Efficient Abstractions for GPGPU Programming

Mathias Bourgoin

Emmanuel Chailloux and Jean-Luc Lamotte

19.09.2014

Efficient abstractions for GPGPU programming

- GPGPU programming → general purpose computations on the GPU
- Abstractions → languages and algorithmic constructs
- Efficient → High Performance Computing
- Applications → computational science and numerical simulation

OpenGPU project

- Systematic Cluster
- Academic and Industrial partners
- Goal : provide open-source solutions for GPGPU programming
- Success : develop real size numerical applications

Graphic card

Properties of a dedicated graphic card

- Several multi-processors
- Dedicated memory
- Connected to a host (CPU) *via* a PCI-Express bus
- Implies data transfers between host and graphic card memories
- Complex and specific programming

Current hardware

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

GPGPU Programming

Two main frameworks

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

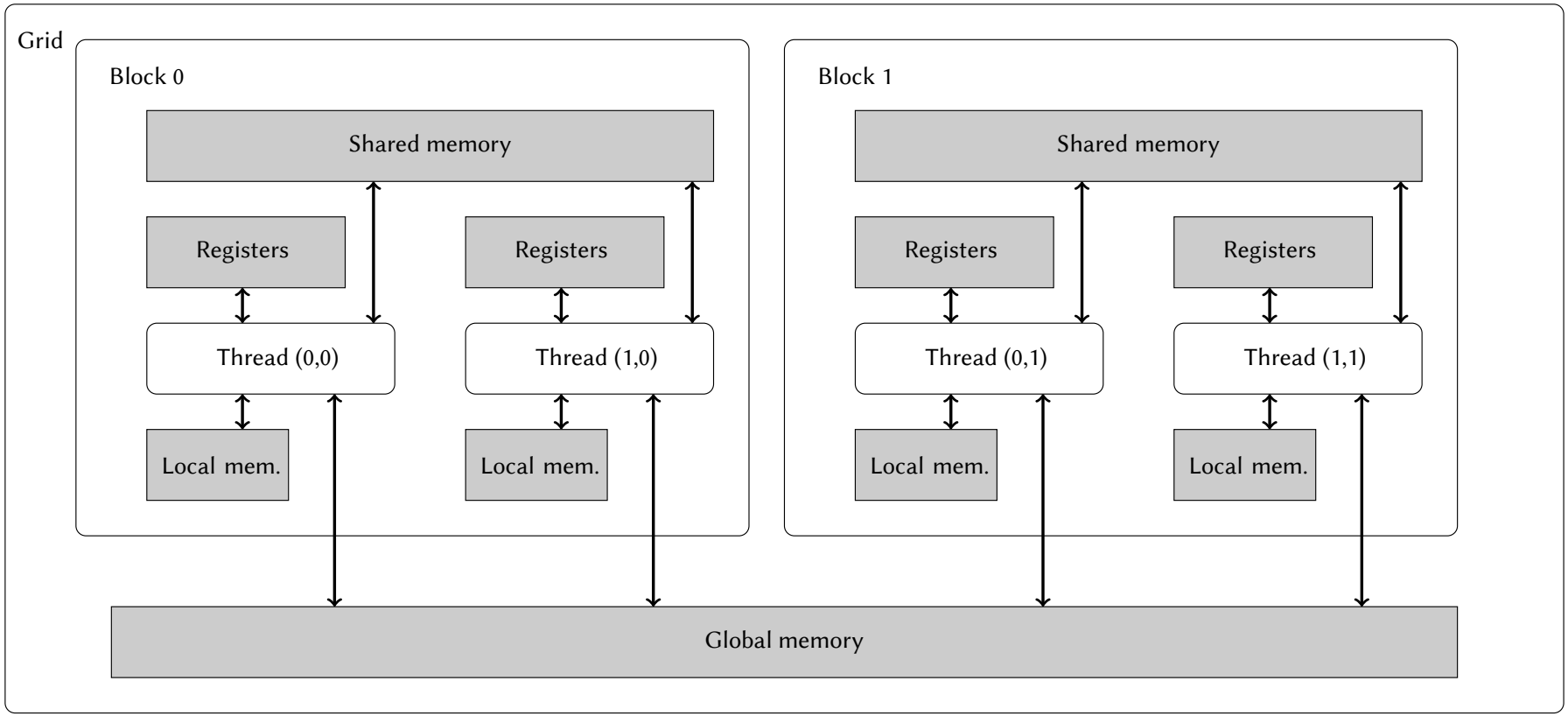
Different languages

- To write kernels
  - **Assembly** (ptx, 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



Do not forget tranfers between the host and its guests

	CPU-X86	GPU Mobile	GPU Gamer		GPU HPC
	i7-3770K	GTX 680M	GTX 680	7970HD	K20X
Memory bandwidth	25.6GB/s	115.2 GB/s	192.2GB/s	264GB/s	250GB/s

PCI-Express 3.0 maximum bandwidth is 16GB/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);
```

Motivations

Problems

- complex tools
- incompatible frameworks
- verbose languages/libraries
- low-level frameworks
- explicit management of devices and memory
- dynamic compilation

- hard to design/develop
- hard to debug
- very hard to achieve high performance

Motivations

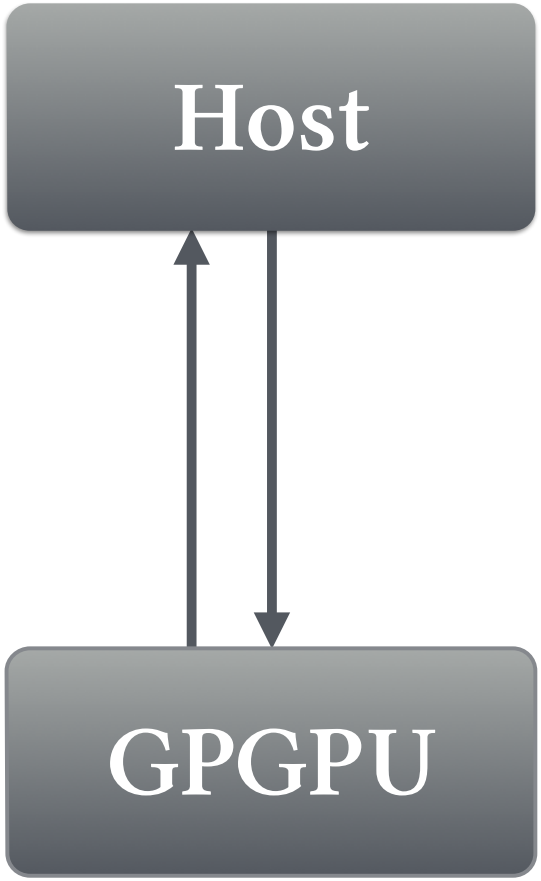
Solutions ?

- simple and expressive tools/languages
- compatible with every GPGPU framework
- benefit from high-level languages
- abstracts devices and memory transfers
- helps with composition
- compilation and static type checking
- easier to debug
- easier to design/develop

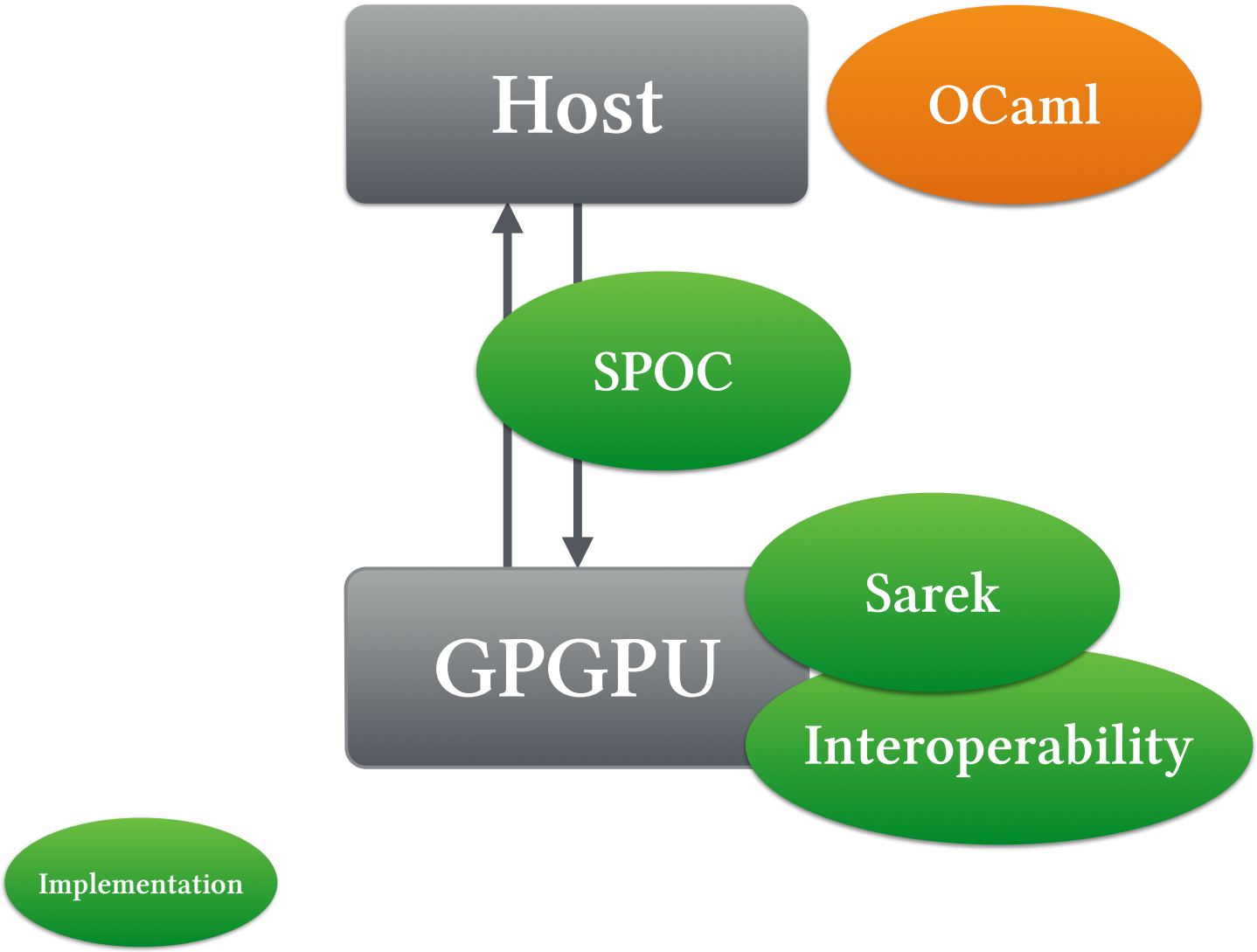
Some constraints ?

- keep a high level of performance
- compatible with highly heterogeneous systems

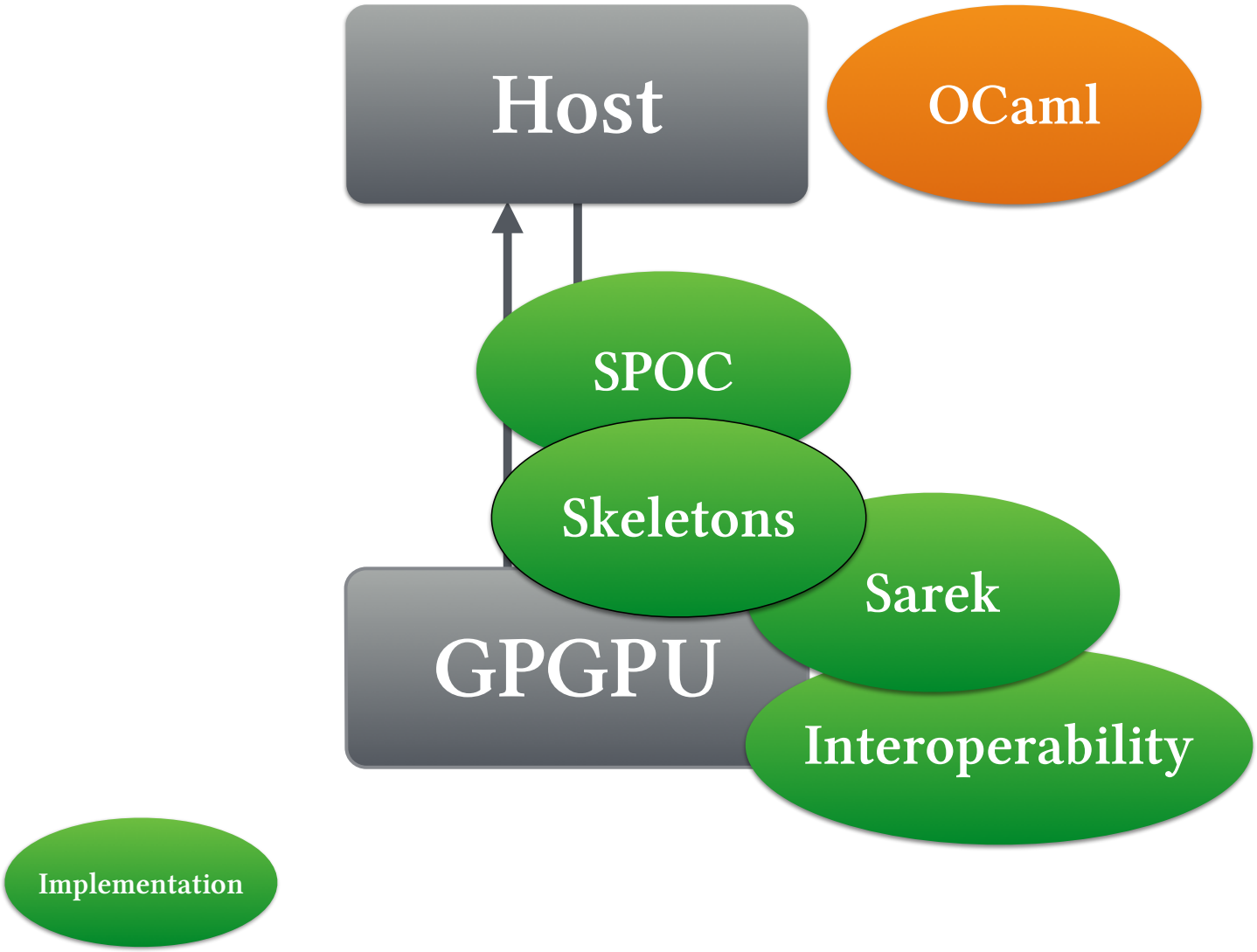
Overview



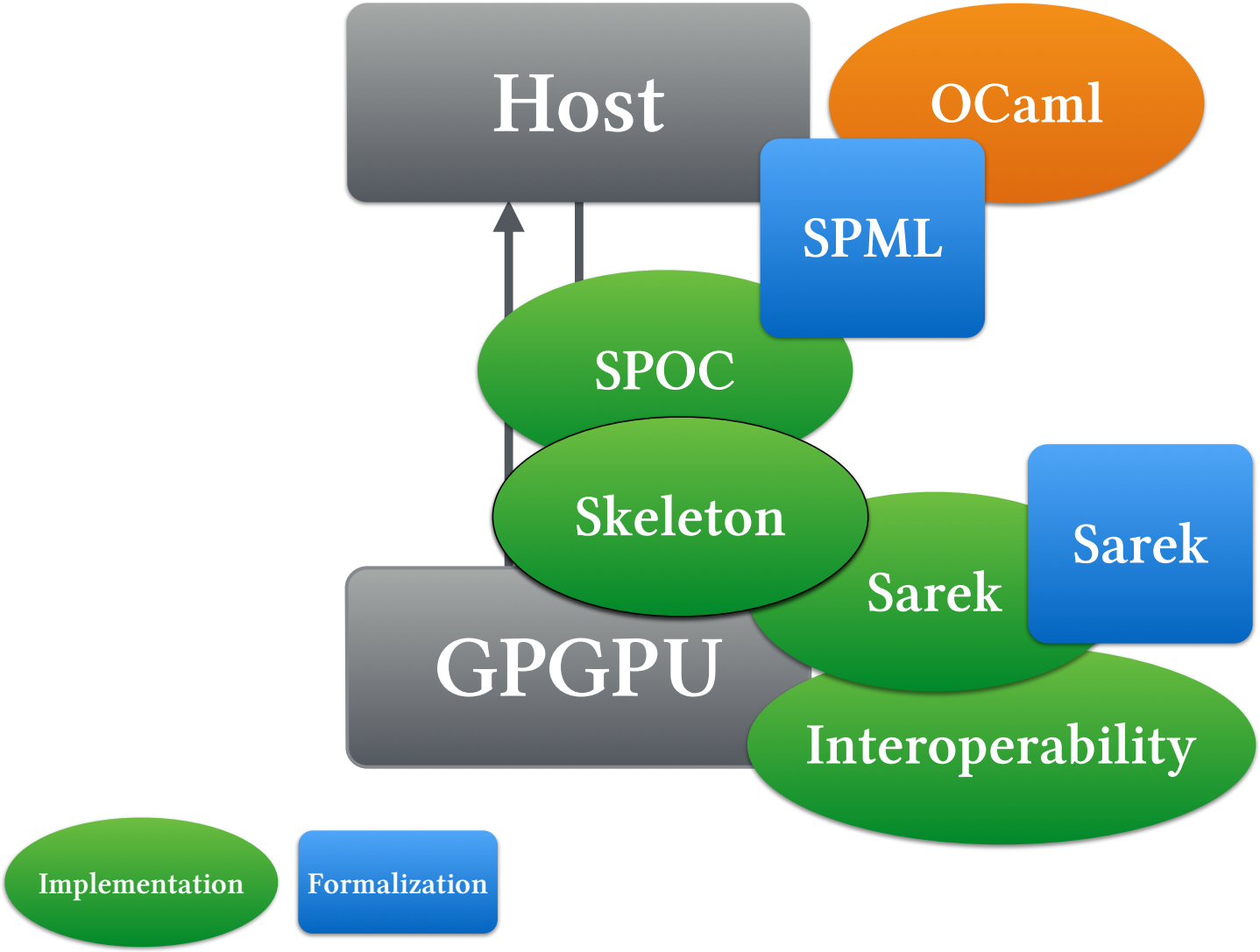
Overview


Host
OCaml
SPOC
GPGPU
Sarek
Interoperability
Implementation

Overview


Host
OCaml
SPOC
Skeletons
Sarek
GPGPU
Interoperability
Implementation

Overview


Host
OCaml
SPML
SPOC
Skeleton
Sarek
Sarek
GPGPU
Interoperability
Implementation
Formalization

Two languages for GPGPU programming

Host side : SPML (Stream Processing miniML)

- Based on ML (simple and expressive)
- Functional and imperative
- Statically typed
- With primitives dedicated to GPGPU programming
- With automatic memory transfers between devices

Kernel side : Sarek (Stream ARchitectures using Extensible Kernels)

- Based on the C extensions from Cuda/OpenCL
- Imperative
- Statically typed
- With an ML-like syntax to be consistent with SPML

Operational semantics of SPML and Sarek

Why ?

- Describe precisely the properties of the implemented languages
- Provide guarantees on data location
- Provide guarantees on the overall behavior of a GPGPU program
- Provide a specification for future implementations in multiple languages

SPML

Transferable vector

- Uniform data set
- Automatically transferable from a memory space to another

SPML properties

- A vector cannot be located in multiple memory spaces at a time.
- To run a kernel, the vectors it depends must be located on GPGPU memory.
- The host cannot modify the data used by a running kernel.
- When a kernel finishes, its vectors are acessible by the host.

Sarek

Transferable vector

- Uniform data set
- Automatically transferable from a memory space to another

Sarek properties

- Kernels cannot access host memory.
- Kernels cannot modify vectors adresses.

Main Goals

- Target Cuda/OpenCL frameworks with OCaml
- Unify these two frmeworks
- Abstract memory transfers (SPML)
- Use static type checking to verify kernels (Sarek)
- Propose abstractions for GPGPU programming
- Keep the high performance

Host-side solution : an OCaml library

SPOC overview

Abstract frameworks

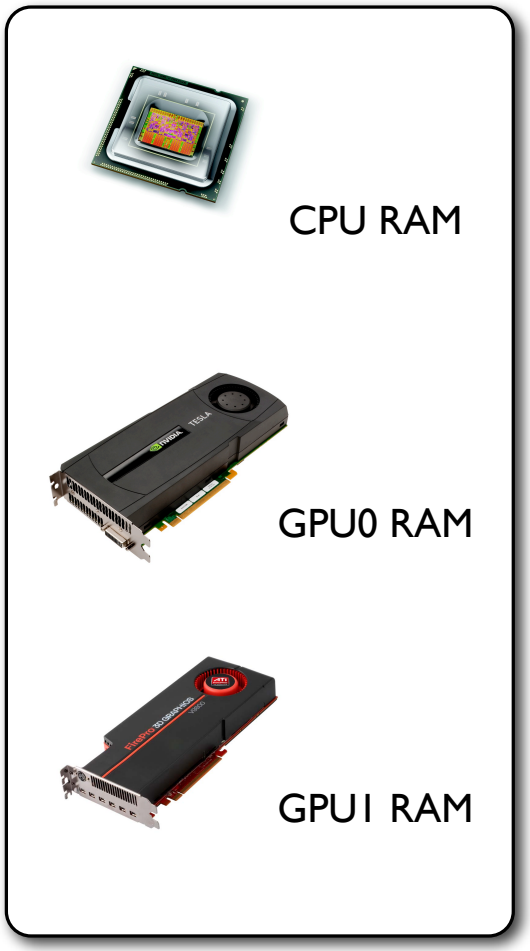
- Unify both APIs (Cuda/OpenCL), **dynamic linking**.
- Portable solution, multi-GPGPU, heterogeneous

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

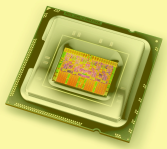
A small example



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 = 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 small example



v1
v2
v3
CPU RAM

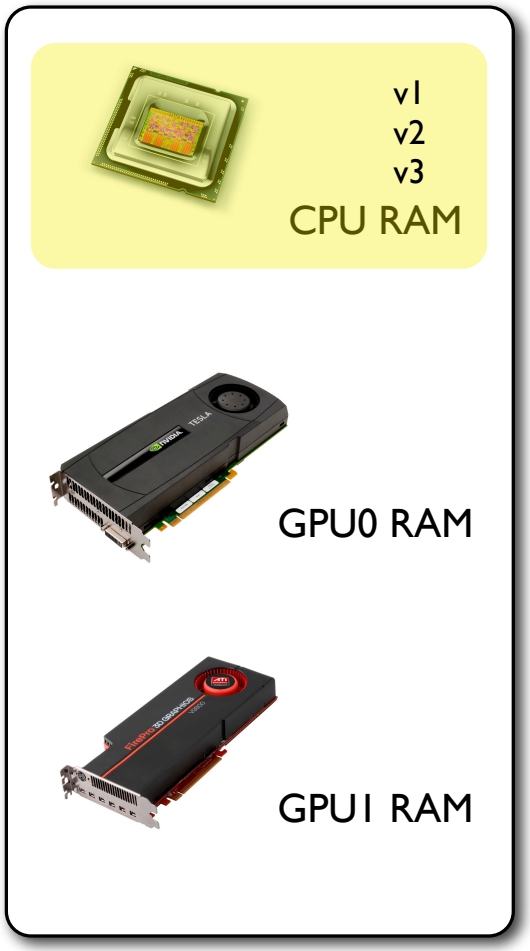
GPU0 RAM

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;
```

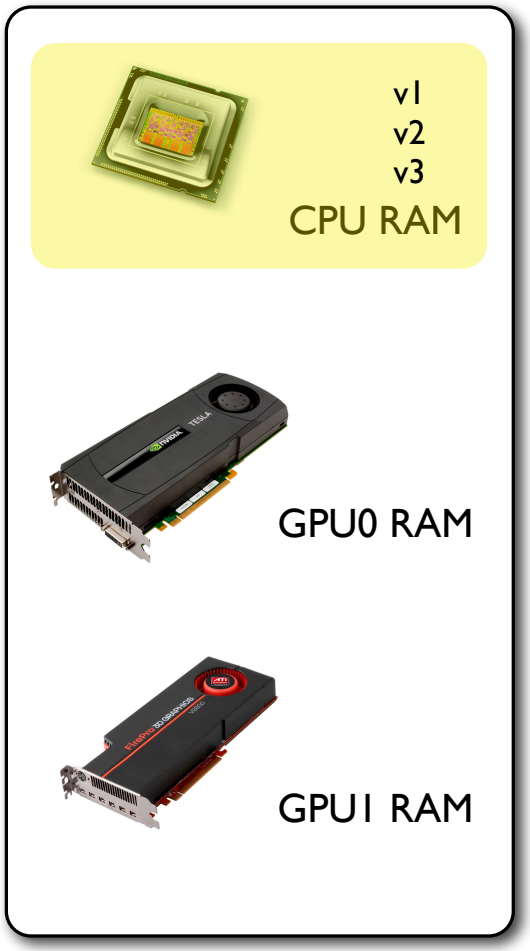
A small example



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 = 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;
```

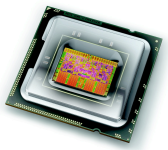
A small example



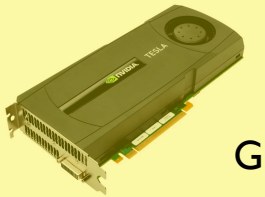
Example

```
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 = 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 small example



CPU RAM



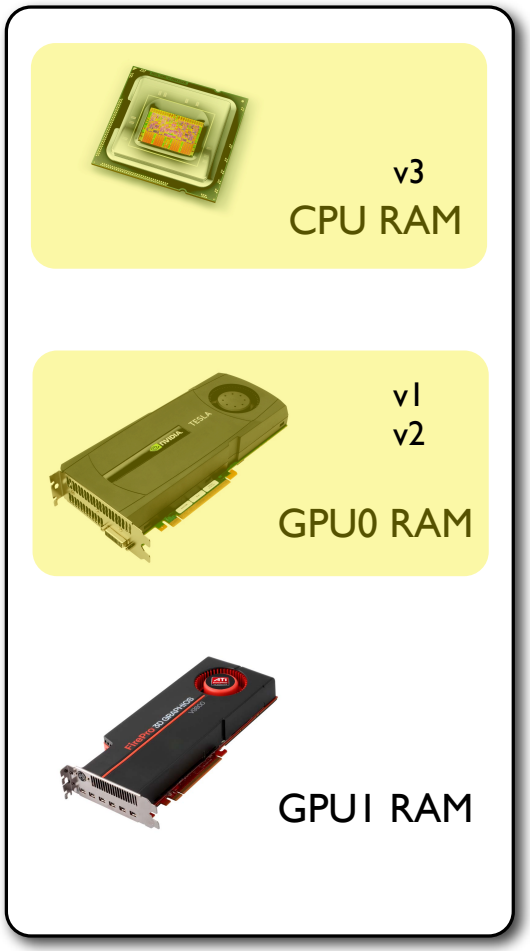
v1
v2
v3
GPU0 RAM

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 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 = 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;
```

GPGPU kernels

What we want

- Simple to express
- Predictable performance
- Easily extensible
- Current high performance libraries
- Optimizable
- Safer

Two Solutions

Interoperability with Cuda/OpenCL kernels

- Higher optimizations
- Compatible with current libraries
- Less safe

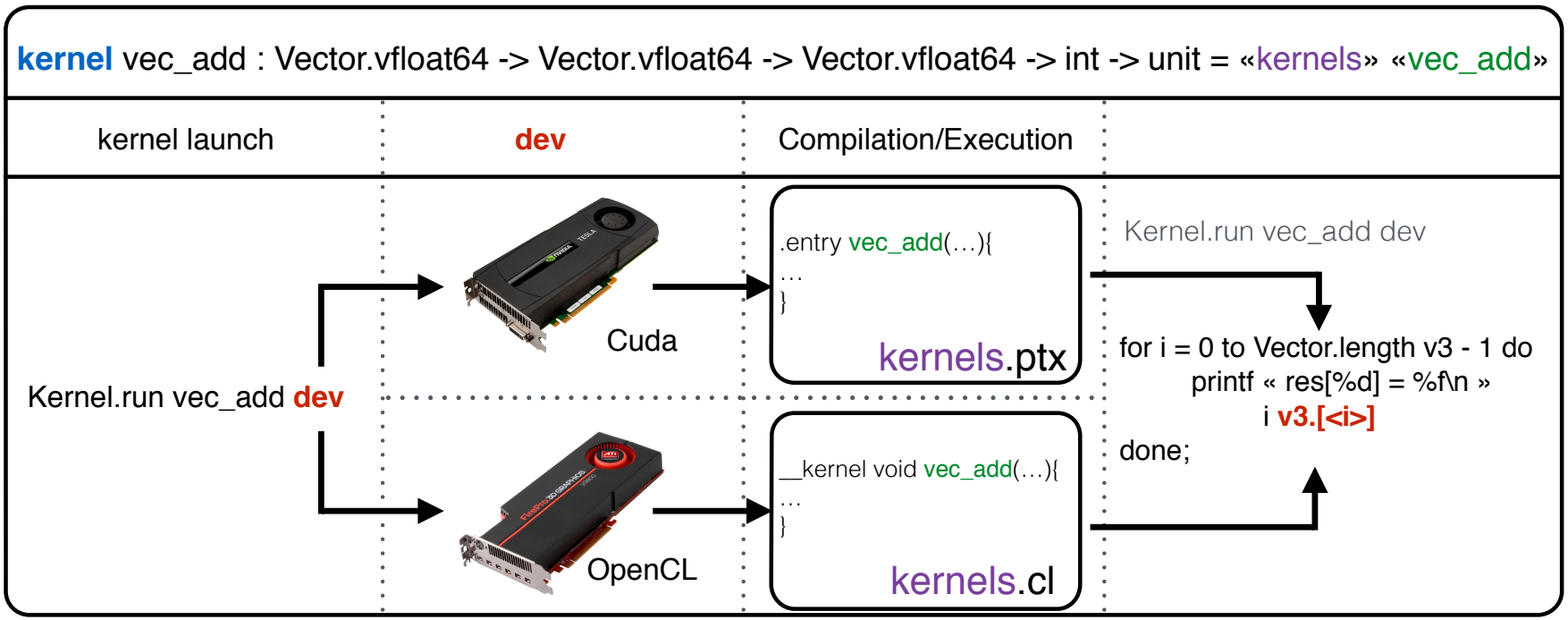
A DSL for OCaml : Sarek

- Easy to express
- Easy transformation from OCaml
- Safer

External kernels

Type safety

- Static type checking of kernel parameters (at compile-time).
- `Kernel.run` compiles kernels from .ptx / .cl sources.



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];
}
```

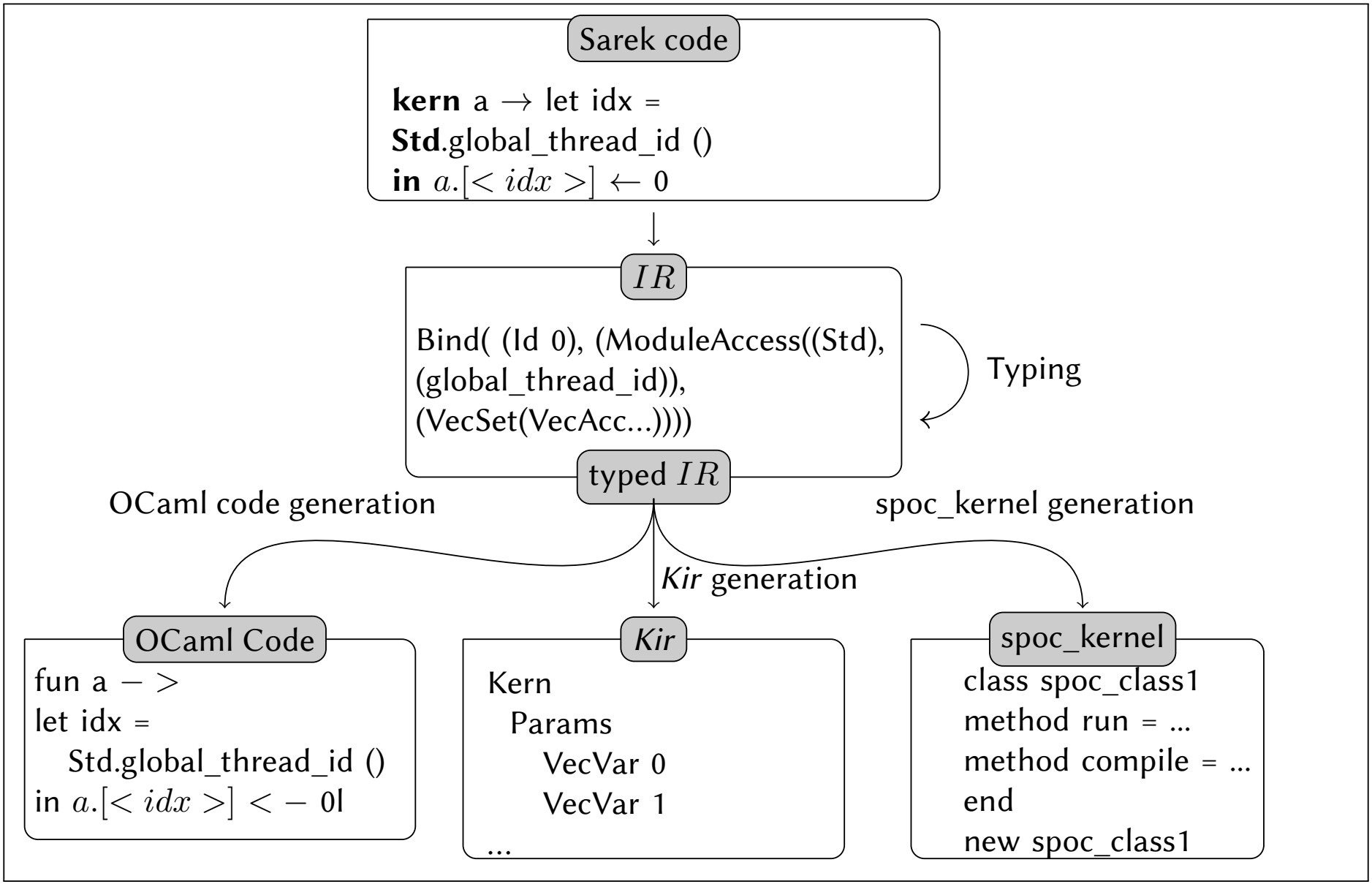
Vector addition with Sarek

```
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>]
```

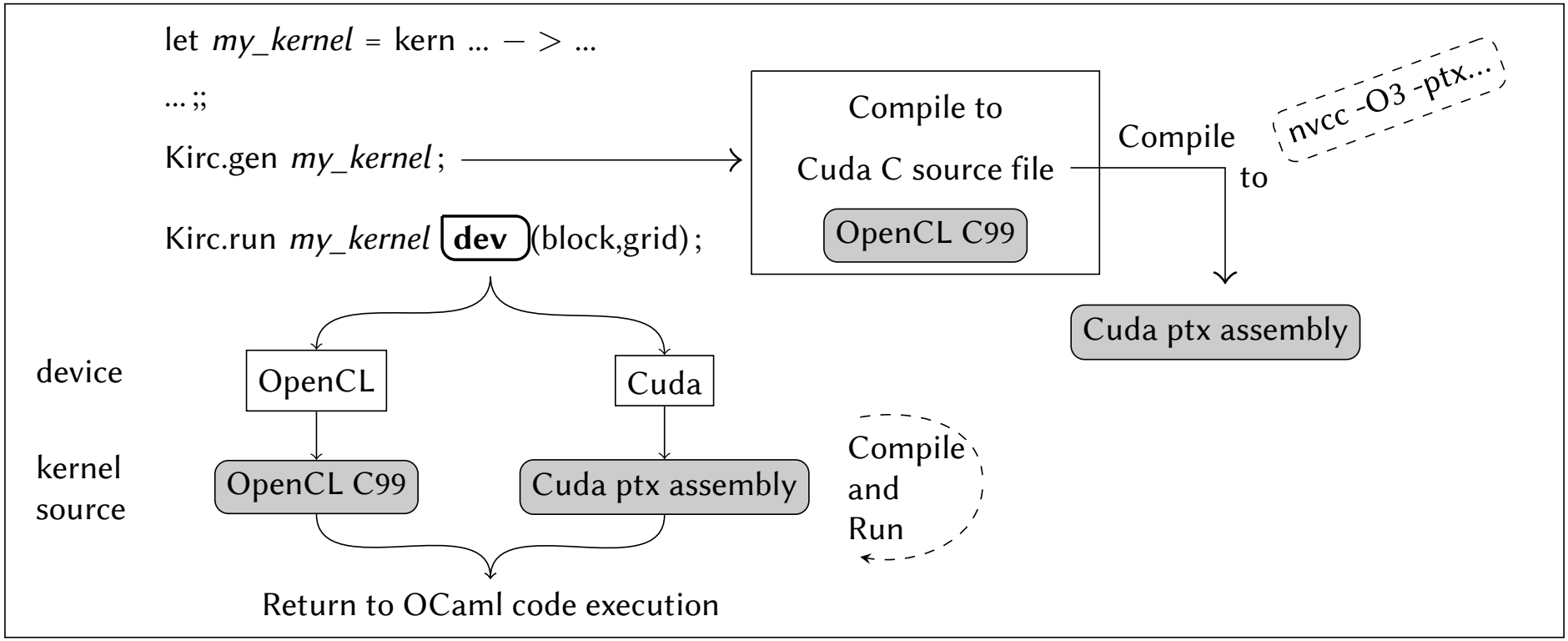
Sarek features

- ML-like syntax
- type inference
- static type checking
- **static** compilation to OCaml code
- **dynamic** compiation 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

Kernel composition

Composition

Compose multiple kernels to express complex algorithms

Why ?

- Eases programming
- Makes new automatic optimizations possible
  - optimize the virtual grid depending on vectors lengths
  - automatic overlapping of transfers by computation
  - make transfers as soon as possible

Problem

Kernels are procedures
Difficult to identify input/output automatically (with external native kernels)

Parallel skeletons

A skeleton is composed of

- a kernel
- an execution environment
- an input
- an output

Two running functions :

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

- Explicitly describes relations between kernels/data
- Provides automatic optimizations

Example

Power iteration

SPOC

```
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
```

Squeletons

```
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

```
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)))))
           (reduce max) u;
  x0<-x;iter<-iter+1;
done
```

Example

Power iteration

SPOC

```
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
```

Squeletons

```
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

```
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)))))
           (reduce max) u;
  x0<-x;iter<-iter+1;
done
```

Example

Power iteration

SPOC

```
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
```

Squeletons

```
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

```
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)))))
           (reduce max) u;
  x0<-x;iter<-iter+1;
done
```

Benefits

Skeleton

```
(* 'a : environment, 'b : input, 'c : output *)
val MAP : 'a external_kernel -> 'b vector -> 'c vector -> ('a,'b,'c) skeleton
val run : ('a,'b,'c) skeleton -> 'a -> 'c vector
```

- Automatic grid/block mapping on GPU
- Automatic parallelization on multiple GPUs

Composition

```
val PIPE : ('a,'b,'c) skeleton -> ('d,'c,'e) skeleton -> ('f,'b,'e) skeleton
```

- Automatic overlapping of transfers by computations

Sarek transformations

Using Sarek

Transformations are OCaml functions modifying Sarek AST :
Example :

```
map (kern a -> b)
```

Scalar computations ($'a \rightarrow 'b$) are transformed into vector ones ($'a\ vector \rightarrow 'b\ vector$).

Vector addition

```
let v1 = Vector.create Vector.float64 10_000
and v2 = Vector.create Vector.float64 10_000 in
let v3 = map2 (kern a b -> a + b) v1 v2

val map2 :
  ('a -> 'b -> 'c) sarek_kernel ->
  ?dev:Spoc.Devices.device ->
  'a Spoc.Vector.vector ->
  'b Spoc.Vector.vector -> 'c Spoc.Vector.vector
```

Sarek transformations

```
sort (kern a b -> a - b) vec1
val sort : ('a -> 'a -> int) sarek_kernel -> 'a vector -> unit
```

Injection into sort kernel

```
let bitonic_sort = kern v j k ->
 let open Std in
 let i = thread_idx_x +
         block_dim_x*block_idx_x in
 let ixj = Math.xor i j in
 let mutable temp = 0. in
 if ixj >= i then (
    if (Math.logical_and i k) = 0  then (
    if v.[<i>] - v.[<ixj>] > 0 then
     (temp := v.[<ixj>];
      v.[<ixj>] <- v.[<i>];
      v.[<i>] <- temp))
     else if v.[<i>] - v.[<ixj>] <= 0 then
       (temp := v.[<ixj>];
        v.[<ixj>] <- v.[<i>];
        v.[<i>] <- temp);)
```

```
while !k <= size do
  j := !k lsr 1;
  while !j > 0 do
    run bitonic_sort
        (vec1,!j,!k)
        device;
    j := !j lsr 1;
  done;
  k := !k lsl 1 ;
done;
```

Host composition

Benchmarks

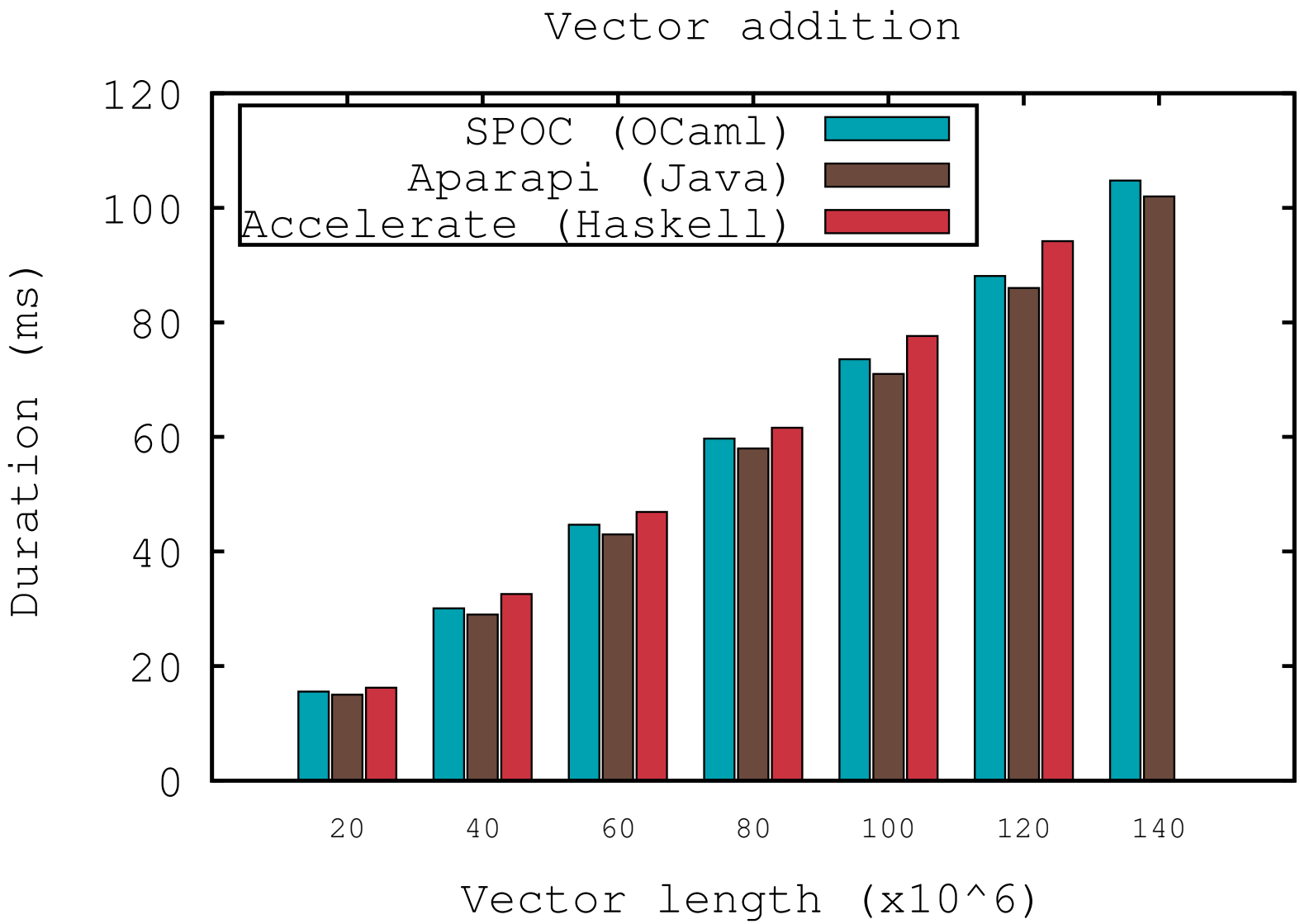
Complex test environments

Different levels of performance when using Cuda or OpenCL on the same device
Program behavior is influenced by drivers
Beware the operating system
Brittle hardware...

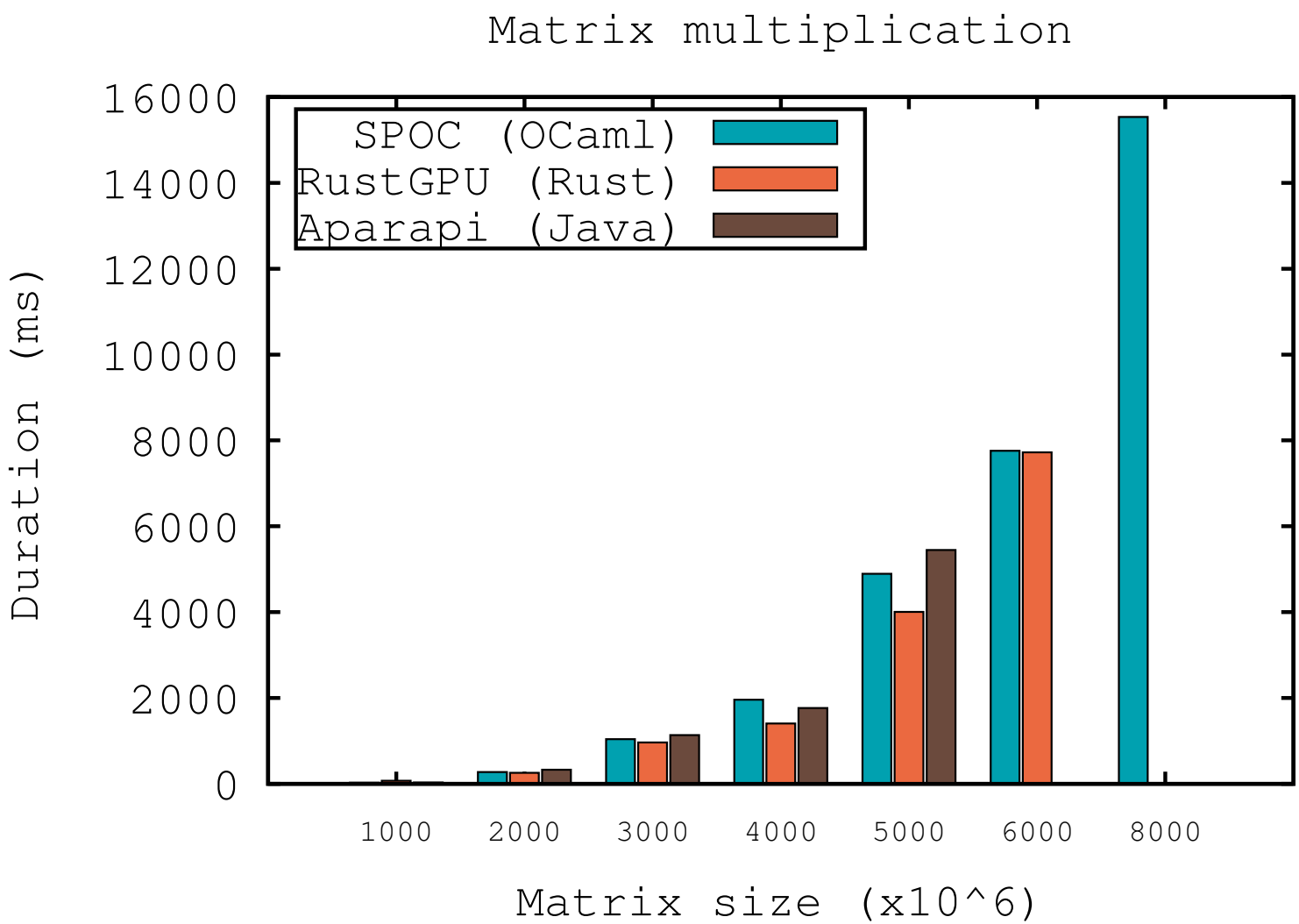
Comparison

Accelerate (Haskell) :	vectors & combinators Cuda / OpenCL / Sequential no external/internal kernels
Aparapi (Java)	vectors OpenCL kernels in Java
Rust-GPU (Rust - Mozilla)	OpenCL for transfers Cuda for the kernels

Benchmarks : small examples : comparisons



Benchmarks : small examples : comparisons



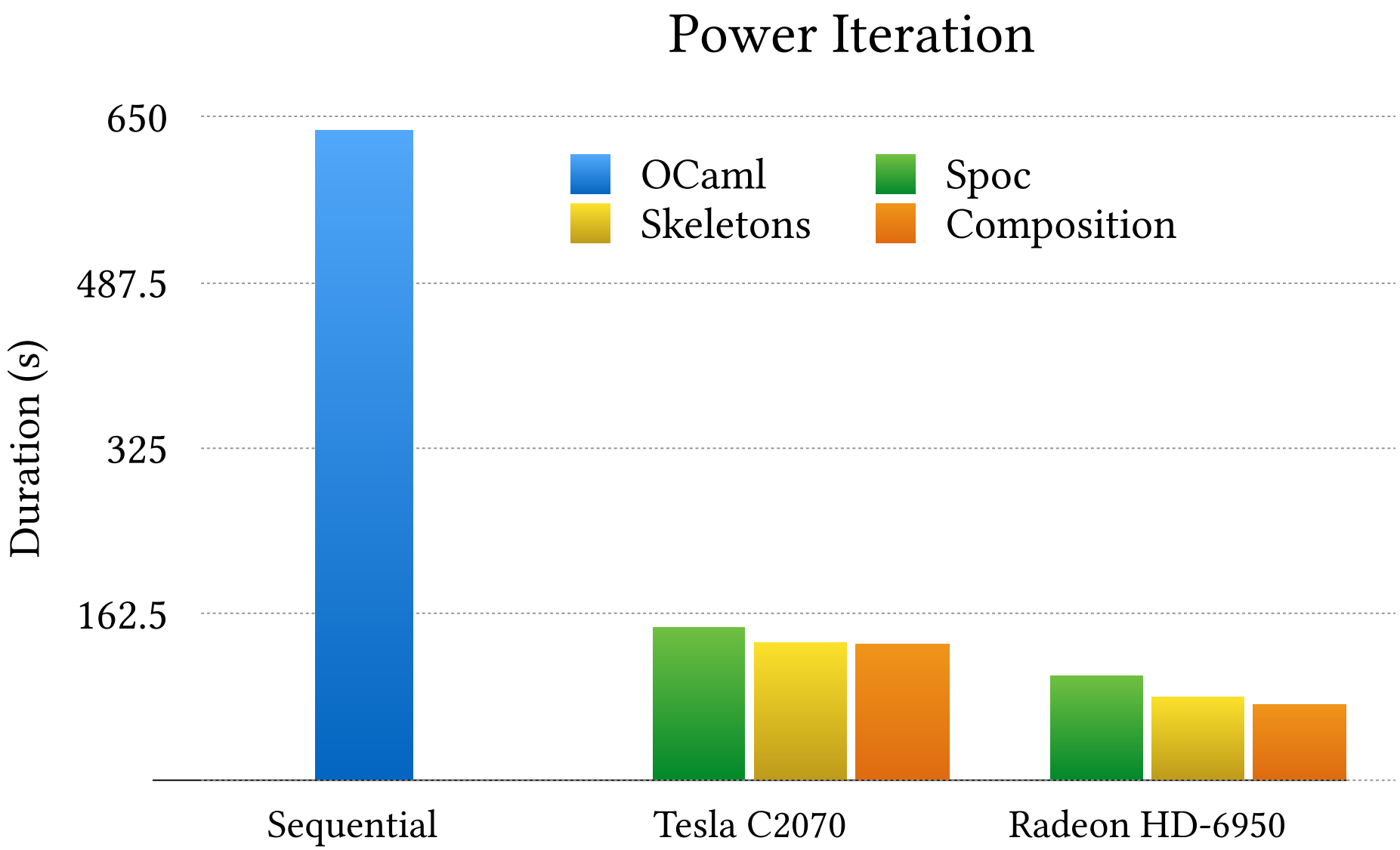
GPU : GTX-680

Benchmarks : small examples : multi-GPGPU

System \ Test	MatMult	
	duration (s)	speedup
OCaml sequential	85.0	×1.0
1 GPU Tesla C2070	1.2	×65.4
Multi-GPU system 1× GTX 680 + 1× GTX 460 + 2× Quadro Q2000	0.4	×193.2

Block muliplication
Dynamic balancing
Simple algorithm (~20 lines of OCaml code)

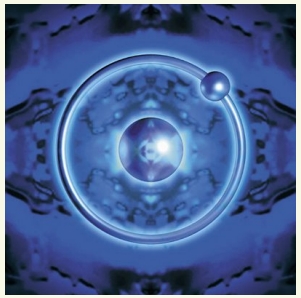
Benchmarks : skeletons and composition

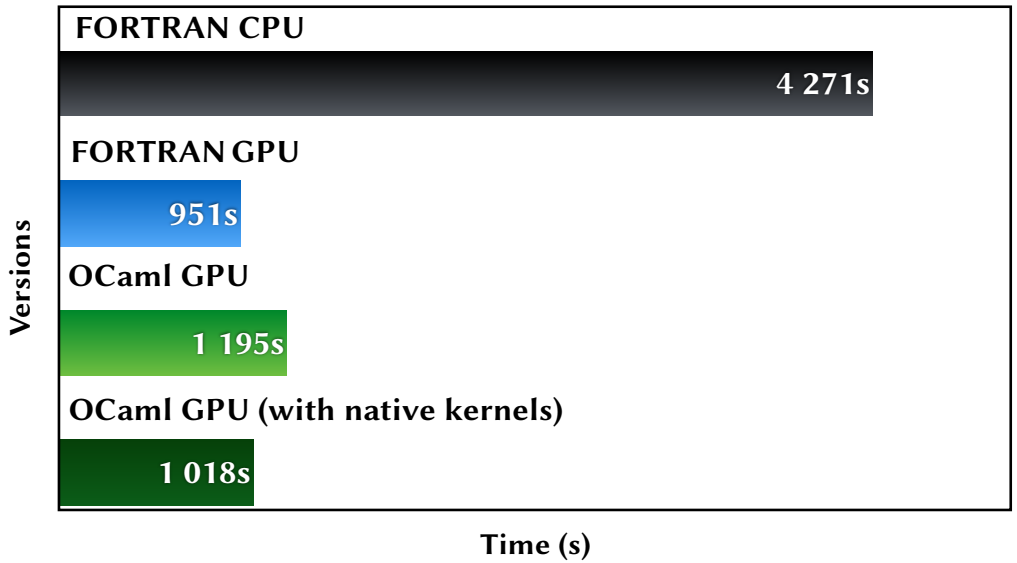


Real size example

PROP

- Awarded by the *UK Research Councils' HEC Strategy Committee*
- Simulates the scattering of e^- in H-like ions at intermediates energies
- Programmed in FORTRAN
- Compatible with : sequential architectures, HPC *clusters*, super-computeurs





SPOC+Sarek achieves 80% of hand-tuned Fortran performance.
SPOC+external kernels is on par with Fortran (93%)

Type-safe
Memory manager + GC

30% code reduction
No more transfers

Conclusion

Formalization : SPML and Sarek

- Operational semantics
- Provide guarantees on data location/usage

Implementation : SPOC

- Unifies Cuda/OpenCL
- Automatic transfers
- Compatible with existing optimized libraries

Implementation : Sarek

- OCaml-like syntax
- Type inference and static type checking
- Easily extensible

Conclusion

Implementation : Skeletons

- Simplifies programming
- Offers additional automatic optimizations

Benchmarks

- Same performance as with other solutions
- Heterogenous
- Efficient with GPGPUs as well as with multicore CPUs

Application : PROP

- More safety (memory/types)
- Keeps the level of performance
- Validates our solution

Last year work (ATER - LIP6)

SPOC for the web

- Access GPGPU from web browsers
- Using the `js_of_ocaml` compiler
- Translation of the lowl-level part of SPOC + development of a dedicated memory manager
- Source and web demos/tutorials : `http://www.algo-prog.info/spoc/`
- SPOC can be installed *via* OPAM (OCaml Package Manager)

Accessibility and teaching

- Simpler than classic tools : no more transfers
- Web = instantly accessible
- Perfect playground for GPGPU/HPC courses
  - focused on kernel optimization
  - but mostly on algorithms composition

Last year (and future) work

Extend implementation

- Extend Sarek : types, functions, *recursion, polymorphism...*
- Optimize code generation
- Dynamic and automatic optmizations for multiples architectures

Extend skeletons

- Cost model for Sarek
- More skeletons based on Sarek
- Skeletons dedicated to very heterogeneous architectures (super-computers)

Thanks

Emmanuel Chailloux
Jean-Luc Lamotte

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

for more information :
mathias.bourgoin@imag.fr