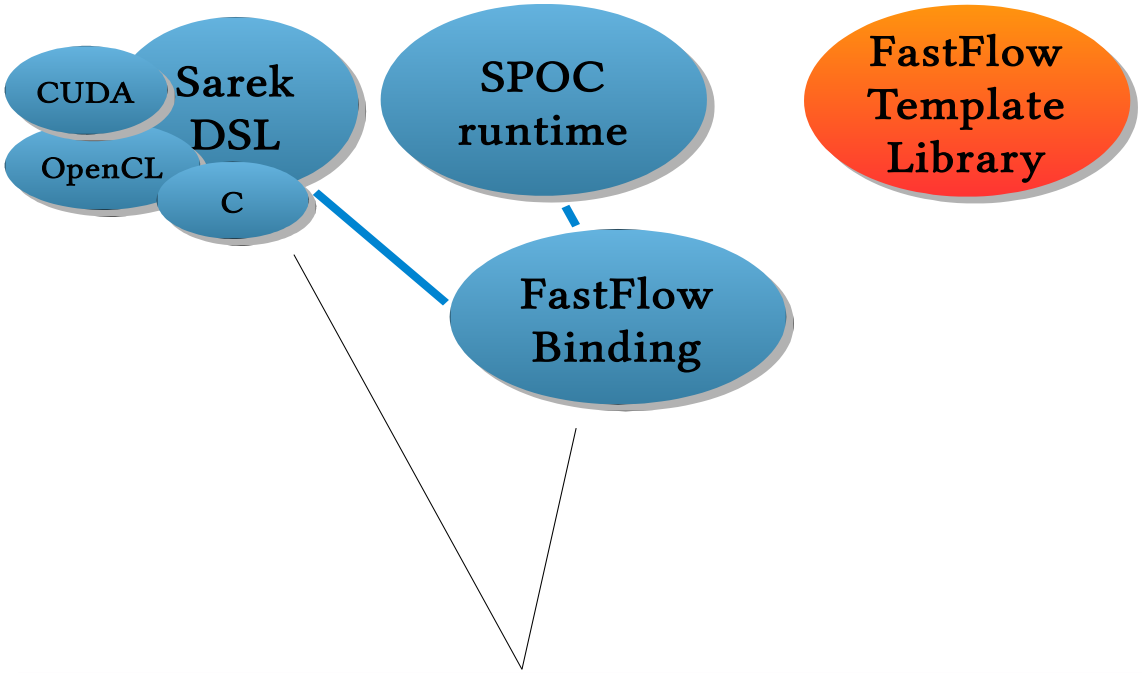
OCaml C++

New components

1. OCaml-FastFlow binding
2. New Sarek target : C
3. Minimal C++ library (+ C interface)

OCaml-FastFlow binding library that dynamically compiles Sarek kernels and loads associated C functions

Second implementation - New Components



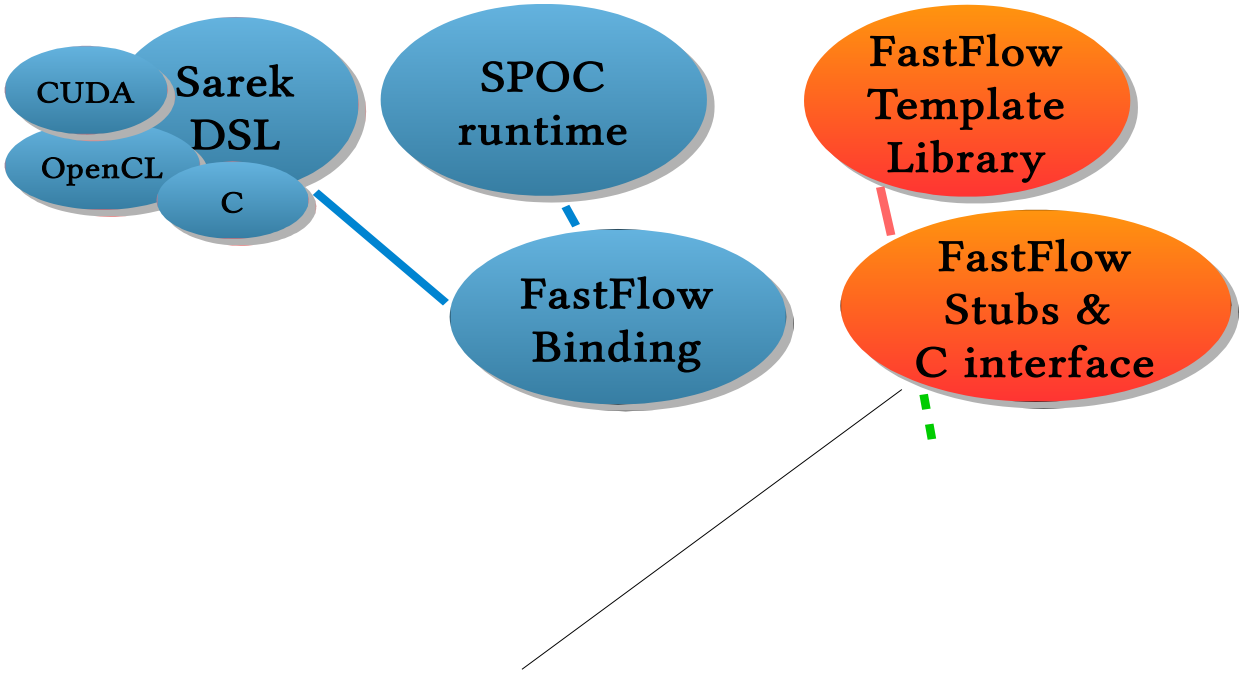
OCaml C++

New components

1. OCaml-FastFlow binding
2. New Sarek target : C
3. Minimal C++ library (+ C interface)

New Sarek target : C

Second implementation - New Components



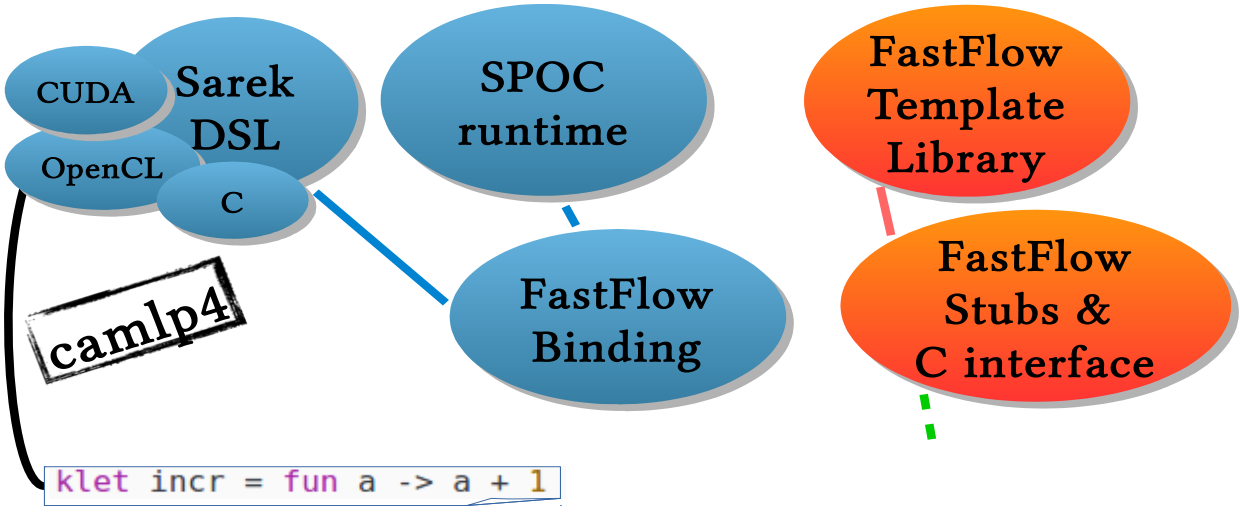
OCaml C++

New components

1. OCaml-FastFlow binding
2. New Sarek target : C
3. Minimal C++ library (+ C interface)

Minimal C++ library (+ C interface) using FastFlow and dynamically loading future C functions

Second implementation - Compile-time



OCaml C++

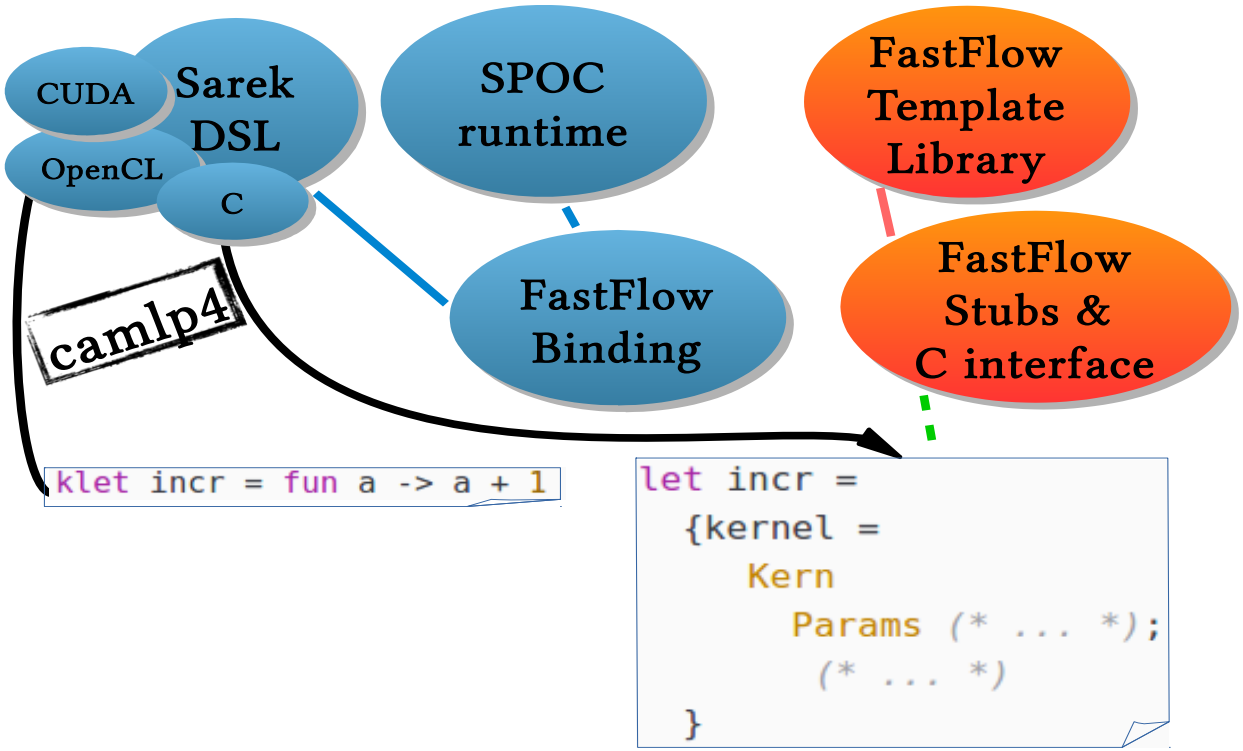
New components

1. OCaml-FastFlow binding
2. New Sarek target : C
3. Minimal C++ library (+ C interface)

At compile-time

1. Generate internal representation
2. Generate a Task module for each kernel

Second implementation - Compile-time



OCaml C++

New components

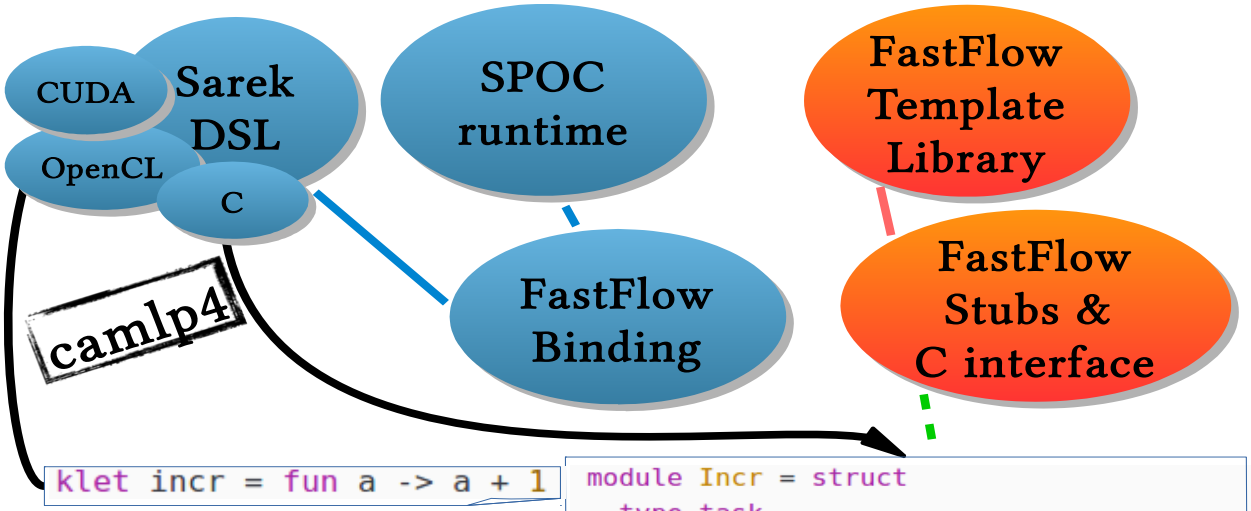
1. OCaml-FastFlow binding
2. New Sarek target : C
3. Minimal C++ library (+ C interface)

At compile-time

1. Generate internal representation
2. Generate a Task module for each kernel

Generate kernel internal representation (as with GPGPU) from Sarek

Second implementation - Compile-time



OCaml C++

```
klet incr = fun a -> a + 1
```

```
module Incr = struct
  type task
  (* ... *)
  let offload = Foreign.foreign (* ... *)
end
let incr =
  {kernel =
     Kern
       Params (* ... *);
   fastflow_acc = object
     method offload = Incr.offload
    (* ... *)
   end
  }
```

New components

1. OCaml-FastFlow binding
2. New Sarek target : C
3. Minimal C++ library (+ C interface)

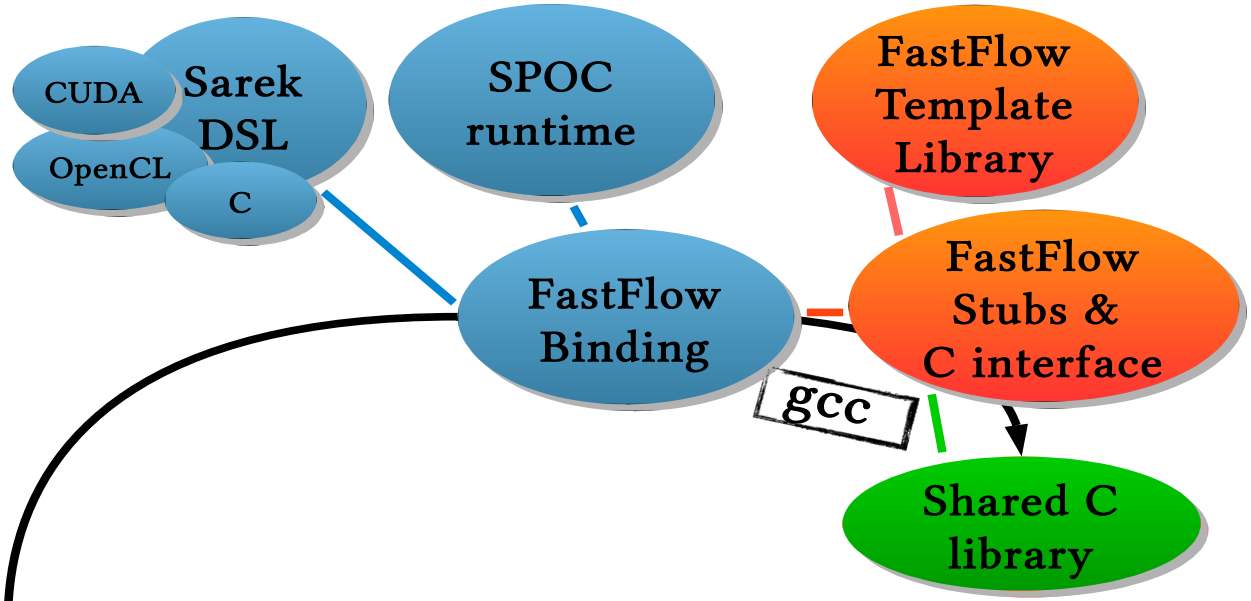
At compile-time

1. Generate internal representation
2. Generate a Task module for each kernel

Generate a Task module for each kernel

- using types information that is lost at run-time
- that dynamically loads C functions from the library

Second implementation - Run-time



OCaml C++ C

```
klet incr = fun a b c start_ end_ ->
  $"printf(\"Task : %d to %d starts\n\", start_, end_)"$;
  for i = start_ to end_ do
    c.[<i>] <- a.[<i>] + b.[<i>]
  done
(* ...  *)
let accelerator = FastFlow.to_farm incr 10 in
(* ... *)
accelerator#offload_task (v1, v2, vres, 0, 511);
(*... *)
accelerator#wait
```

Generate minimal C function from the internal representation
Compile as a shared library

New components

1. OCaml-FastFlow binding
2. New Sarek target : C
3. Minimal C++ library (+ C interface)

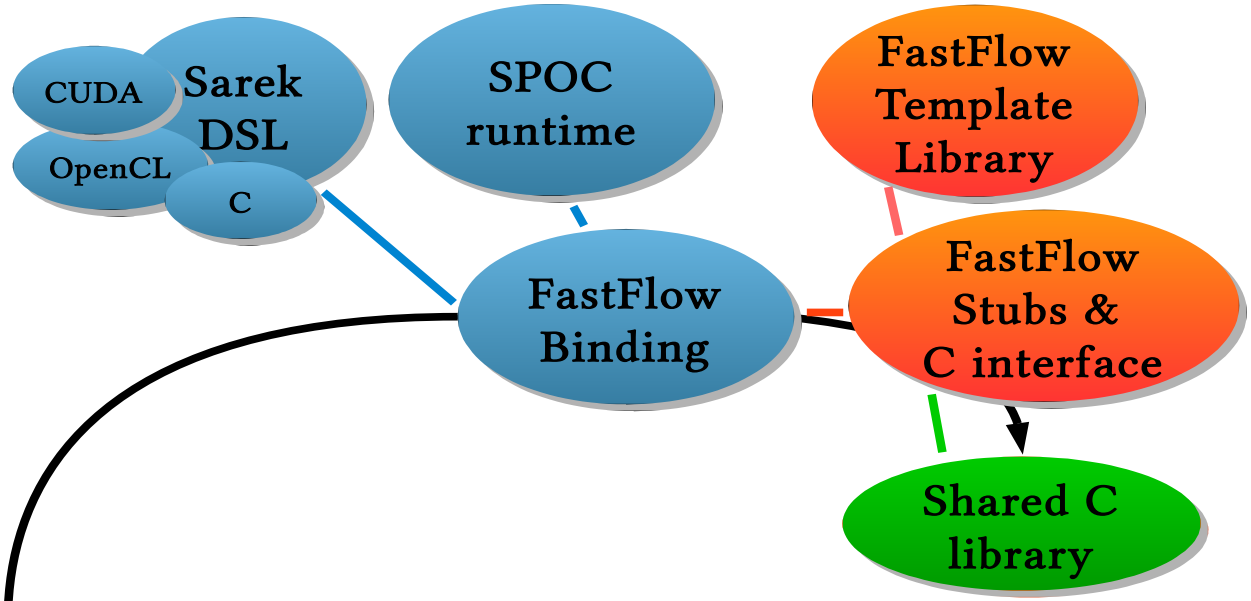
At compile-time

1. Generate internal representation
2. Generate a Task module for each kernel

At run-time

1. Generate minimal C function from the internal representation
2. Compile as a shared library
3. Use the Task module to offload existing C functions
4. Wait and hope for performance

Second implementation - Run-time



OCaml C++ C

```
klet incr = fun a b c start_ end_ ->
  $"printf(\"Task : %d to %d starts\n\", start_, end_)"$;
  for i = start_ to end_ do
    c.[<i>] <- a.[<i>] + b.[<i>]
  done
(* ...  *)
let accelerator = FastFlow.to_farm incr 10 in
(* ... *)
accelerator#offload_task (v1, v2, vres, 0, 511);
(*... *)
accelerator#wait
```

Use the previously (compile-time) generated Task module to offload existing C functions

New components

1. OCaml-FastFlow binding
2. New Sarek target : C
3. Minimal C++ library (+ C interface)

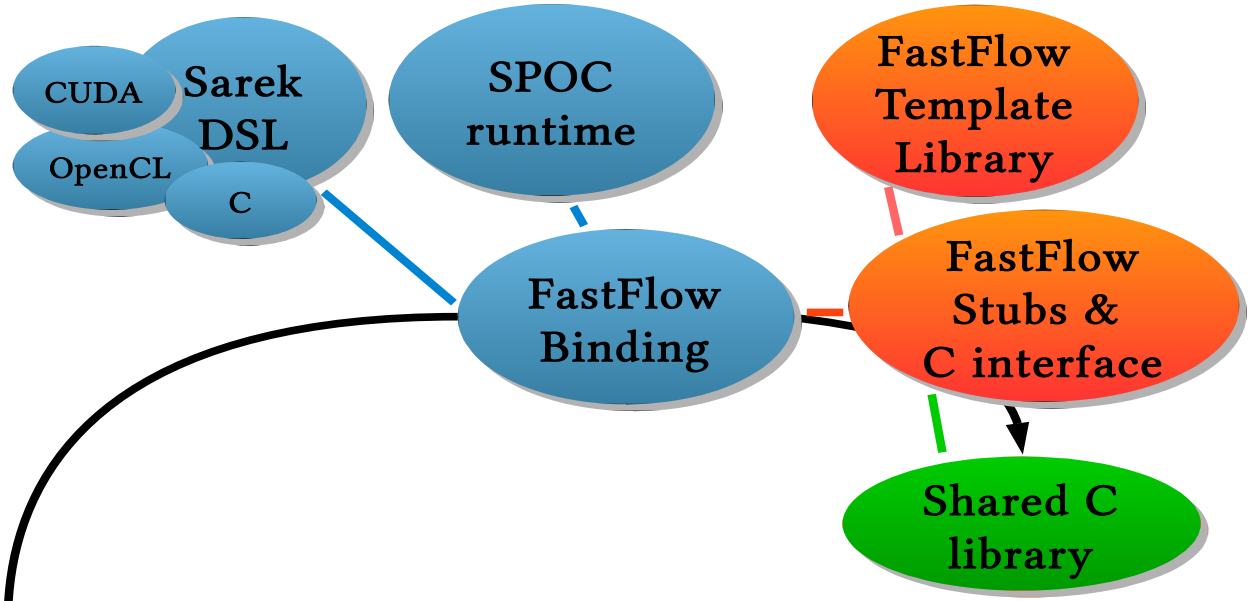
At compile-time

1. Generate internal representation
2. Generate a Task module for each kernel

At run-time

1. Generate minimal C function from the internal representation
2. Compile as a shared library
3. Use the Task module to offload existing C functions
4. Wait and hope for performance

Second implementation - Run-time




```
klet incr = fun a b c start_ end_ ->
  $"printf(\"Task : %d to %d starts\n\", start_, end_)"$;
  for i = start_ to end_ do
    c.[<i>] <- a.[<i>] + b.[<i>]
  done
(* ...  *)
let accelerator = FastFlow.to_farm incr 10 in
(* ... *)
accelerator#offload_task (v1, v2, vres, 0, 511);
(*... *)
accelerator#wait
```

Wait and hope for performance

OCaml C++ C

New components

1. OCaml-FastFlow binding
2. New Sarek target : C
3. Minimal C++ library (+ C interface)

At compile-time

1. Generate internal representation
2. Generate a Task module for each kernel

At run-time

1. Generate minimal C function from the internal representation
2. Compile as a shared library
3. Use the Task module to offload existing C functions
4. Wait and hope for performance

A small example

Compute function in Sarek

```
klet compute_sin = fun x ->
 let open Std in
 let open Math.Float32 in
 let mutable a = float x in
 for i = 0 to x  do
  a := sin a
 done;
 $"printf (\"worker : %d -> res %g\\n\",
              x, a)"$
```

Composition in OCaml

```
let _ =
  let sinFarm =
    FastFlow.to_farm compute_sin 10 in
  sinFarm#run_accelerator ();
  for i = 0 to size - 1 do
    sinFarm#offload_task i;
  done;
  sinFarm#no_more_tasks;
  for i = 0 to size - 1 do
    sinFarm#get_result i;
  done;
  accelerator#wait())
```

A small example

Compute function in Sarek

```
klet compute_sin = fun x ->
 let open Std in
 let open Math.Float32 in
 let mutable a = float x in
 for i = 0 to x  do
  a := sin a
 done;
 $"printf (\"worker : %d -> res %g\\n\",
                x, a)"$
```

Composition in OCaml

```
let _ =
  let sinFarm =
    FastFlow.to_farm compute_sin 10 in
  sinFarm#run_accelerator ();
  for i = 0 to size - 1 do
    sinFarm#offload_task i;
  done;
  sinFarm#no_more_tasks;
  for i = 0 to size - 1 do
    sinFarm#get_result i;
  done;
  accelerator#wait())
```

C translation

```
typedef struct __task{
 int x;
 } TASK;

TASK * f(TASK * task) {
     int x = task->x;

     float a;
      a = (float) (x)  ;
      for (int i = 0; i <= x; i++){
        a = sin (a) ;} ;
      printf ("worker : %d -> res %g\n", x, a) ;
      ;
     return task;
   }
```

A small example

Compute function in Sarek

```
klet compute_sin = fun x ->
 let open Std in
 let open Math.Float32 in
 let mutable a = float x in
 for i = 0 to x  do
  a := sin a
 done;
 $"printf (\"worker : %d -> res %g\\n\",
                x, a)"$
```

Composition in OCaml

```
let _ =
  let sinFarm =
    FastFlow.to_farm compute_sin 10 in
  sinFarm#run_accelerator ();
  for i = 0 to size - 1 do
    sinFarm#offload_task i;
  done;
  sinFarm#no_more_tasks;
  for i = 0 to size - 1 do
    sinFarm#get_result i;
  done;
  accelerator#wait())
```

GPGPU kernel in Sarek

```
let gpu_sin = kern x n ->
let open Std in
let i = thread_idx_x + block_dim_x * block_idx_x in
if i < n then
  begin
    let mutable a = (Std.float i) in
    for c = 0 to i do
      a := Math.Float32.sin a;
    done;
    x.[<i>] <- a;
  end
```

Composition in OCaml

```
let v = Vector.create Vector.float32 size in
let thrdPerBlck = Devices.(match d.specific_info with
  | OpenCLInfo {device_type = CL_DEVICE_TYPE_CPU; _} -> 1
  | _ -> 256) in
let blckPerGrd = (size + thrdPerBlck - 1) / thrdPerBlck
in
let block = {Spoc.Kernel.blockX = thrdPerBlck;
             Spoc.Kernel.blockY =1; Spoc.Kernel.blockZ =1}
and grid= {Spoc.Kernel.gridX = blckPerGrd;
           Spoc.Kernel.gridY =1; Spoc.Kernel.gridZ =1} in
ignore(Kirc.gen ~only:Devices.OpenCL
         gpu_sin);
Kirc.run gpu_sin (v,size) (block, grid) d;
Devices.flush d ();
for i = 0 to size - 1 do
  Printf.printf "worker : %d -> res %g\n " i
    (Mem.get v i)
done;
```

A small example

Compute function in Sarek

```
klet compute_sin = fun x ->
 let open Std in
 let open Math.Float32 in
 let mutable a = float x in
 for i = 0 to x  do
  a := sin a
 done;
 $"printf (\"worker : %d -> res %g\\n\",
                x, a)"$
```

GPGPU kernel in Sarek

```
let gpu_sin = kern x n ->
let open Std in
let i = thread_idx_x + block_dim_x * block_idx_x in
if i < n then
  begin
    let mutable a = (Std.float i) in
    for c = 0 to i do
      a := Math.Float32.sin a;
    done;
    x.[<i>] <- a;
  end
```

Performance

Intel(R) Core(TM) i7-4710MQ CPU @ 2.50GHz

OCaml (sequential)	OCaml + Spoc (CPU as GPU)	OCaml + FastFlow
0.80s ($\times 1$)	0.15s ($\times 5.33$)	0.07s ($\times 11.42$)

Conclusion

Using GPGPU programming on multicore CPU

- Is efficient for data-parallel computation
- Has many limitations

OCaml + Fastflow

- (Very) new experiment
- Easier to use/debug than GPGPU
- Promises better performance with ”irregular” programs
- Independant from OpenCL

SPOC and Sarek

- Can be used as an interface for C programming
- Sarek can easily target other (imperative) languages

Future work

OCaml + Fastflow

- Complete implementation
- New ”cpu-oriented” features for Sarek (dynamic allocation, pointer, recursion, exceptions, ocaml values...)
- Try/Compare another approach (load multiple OCaml runtimes/virtual machines in Fastflow tasks)
- Use OCaml to compose new skeletons
- ~~Maybe~~ get Sarek out of SPOC...

Thanks

SPOC : `http://www.algo-prog.info/spoc/`
Spoc is compatible with x86_64 Unix (Linux, Mac OS X), Windows

For OCaml-Fastflow : look at SPOC `proto_fastflow` branch and
`https://github.com/mathiasbourgoin/OCaml_FastFlow`
or `docker pull mathiasbourgoin/ocaml_fastflow`
`docker run -it mathiasbourgoin/ocaml_fastflow`

for more information :
mathias.bourgoin@univ-orleans.fr

Marco Danelutto - Università di Pisa
Emmanuel Chailloux - Université Pierre et Marie Curie