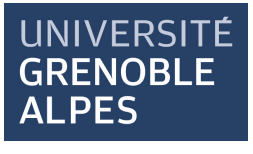
Efficient Abstractions for GPGPU Programming

Mathias Bourgoin

10.03.2015



Efficient abstractions for GPGPU programming

PhD (LIP6/UPMC)

- 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)

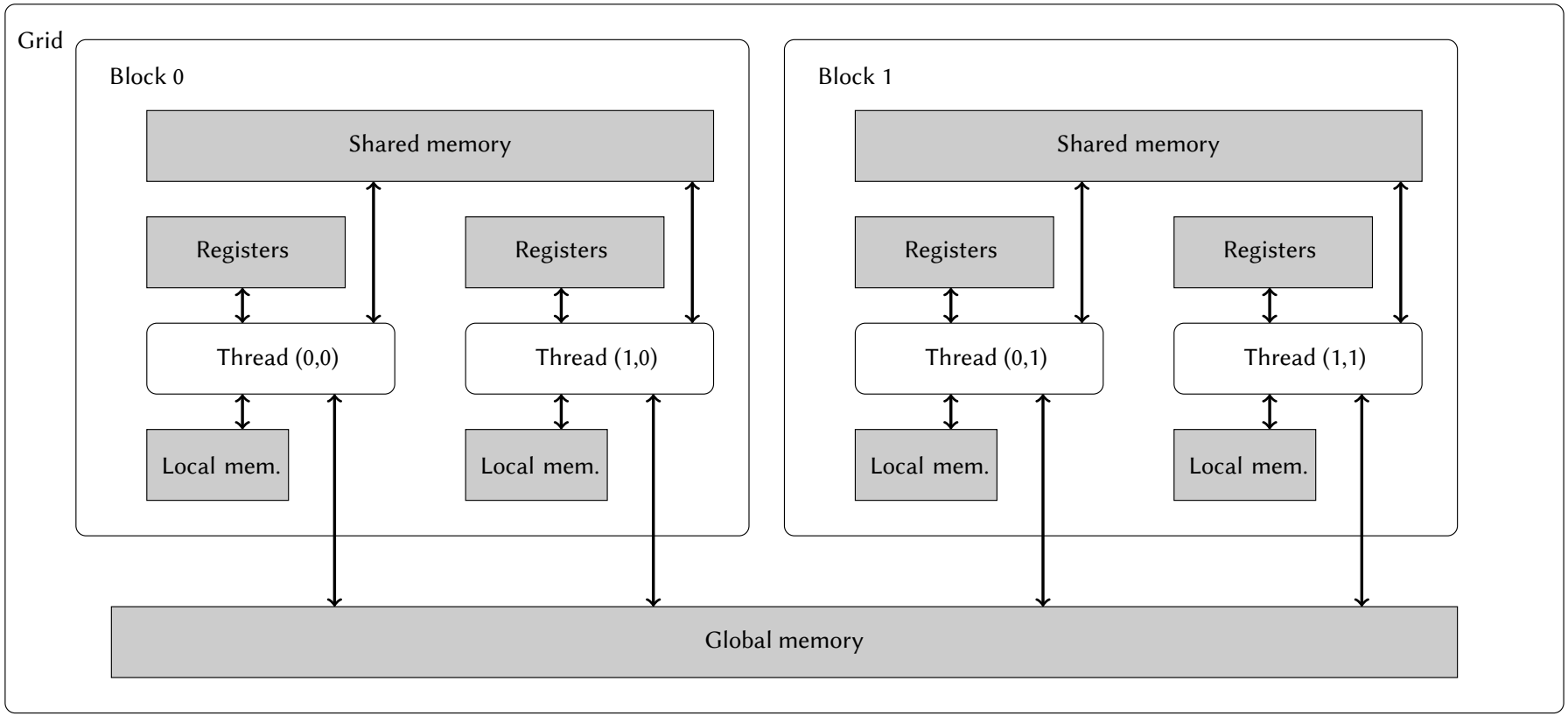
Different languages

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



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

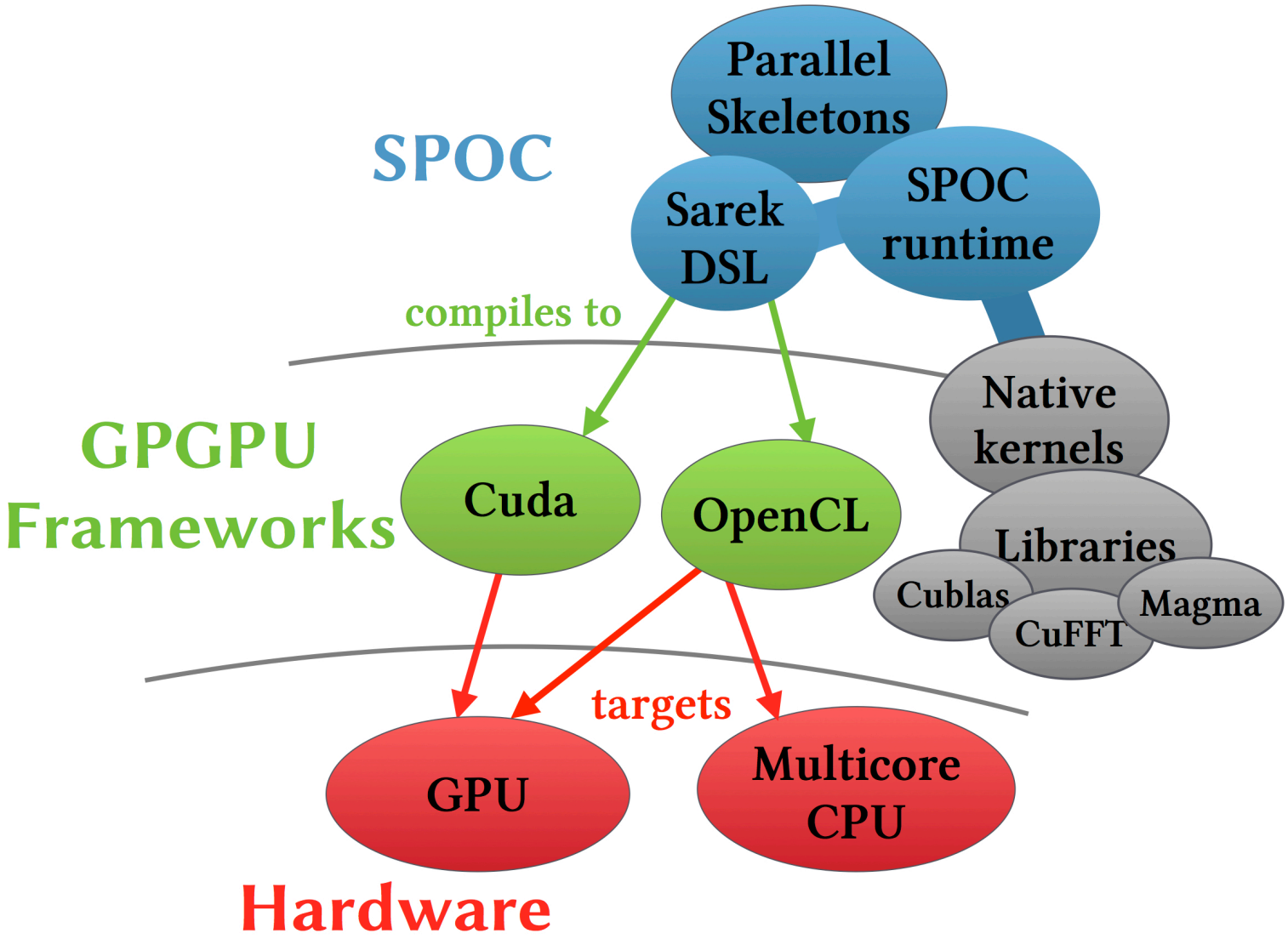
```
__kernel void vec_add(__global const double * a,
                      __global const double * b,
                      __global double * c, int N)
{
    int nIndex = get_global_id(0);
    if (nIndex >= N)
        return;
    c[nIndex] = a[nIndex] + b[nIndex];
}
```

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);
```

GPGPU Programming with OCaml



Main Goals

- Target Cuda/OpenCL frameworks with OCaml
- Unify these two frmeworks
- Abstract memory transfers
- Use static type checking to verify kernels
- 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

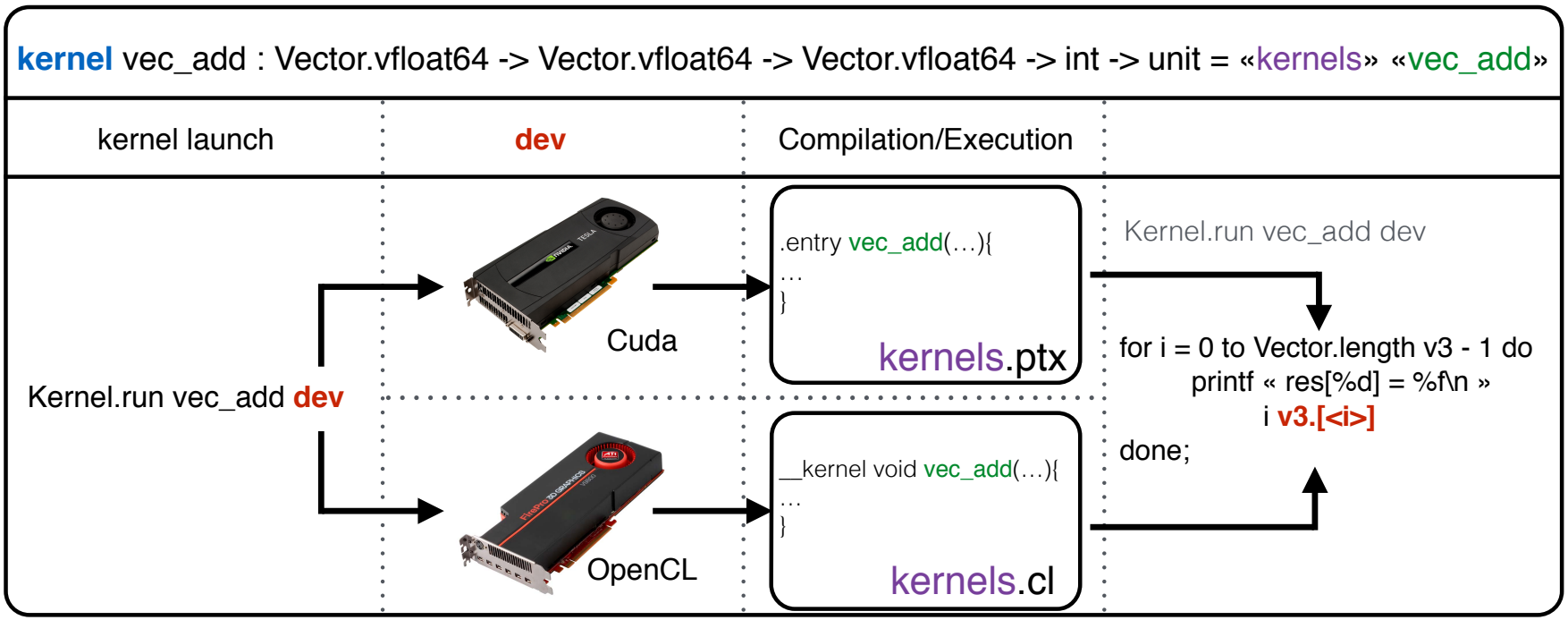
Vectors move automatically between CPU and GPGPUs

- 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

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];
}
```

Sarek

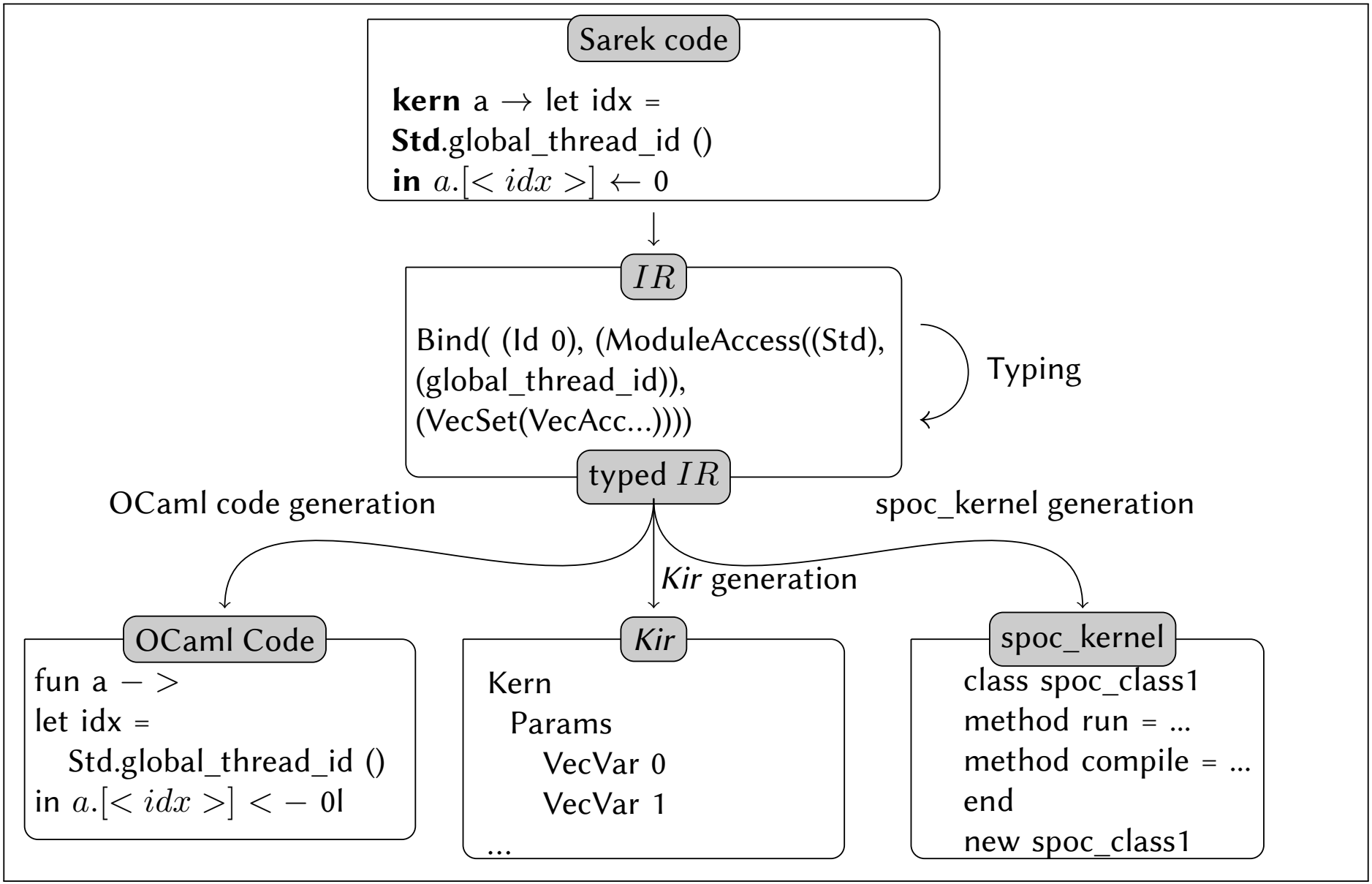
Vector addition with Sarek

```
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>]
```

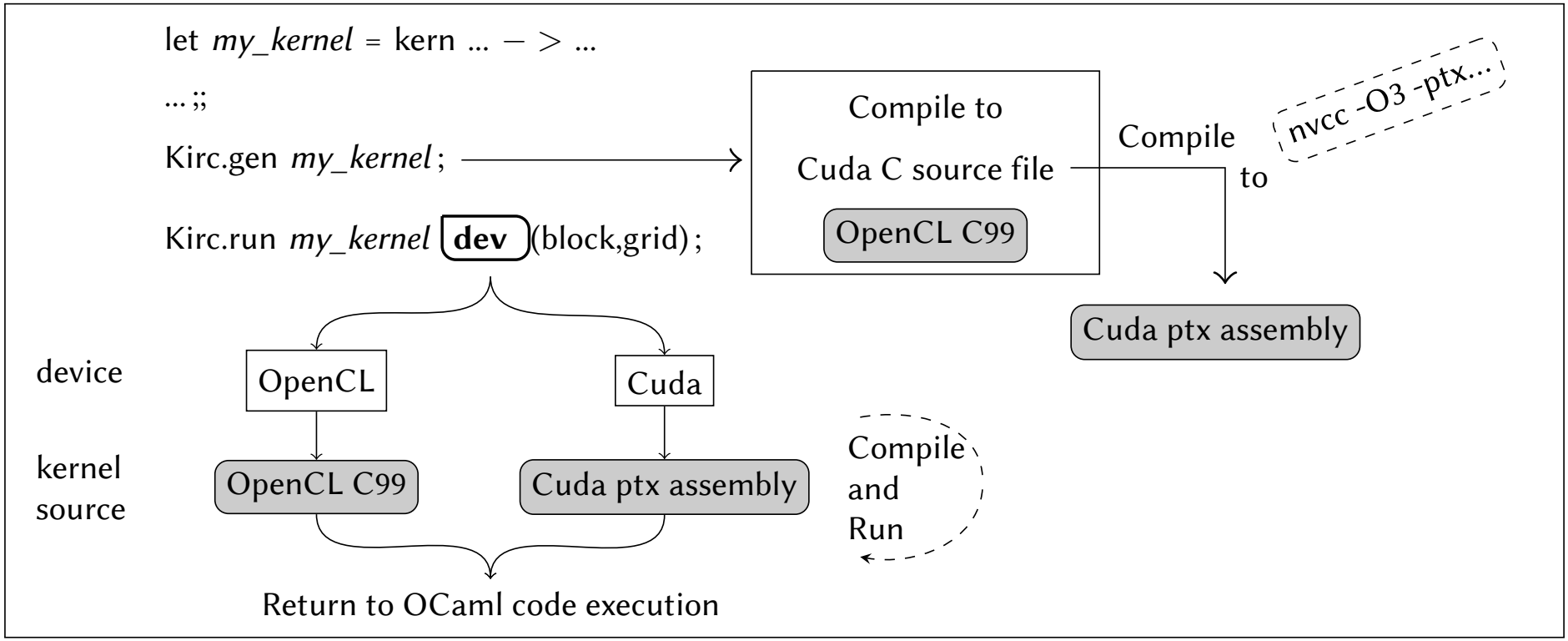
Sarek features

- 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

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
```

Skeletons and Composition

Skeleton

```
(* 'a : environment, 'b : input, 'c : output *)
val SKEL_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

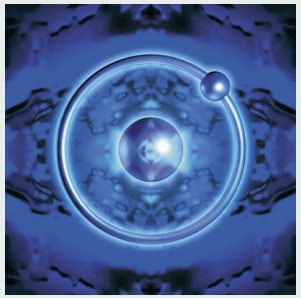
```
val SKEL_PIPE : ('a,'b,'c) skeleton -> ('d,'c,'e) skeleton ->
                ('f,'b,'e) skeleton
```

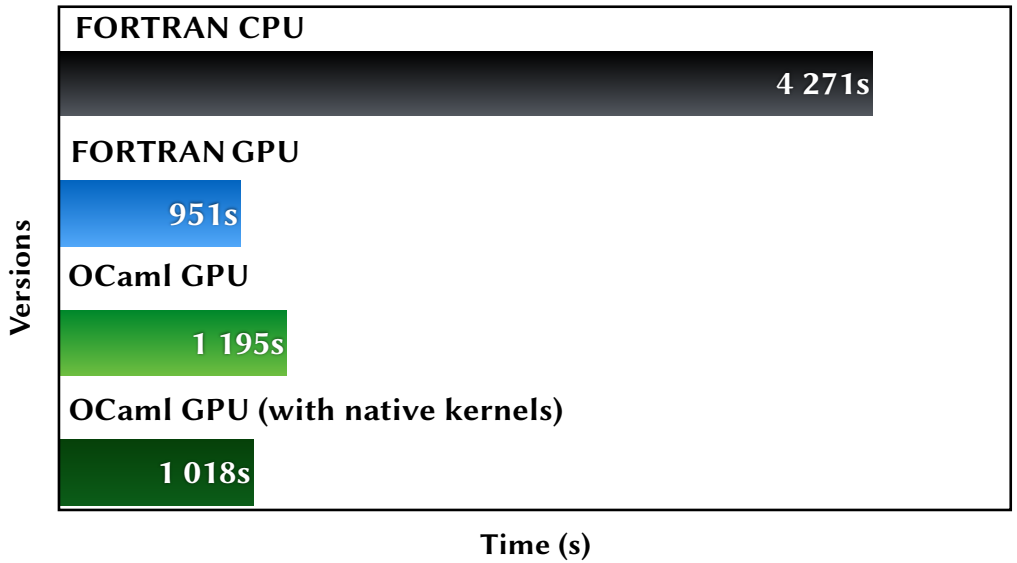
- Automatic overlapping of transfers by computations

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	30% code reduction
Memory manager + GC	No more transfers

Conclusion

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

Implementation : Skeletons

- Simplifies programming
- Offers additional automatic optimizations

Conclusion

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

Current (and future) work

Extend implementation

- Extend Sarek : types, functions, *recursion, polymorphism...*
- Optimize code generation
- Dynamic and automatic optmizations for multiples architectures
- Target new architectures (Kalray MPPA 256)

Extend skeletons

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

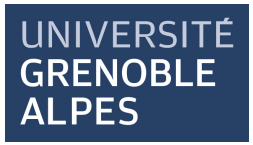
Current (and future) work

Extend implementation

- Extend Sarek : types, functions, *recursion, polymorphism...*

```
ktype color = Spades | Hearts | Diamonds | Clubs ;;
ktype colval = {c:color; v : int32} ;;
ktype card =  Ace of color | King of color | Queen of color
            | Jack of color | Other of colval;;

let compute = kern cards trump values n ->
  let value = fun a trump->
    match a with
    | Ace c -> 11
    | King c -> 4
    | Queen c -> 3
    | Jack c -> if c = trump then 20 else 2
    | Other cv ->
      if cv.v = 10 then 10 else if (cv.c = trump) && (cv.v = 9) then 14 else 0
  in
  let open Std in
  let i = thread_idx_x + block_dim_x * block_idx_x in
  if i < n then
    values.[<i>] <- value cards.[<i>] trump.[<0>]
```

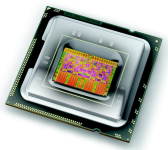
Thanks



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

A small example



CPU RAM

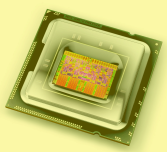
GPU0 RAM

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 = 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



v1
v2
v3
CPU RAM

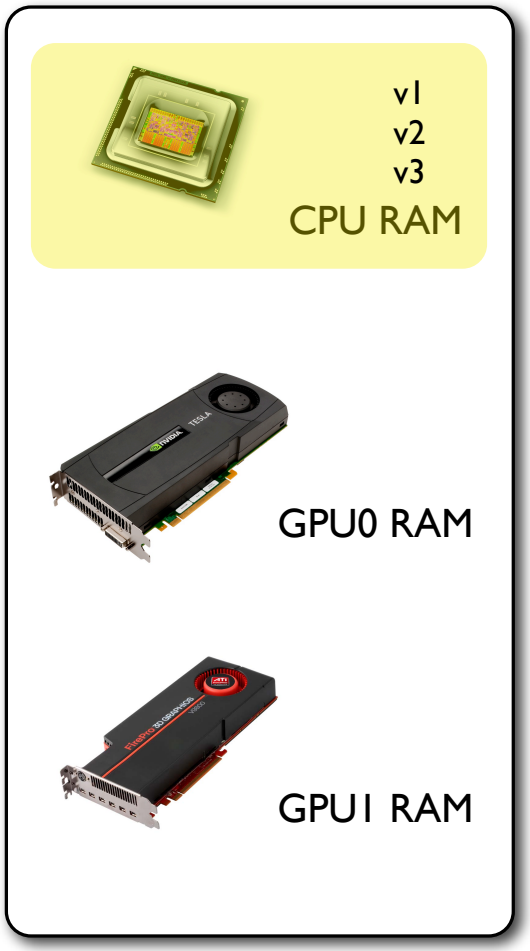
GPU0 RAM

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 = 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;
```

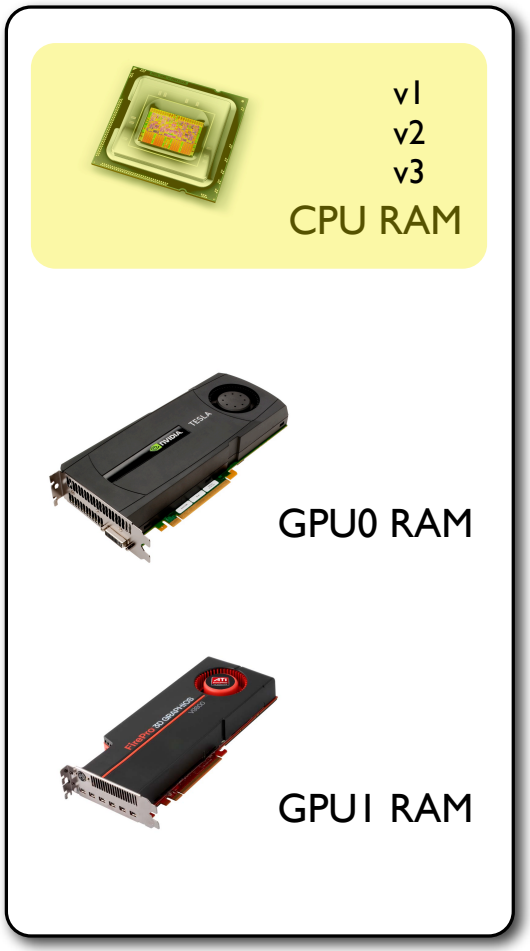
A small example



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;
```

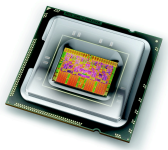
A small example



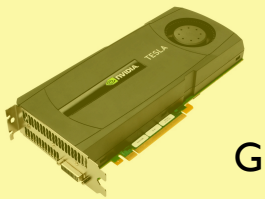
Example

```
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



CPU RAM



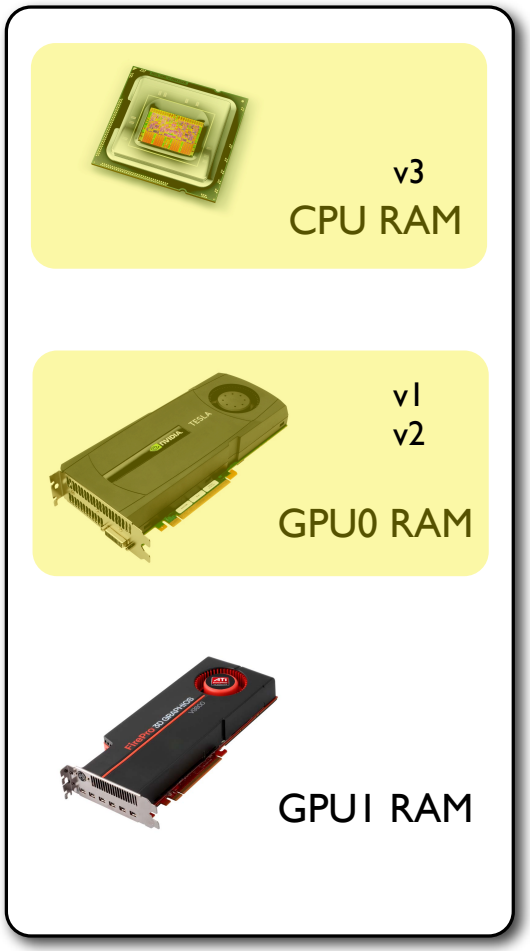
v1
v2
v3
GPU0 RAM

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 = 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 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