Profiling High-Level Heterogeneous Programs

with SPOC : the GPGPU framework for OCaml

GDR - GPL 2017

Mathias Bourgoin

13 juin 2017

Heterogeneous computing

Multiple types of processing elements

- Multicore CPUs
- **GPUs**
- FPGAs
- Cell
- Other co-processors

Each with its own (specific) programming environment

- Programming languages (often subsets of C/C++ or assembly language)
- Compilers
- Libraries
- Debuggers and profilers

GPGPU Programming

Two main frameworks

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

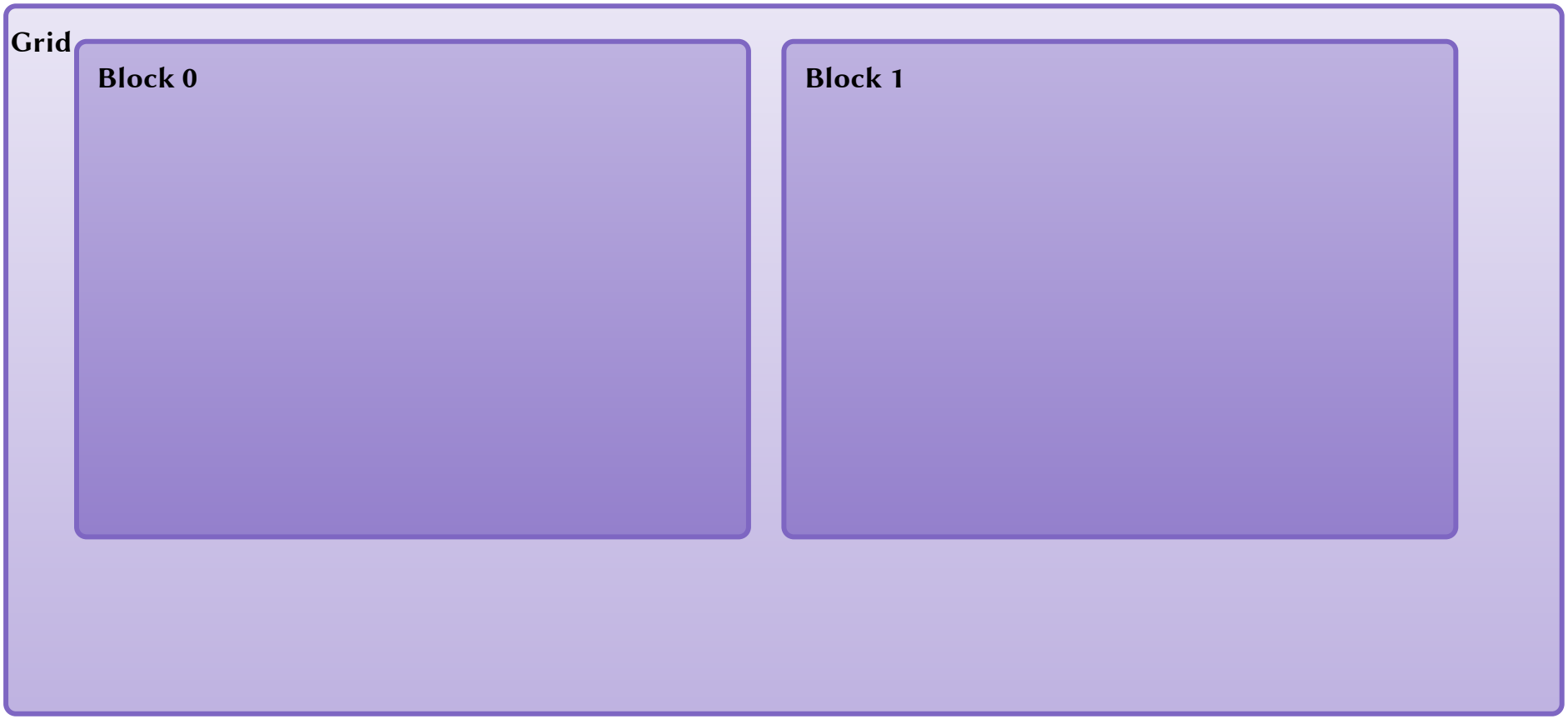
Different languages

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

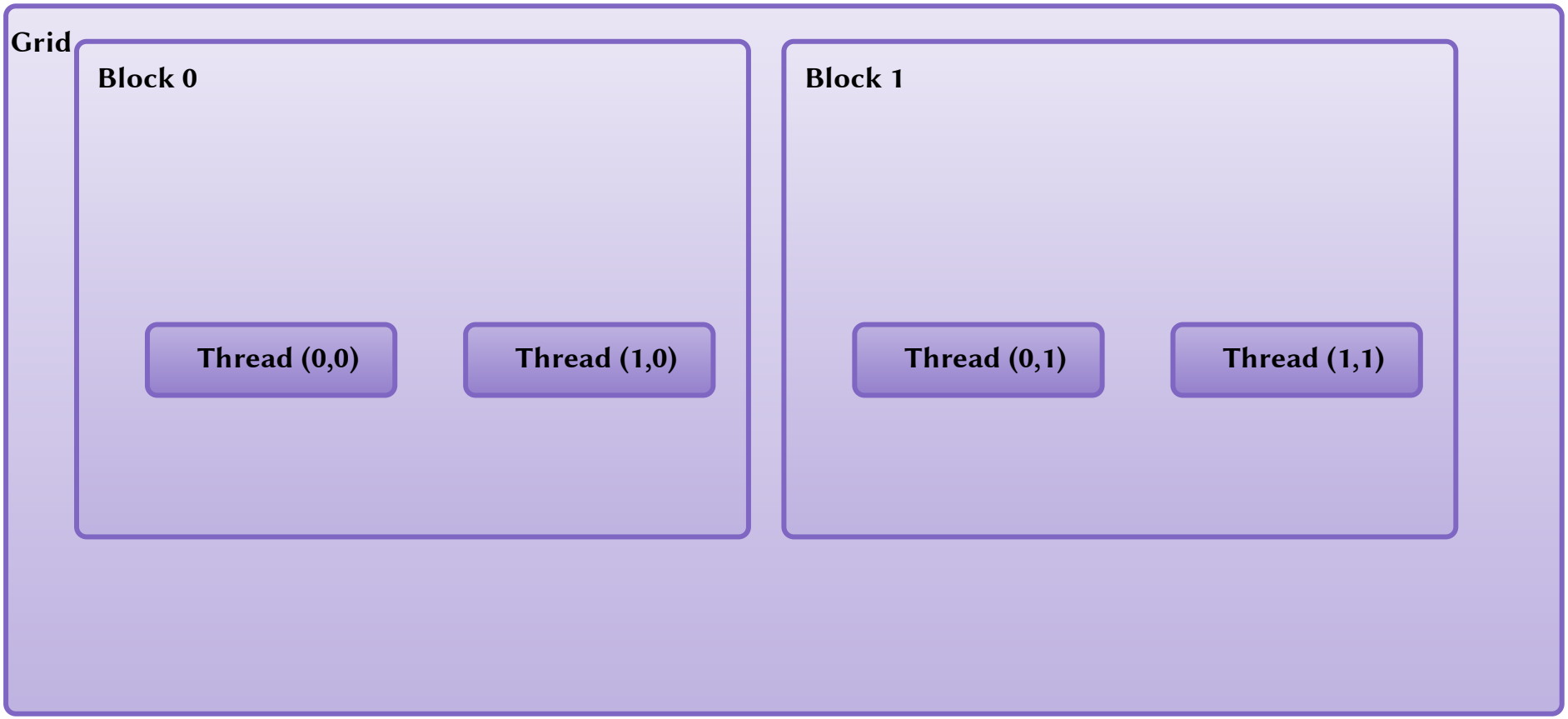
GPGPU programming in practice 1

Grid

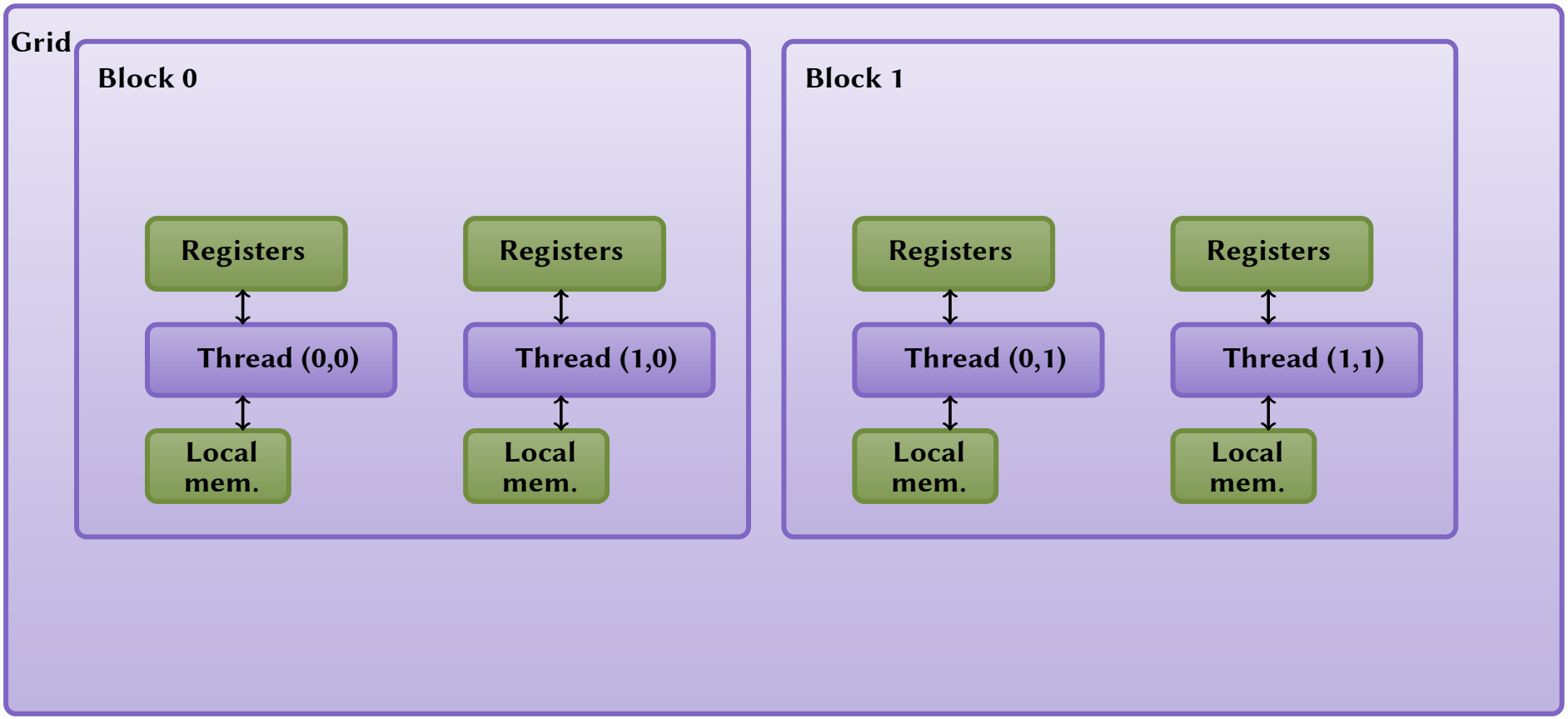
GPGPU programming in practice 1



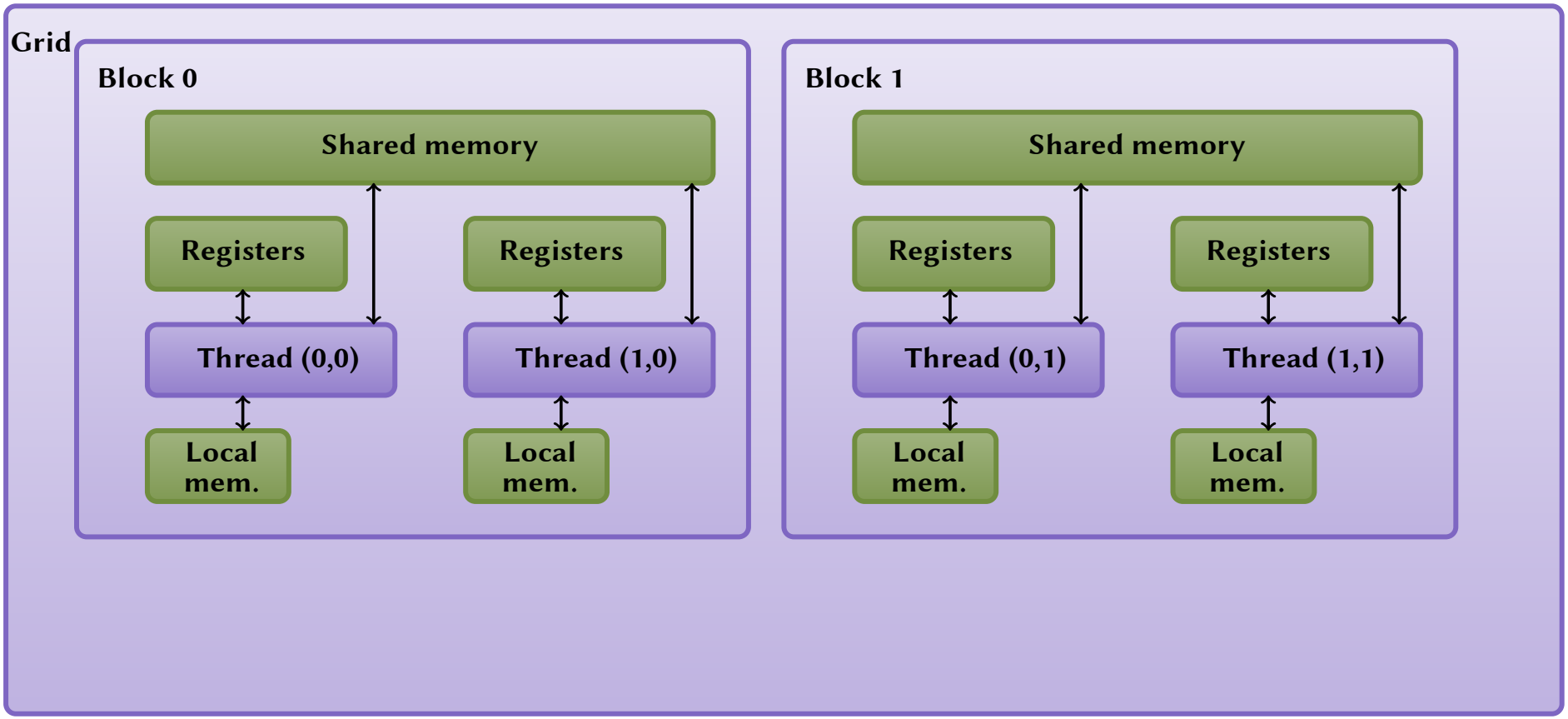
GPGPU programming in practice 1



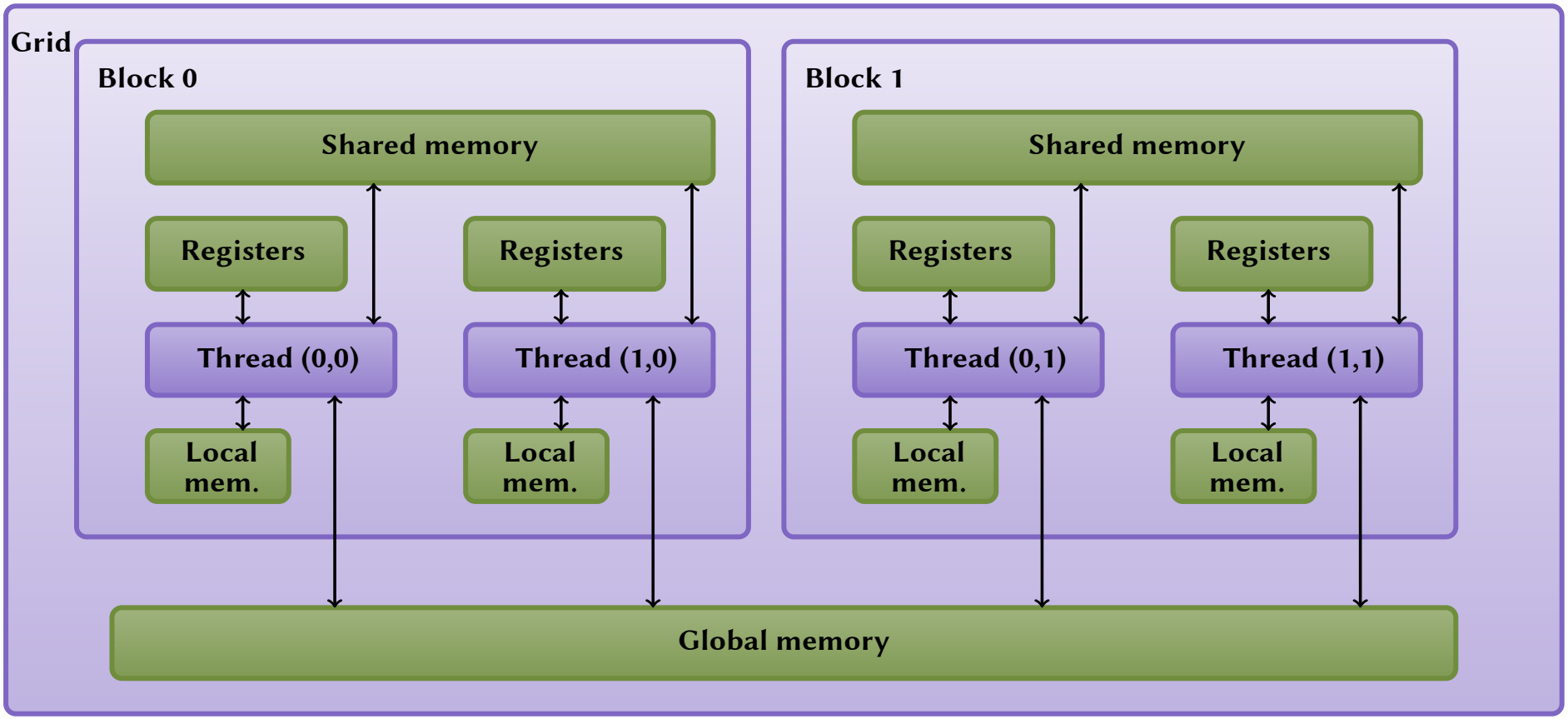
GPGPU programming in practice 1



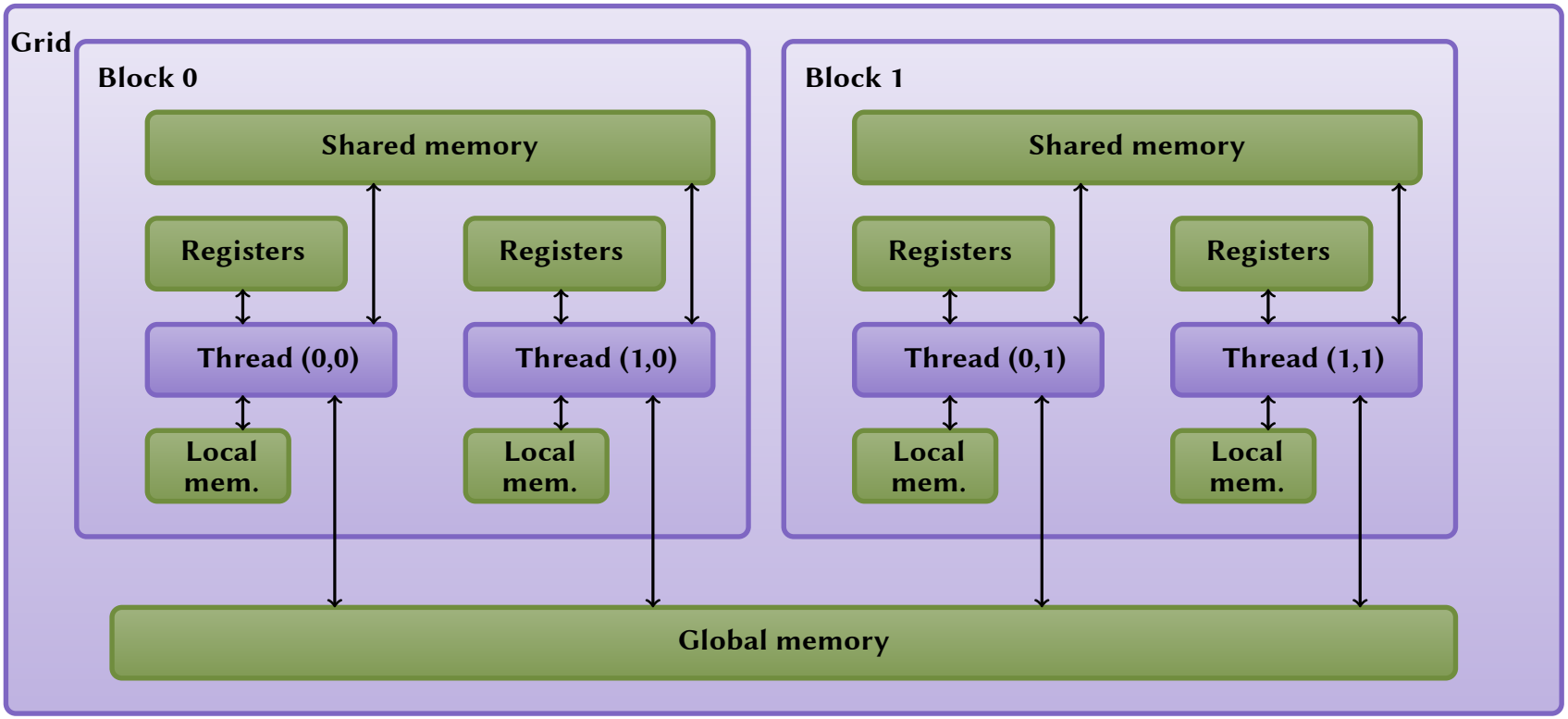
GPGPU programming in practice 1



GPGPU programming in practice 1



GPGPU programming in practice 1



Do not forget transfers between the host and its guests

	CPU-X86 Intel i7 6950x v3	GPU Gamer NVIDIA Titan XP	GPU HPC NVIDIA Tesla P100
Memory bandwidth	77.8GB/s	547.7GB/s	720GB/s

PCI-Express 3.0 maximum bandwidth is 16GB/s

Soon : PCI-Express 4.0 -> 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);
```

Why?

Performance!

		Mem	# Cores	GFLOPS 32	GFLOPS 64	TDP	Price
CPU x86	i7-6950X	128GB	10	< 1000	< 480	140W	$1800
	Ryzen 1800x	64GB	8	460	230	95W	$600
Embedded GPU	Tegra P1?	8GB	256	750	-	<15W	$400
	Mali G71	-	20	371.2	-	-	-
Gaming GPU	Titan Xp	12GB	3840	11366	355	250W	$1400
HPC GPU	Tesla P100	16GB	3584	9340	4670	250W	$15K
	Tesla V100	16GB	5120	14899	7450	300W	$????

Compared to CPUS, GPUs offer higher GFLOPS/$ **and** higher GFLOPS/W

GPGPU computing

Problems

- Complex tools
- Incompatible frameworks
- Verbose languages/libraries
- Low-level frameworks
- Explicit management of devices and memory
- Dynamic compilation

- **Hard to design**
- **Hard to debug**
- **Hard to profile**
- **Very hard to achieve high performance**

Solutions

Libraries

- Linear algebra
- Image processing
- Machine learning ...

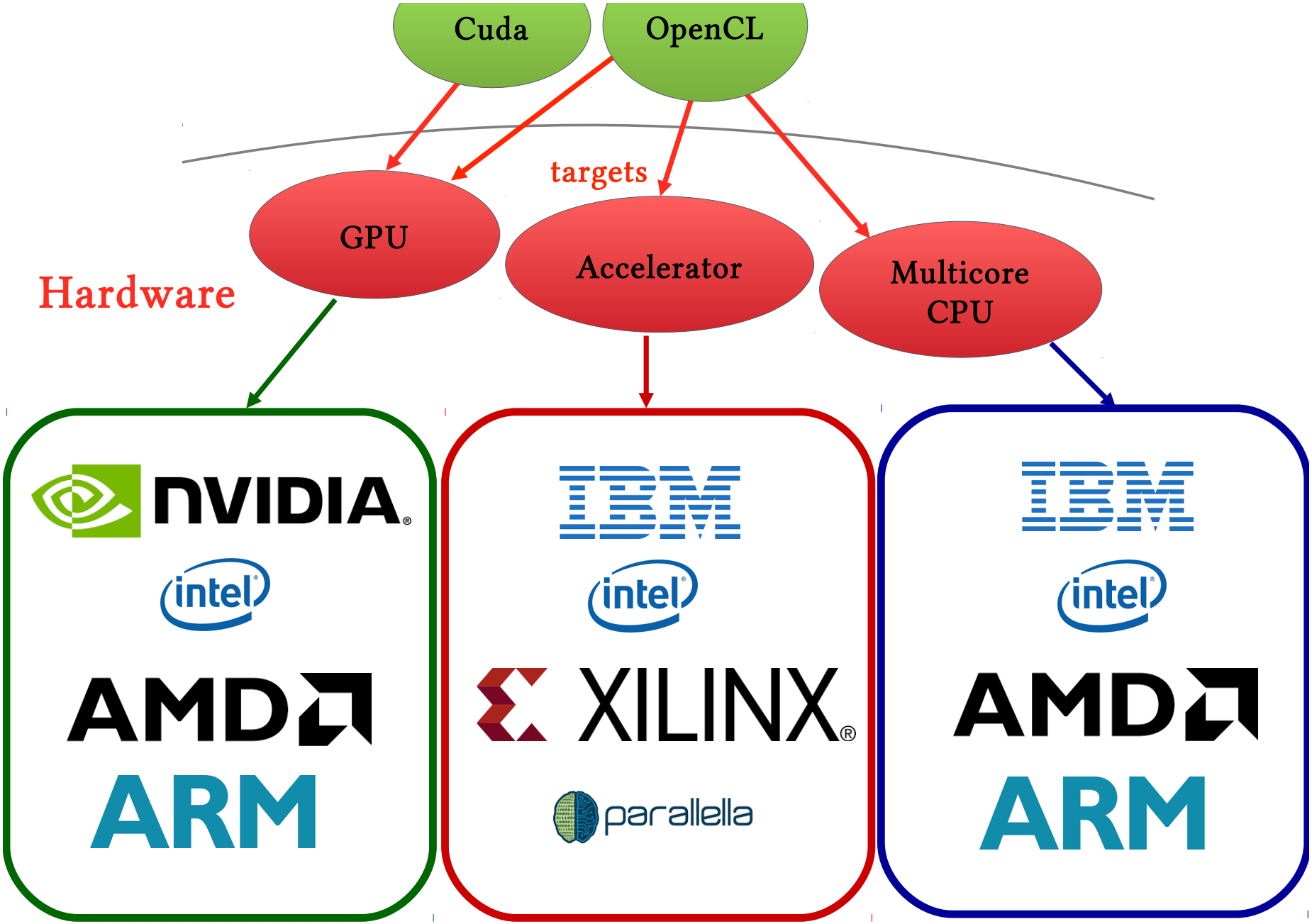
Compiler directives

- OpenMP 4
- OpenACC ...

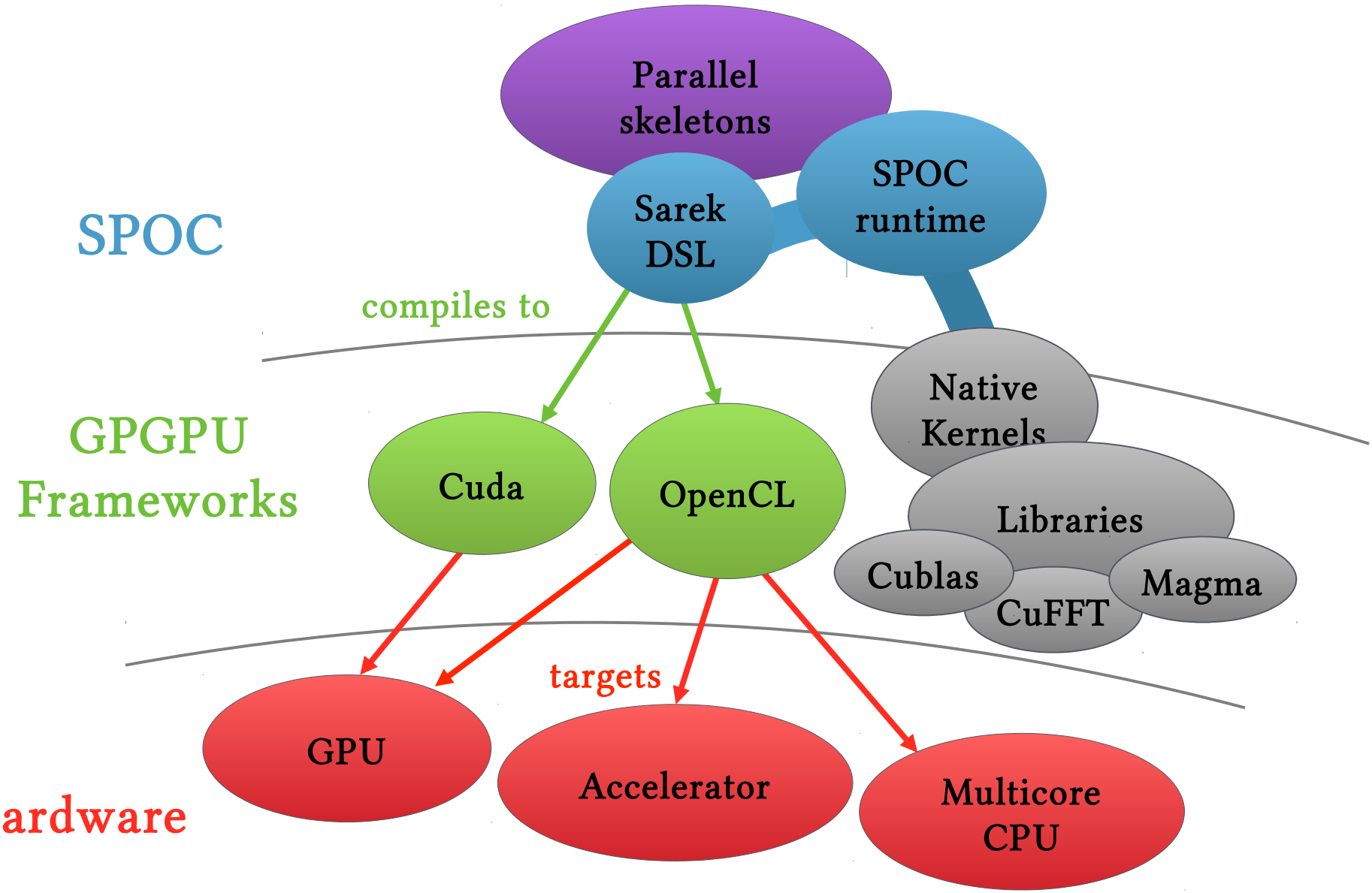
High-level abstractions

- Language extensions
- Domain Specific Languages
- Algorithmic skeletons ...

SPOC : GPGPU Programming with OCaml



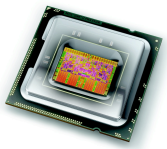
SPOC : GPGPU Programming with OCaml



OCaml

- High-level general-purpose programming 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...

A small example



CPU RAM

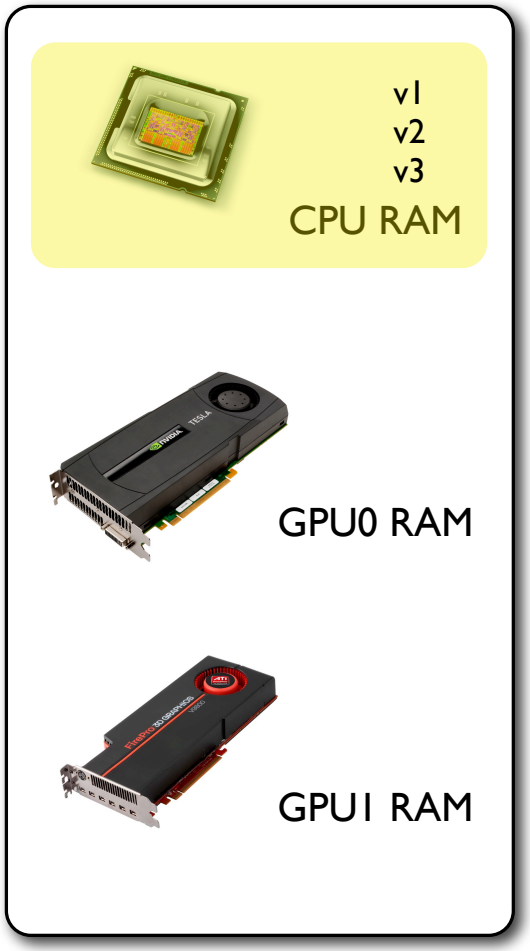
GPU0 RAM

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;
```

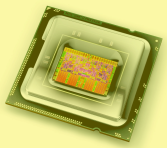
A small example



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

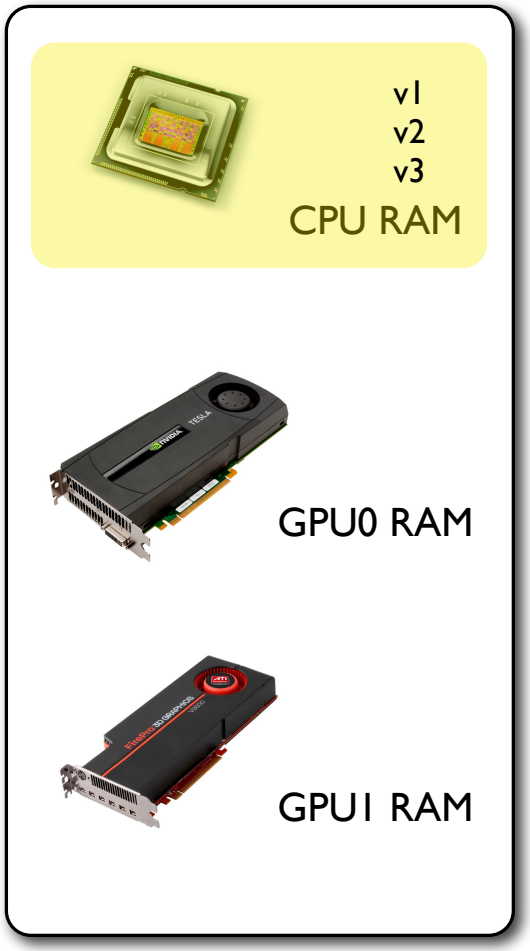
GPU0 RAM

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;
```

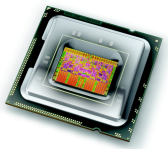
A small example



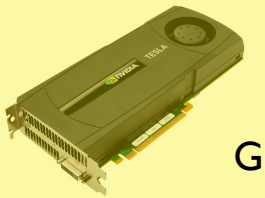
Example

```
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



CPU RAM



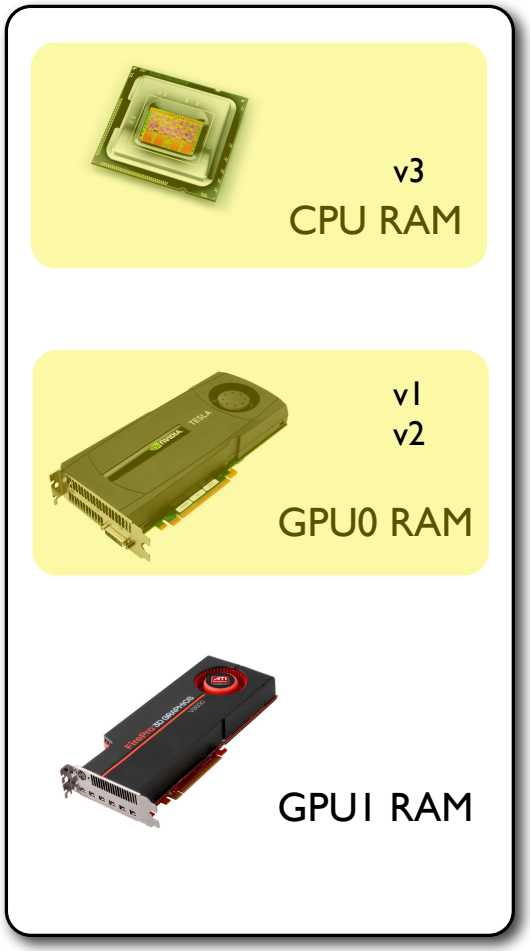
v1
v2
v3
GPU0 RAM

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 : 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
- ML-like data-types
- simple pattern matching
- type inference
- static type checking

Sarek AST is embedded in the OCaml code

- **Static** compilation to OCaml code
- **Dynamic** compilation to Cuda/OpenCL

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 skeletons

Using Sarek

Skeletons 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
```

Solutions

Libraries

- Linear algebra
- Image processing
- Machine learning ...

Compiler directives

- OpenMP 4
- OpenACC ...

High-level abstractions

- Language extensions
- Domain Specific Languages
- Algorithmic skeletons ...

Solutions

Libraries

- Linear algebra
- Image processing
- Machine learning ...

Compiler directives

- OpenMP 4
- OpenACC ...

High-level abstractions

- Language extensions
- Domain Specific Languages
- Algorithmic skeletons ...

New problems

- Written by heterogeneous programming experts
- Dedicated to few (one?) architectures or frameworks
- Limited to specific constructs
- Complex (hidden) scheduling runtime libraries
- Generates most of the heterogeneous (co-processor) code

High level programming heterogeneous applications challenges

From the expert developer point of view

- How to make it portable ?
- How to make performance portable ?
- How will it behave in very heterogeneous systems ?

From the end-user point of view

- How does it work ?
- How to debug the code that uses it ?
- How to optimize the code that uses it ?

Motivation

Provide experts tool developers and end-users feedback :

- they can tie to the code they write
- they can use in very heterogeneous systems

Profile GPGPU programs using SPOC and Sarek

Host part

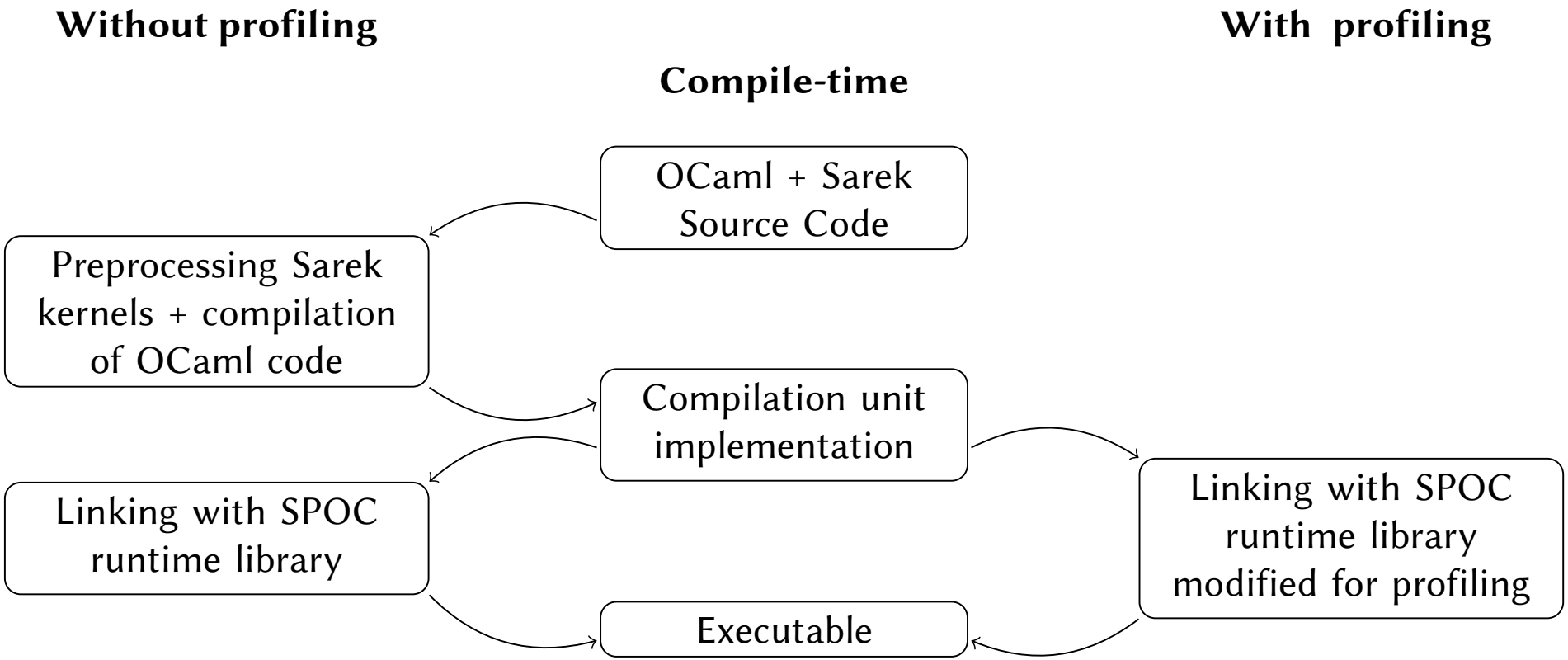
- Where are the vectors ?
- When are transfers triggered ?
- How much time are transfers or kernel calls taking?

Kernel part

- What control path did my threads take ?
- How many computations occurred ?
- Was memory used efficiently ?
- How much time was spent in different parts of the kernels?

- Keep it portable
- Compatible with very heterogeneous systems

Profiling Overview



Profiling Overview

Without profiling

With profiling

Run-time

Detects devices compatible with SPOC

Generates and run native kernel

```
let add = kern v1 v2 v3 n ->
  let i = thread_id_x +
    thread_dim_x * block_id_x in
  if i > n then
    return ();
  else
    v3.[<i>] <- v1.[<i>] + v2.[<i>]

let main () =
  let devs = Devices.init () in
  let v1 = Vector.create float32 n
  and v2 = Vector.create float32 n
  and v3 = Vector.create float32 n
  in
  Kernel.run add
          (v1, v2, v3, n) devs.(0);
  ...
```

Prepares profiling data structures

Fills profiling file with **host** profiling info

Generates and run native kernel instrumented for profiling

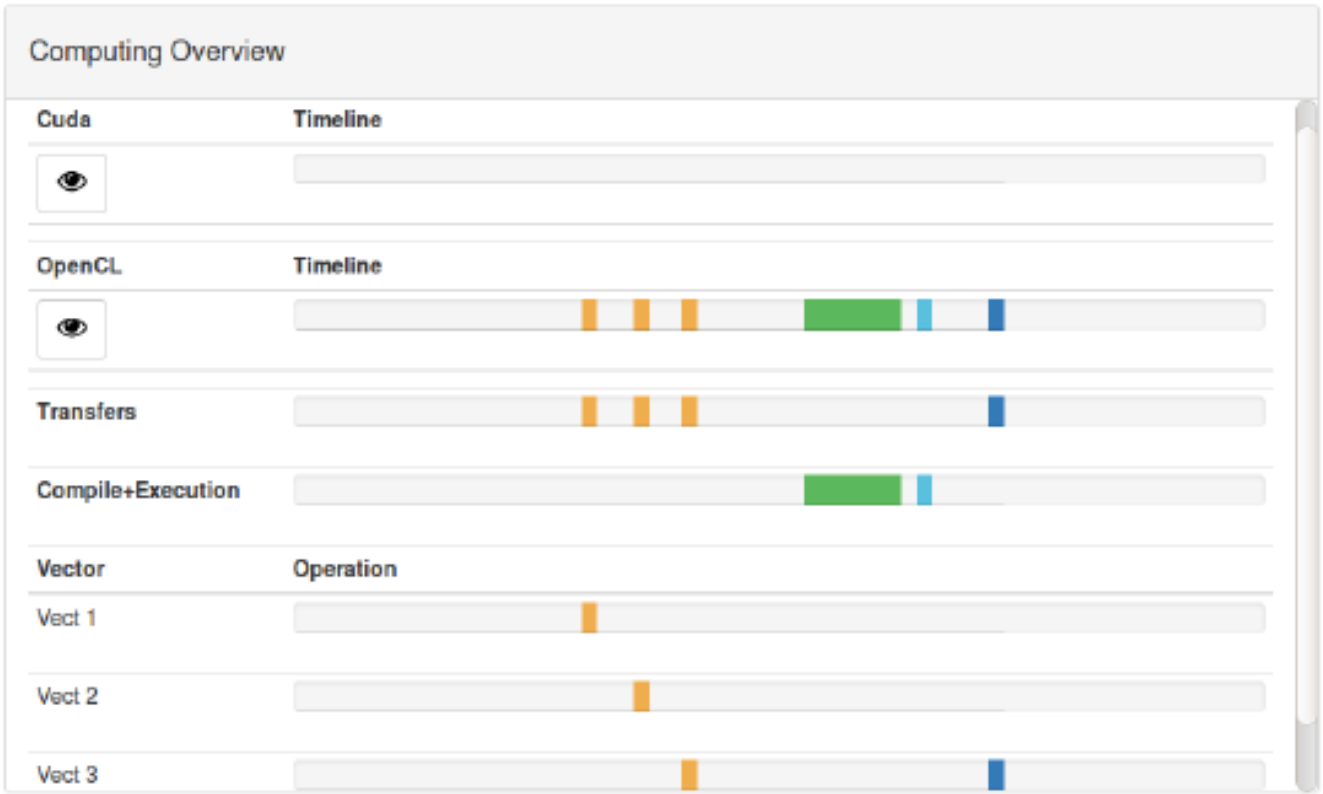
Injects Sarek source commented with **kernel** profiling info into profiling file

Host part profiling

Instrumented SPOC library

- Trace every SPOC runtime operations
- Collect the following info :
  - List of all co-processors (GPUs...) + associated info (name, clock frequency ...)
  - Allocation/Deallocation of vectors in CPU/Co-processor memory
  - Memory transfers (direction, from/to which device, size, duration...)
  - Kernels (compilation/loading/execution time)

Host part profiling : Visualizer



Kernel part profiling

Transform sarek kernel to get profiling information

- Control flow counter
- Memory counters
- Compute operations (FLOPS)

How?

- Add counter vector to co- global memory
- Use atomics operations (mostly atomic_add) offered in both Cuda and OpenCL
- Get updated counters to the CPU after kernel execution
- Compilation Sarek to Sarek with comments using the computed counters

A simple example : Sarek kernel

Sarek kernel used in a k-NN computation

```
let compute = kern trainingSet data res setSize dataSize ->
  let open Std in
  let computeId = thread_idx_x + block_dim_x * block_idx_x in
  if computeId < setSize then (
    let mutable diff = 0 in
    let mutable toAdd = 0 in
    let mutable i = 0 in
    while(i < dataSize) do
      toAdd := data.[<i>] - trainingSet.[<computeId*dataSize + i>];
      diff := diff + (toAdd * toAdd);
      i := i + 1;
    done;
    res.[<computeId>] <- diff)
  else
    return ()
```

A simple example : Generated OpenCL profiling kernel

```
__kernel void spoc_dummy (
                __global unsigned long  * prof_cntrs,
                __global int* trainingSet, __global int* data,
                __global int* res, int setSize, int dataSize ) {
  int tid, diff, toAdd, i;
  tid = ((get_local_id (0)) + ((get_local_size (0)) * (get_group_id (0))));
  bool spoc_prof_cond_6 = (tid < setSize);
  branch_analysis(prof_cntrs, spoc_prof_cond_6, 6);
  if ( spoc_prof_cond_6 ){
    diff = 0 ;
    toAdd = 0 ;
    i = 0 ;
    bool spoc_prof_cond_10 = (i < dataSize);
    branch_analysis(prof_cntrs, spoc_prof_cond_10, 10);
    while ( spoc_prof_cond_10 ){
    toAdd =
      (memory_analysis(prof_cntrs, data+(i), 0, 1) -
       memory_analysis(prof_cntrs,
              trainingSet+(((tid * dataSize) + i)), 0, 1)) ;
    diff = (diff + (toAdd * toAdd)) ;
    i = (i + 1);
    spoc_prof_cond_10 = (i < dataSize);
    while_analysis(prof_cntrs, spoc_prof_cond_10);
    }
    memory_analysis(prof_cntrs, res+(tid), 1, 0);
    res[tid] = diff;;
  }
  else return;
}
```

A simple example : Profiling output

```
(* Profile Kernel *)
kern trainingSet data res setSize dataSize ->
(** ### global_memory stores : 5000 **)
(** ### global_memory loads : 7840000 **)
(** ### Total branches : 10120 **)
(** ### Divergent branches : 0 **)

let mutable tid = (thread_idx_x + (block_dim_x * block_idx_x)) in
(** ### Active threads : 5120 **)
(** ### Branch taken : 5000 **)
(** ### Branch not taken : 120 **)
if (tid < setSize) then
  let mutable diff = 0 in
  let mutable toAdd = 0 in
  let mutable i = 0 in
  (** ### Active threads : 5000 **)
  (** ### Branch taken : 5000 **)
  (** ### Branch not taken : 0 **)
  while i < dataSize do
    toAdd := (data.[<i>] - trainingSet.[<((tid * dataSize) + i)>]);
    diff := (diff + (toAdd * toAdd));
    i := (i + 1);
  done;
  res.[<tid>] <- diff;
else
  return ()
```

Conclusion

GPGPU programming is hard... High-level generative frameworks can help

- Automatic memory transfers
- Data structures
- Type safety

Conclusion

GPGPU programming is hard... High-level generative frameworks can help

- “Simple” code that runs fast and correctly!

Conclusion

GPGPU programming is hard... High-level generative frameworks can help

- “Simple” code that runs fast and correctly!
- But generates lots of complex code that end-users do not understand

Conclusion

GPGPU programming is hard... High-level generative frameworks can help

- “Simple” code that runs fast and correctly!
- But generates lots of complex code that end-users do not understand

Profiling with High-level generative frameworks

- Traces implicit asynchronous events (transfers, kernel launches, ...)
- Provides metrics from the execution of the kernels

That can be tied back to the written code!

Conclusion

GPGPU programming is hard... High-level generative frameworks can help

- “Simple” code that runs fast and correctly!
- But generates lots of complex code that end-users do not understand

Profiling with High-level generative frameworks

- Traces implicit asynchronous events (transfers, kernel launches, ...)
- Provides metrics from the execution of the kernels

That can be tied back to the written code!

Portable and heterogeneous

- Compatible with Cuda/OpenCL frameworks/devices
- Can be used in very heterogeneous systems
- Same level of information for every device

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@univ-orleans.fr

Thanks

E. Chailloux
J-L. Lamotte
A. Doumoulakis