Profiling High Level Heterogeneous Programs

Using the SPOC GPGPU framework for OCaml

Mathias Bourgoin Emmanuel Chailloux Anastasios Doumoulakis

27 mars 2017

Heterogeneous computing

Multiple types of processing elements

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

Each with its own programming environment

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

Heterogeneous computing

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**
- **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 ...

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

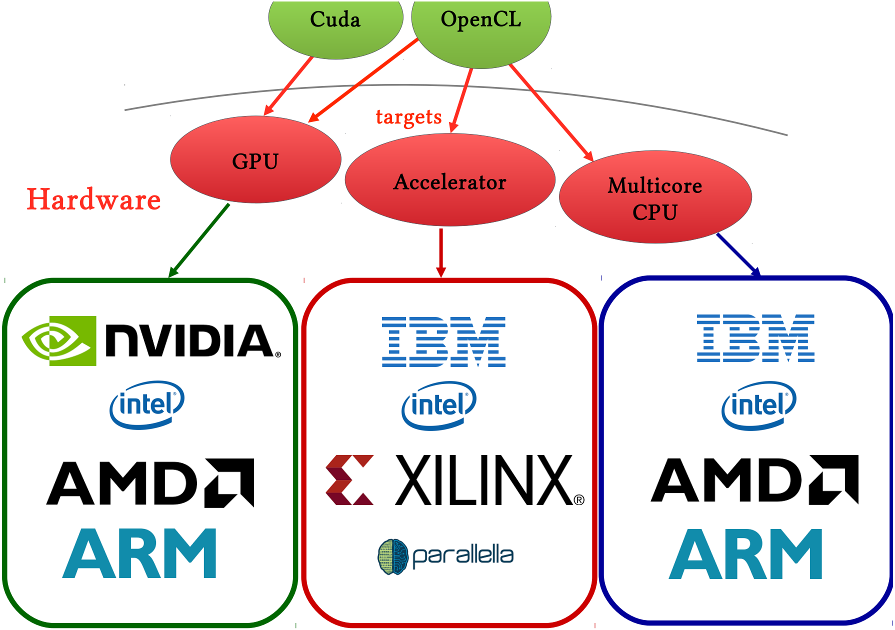
- How does it work ?
- How to debug the code that uses it ?
- How to optimize the code that uses it ?

Motivation

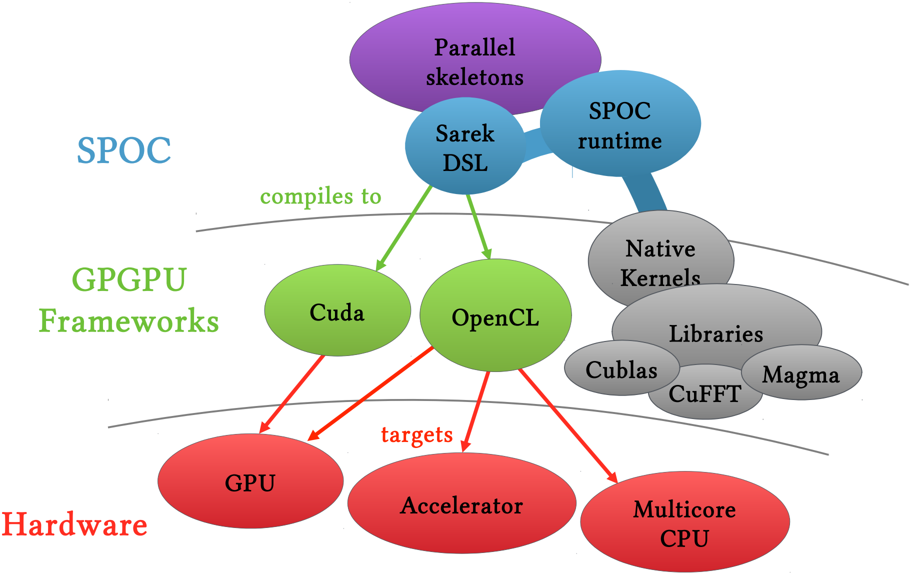
Provide experts tool developers and end-users feedback :

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

SPOC : GPGPU Programming with OCaml



SPOC : GPGPU Programming with OCaml



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

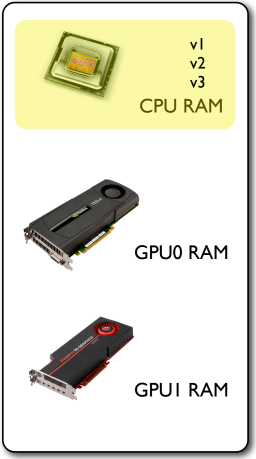
Example

```
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



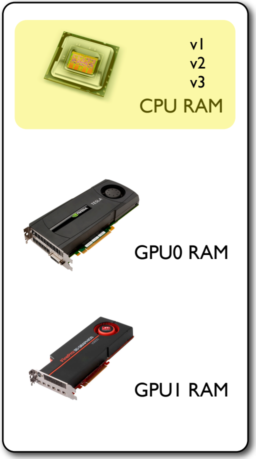
Example

```
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



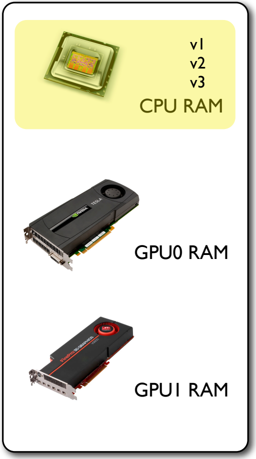
Example

```
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



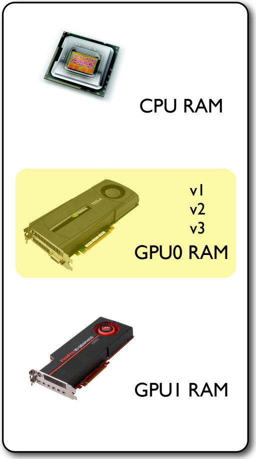
Example

```
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



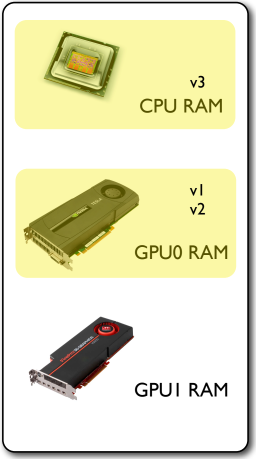
Example

```
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

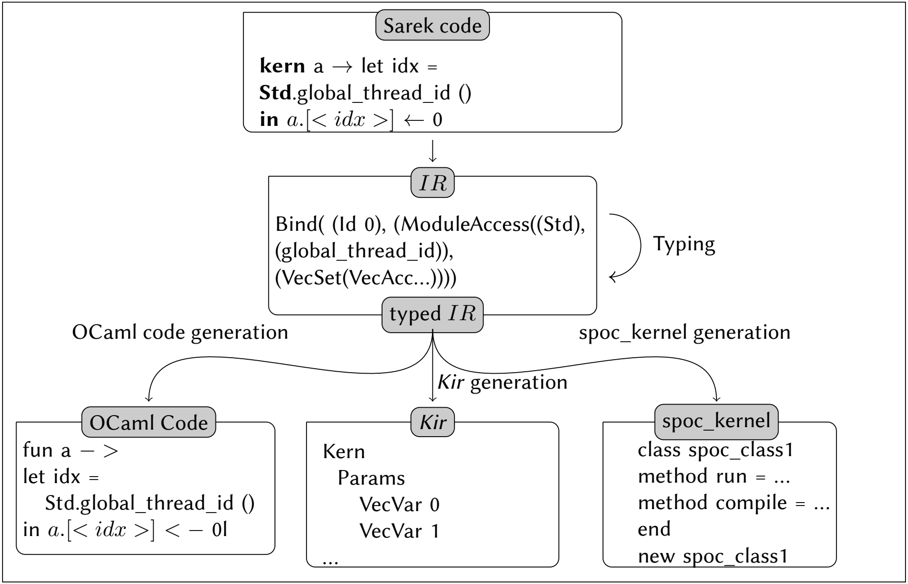
Vector addition with Sarek

```
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>]
```

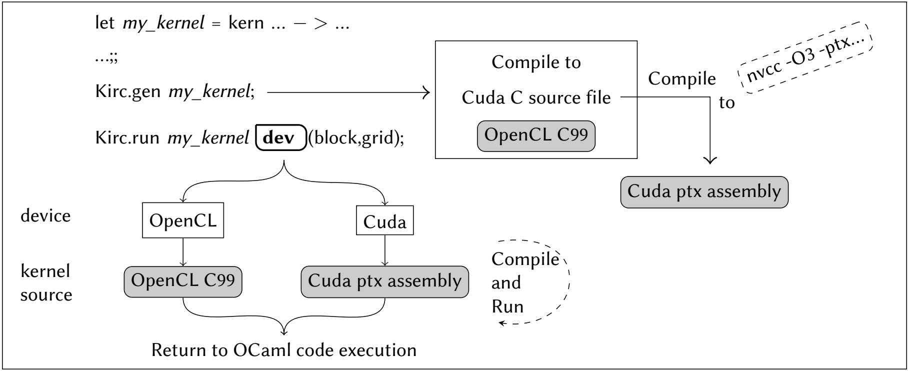
Sarek features

- ML-like syntax
- ML-like data-types
- simple pattern matching
- 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 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
```

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?

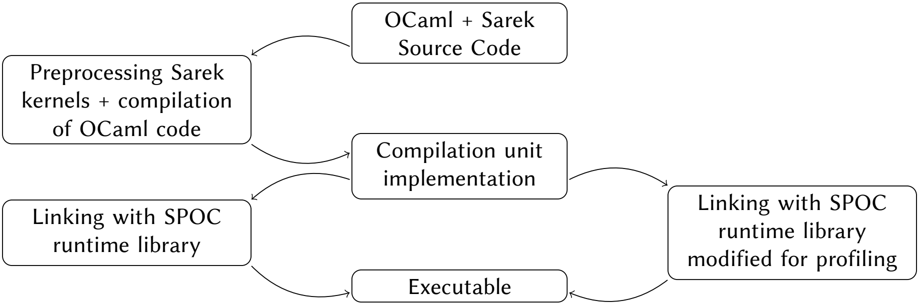
- Keep it portable
- Compatible with very heterogeneous systems

Profiling Overview

Without profiling

With profiling

Compile-time



Profiling Overview

Without profiling

Run-time

With profiling

Detects devices compatible with SPOC

Generates and run native kernel

```
let add = kern v1 v2 v3 n ->
  let i = thread_idx_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
  - Add events to Cuda/OpenCL streams/command queues to get precise measures and stay compatible with SPOC async calls
- Collect the following info :
  - List of all co-processors + 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 : Example

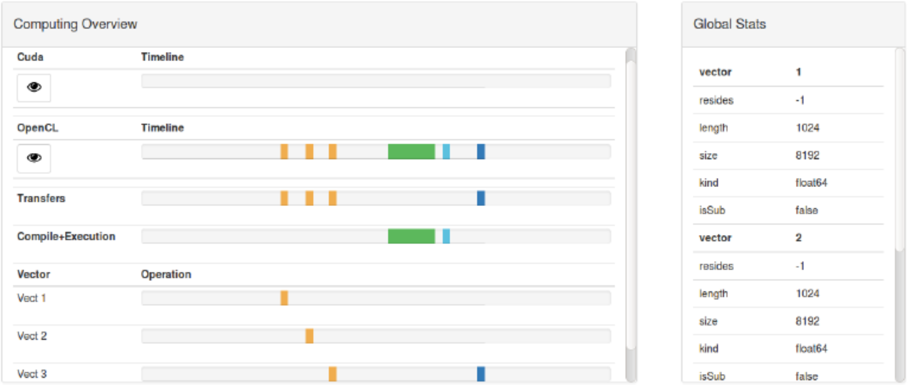
Information collected

- Kind of event (transfer, compilation, execution,...)
- State of event (start, end)
- Time
- Co-processor targeted
- Vector transfered
- Size (in bytes)

Example


```
{
 "type":"execution",
 "desc" : "OPENCL_KERNEL_EXEC",
 "state" : "start",
 "time" : 160304,
 "id" : 40,
 "deviceId" : "1",
},
{
 "type":"execution",
 "state" : "end",
 "time" : 160374,
 "id" : 40,
 "duration":15
},
```

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-processor 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  * profile_counters,
                          __global int* trainingSet, __global int* data,
                          __global int* res, int setSize, int dataSize ) {
  int computeId;
  int diff;
  int toAdd;
  int i;
  computeId = ((get_local_id (0)) +
                          ((get_local_size (0)) * (get_group_id (0)))) ;
  if ( computeId < setSize ){
    spoc_atomic_add(profile_counters+3, 1); // control if
    spoc_atomic_add(profile_counters+0,1); // global mem store
    diff = 0 ;
    toAdd = 0 ;
    i = 0 ;
    while (i < dataSize){
      spoc_atomic_add(profile_counters+1,2); // global mem load
      spoc_atomic_add(profile_counters+2, 1); // control while
      toAdd = (data[i] - trainingSet[((computeId * dataSize) + i)]) ;
      diff = (diff + (toAdd * toAdd)) ;
      i = (i + 1);} ;
    res[computeId] = diff;;
  }
  else{
    spoc_atomic_add(profile_counters+4, 1); // control else
    return  ;
  }
}
```

A simple example : Profiling output

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

let mutable computeId = (thread_idx_x + (block_dim_x * block_idx_x)) in
  if (computeId < setSize) then
  (** ### visits : 5000 **)
    let mutable diff = 0 in
    let mutable toAdd = 0 in
    let mutable i = 0 in
    while i < dataSize do
    (** ### visits : 3920000 **)
      toAdd := (data.[<i>] - trainingSet.[<((computeId * dataSize) + i)>]);
      diff := (diff + (toAdd * toAdd));
      i := (i + 1);
    done;
    res.[<computeId>] <- diff;
  else
  (** ### visits : 120 **)
    return ()
```

Conclusion

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

Conclusion

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

Future work

- Provide more counters
- User defined counters
- Improve Graphical Visualizer (include kernel traces)
- Analyze counters to provide advices to the user

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