GPGPU programming with OCaml

SPOC and Sarek

Mathias Bourgoin - Emmanuel Chailloux - Jean-Luc Lamotte

May 17th, 2013

Outline

GPGPU?

Classic Dedicated GPU Hardware

- Several Multiprocessors
- Dedicated Memory
- Connected to a host through a PCI-Express slot
- Data are transferred between the GPU and the Host using DMA

Current Hardware

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

GPGPU?

GPGPU Usage

- High Performance Computing : better ratio GFLOPS/$ and GFLOPS/W
- Multimedia
- Video Games
- Anywhere there are a lot of data to compute on...

GPGPU Programming

Two main frameworks

- **Cuda**
- **OpenCL**

Different Languages

- To write kernels
  - **Assembly** (ptx, il,...)
  - subsets of **C/C++**
- To manage kernels
  - **C/C++/Objective-C**
  - Fortran
  - Python
  - Scala
  - **Java**
  - **Scilab**
  - ...

Virtual Hardware

- Several Multiprocessors
  - Each contains several Stream Processors (Cores)
  - SIMT (Single Instruction, Multiple Threads)

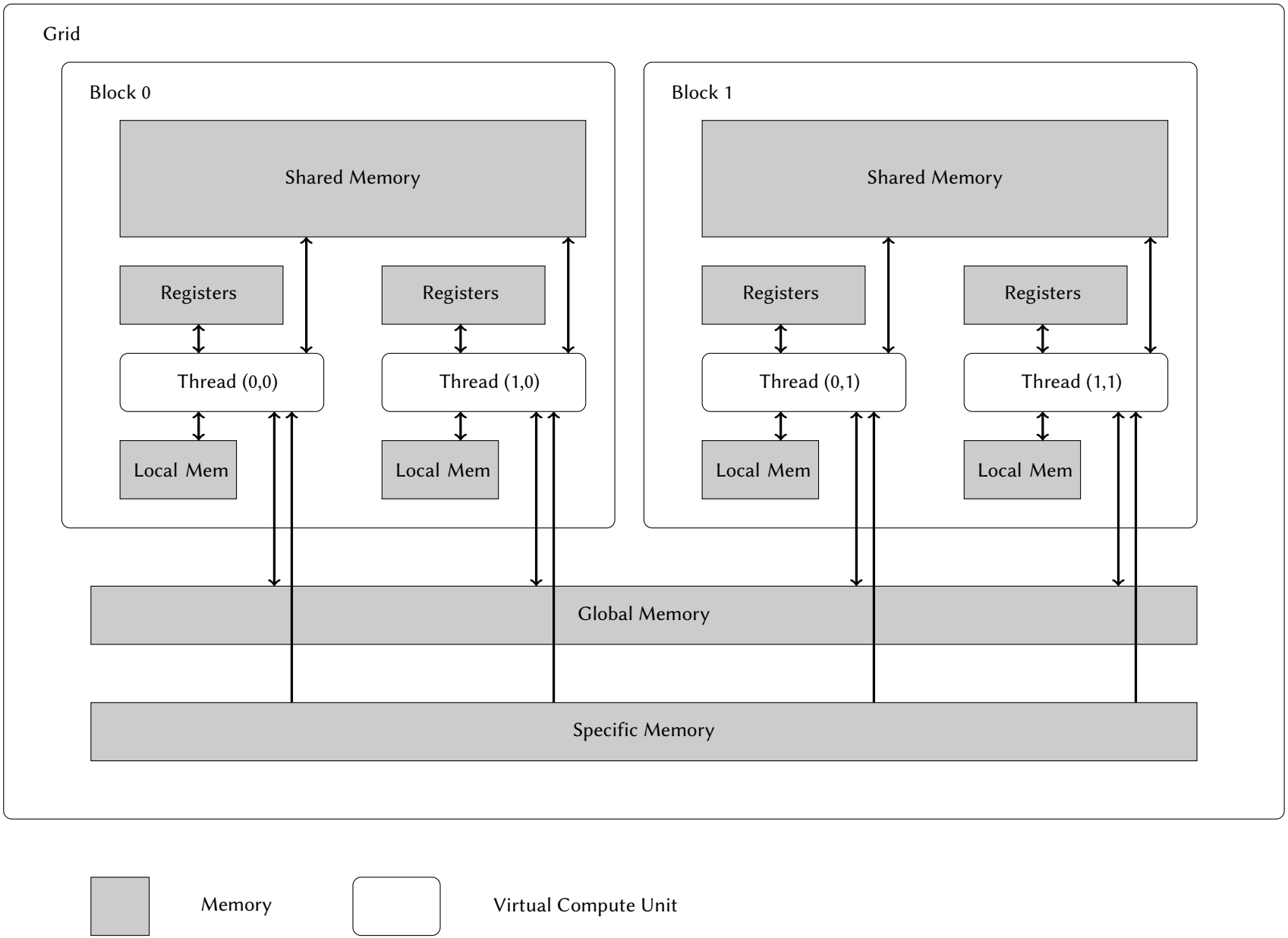
Stream Processing

Given a set of data (a **stream**), a series of operations (**kernel** functions) are applied to each element in the stream.

Different layers of parallelism/memory

- Threads : computation units
- Blocks : grouping threads
- Grid : grouping blocks
- Local Memory : local to a thread
- Shared Memory : shared inside a block
- Global Memory : shared by the whole grid

GPGPU Programming In Practice 1



Different layers of parallelism/memory

- Threads : computation units
- Blocks : grouping threads
- Grid : grouping blocks
- Local Memory : local to a thread
- Shared Memory : shared inside a block
- Global Memory : shared by the whole grid

GPU is a guest!

Don't forget the host memory and **DMA copy** between host and guest

	X86-CPU	Laptop GPU	Desktop GPU		HPC GPU
	i7-3770K	GTX 680M	GTX 680	7970HD	K20X
Mem. Bandwidth	25.6GB/s	115.2 GB/s	192.2GB/s	264GB/s	250GB/s

Max PCI-Express 3.0 Bandwidth 16GB/s

Vector Addition

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

GPGPU Programming In Practice 2

A Small Example in 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);
```

OCaml

- High-Level 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...

OCaml

- High-Level 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...

Motivations

OCaml and GPGPU complement each other

GPGPU frameworks are

- Highly Parallel
- Architecture Sensitive
- Very Low-Level

OCaml is

- Mainly Sequential
- Multi-platform/architecture
- Very High-Level

Idea

- Allow OCaml developers to use GPGPU with their favorite language.
- Use OCaml to develop high level abstractions for GPGPU.
- Make GPGPU programming safer and easier

Overview

Main Objectives

Goals

- Allow use of Cuda/OpenCL frameworks with OCaml
- Abstract these two frameworks
- Abstract memory transfers
- Use OCaml type-checking to ensure kernels type safety
- Propose Abstractions for GPGPU programming

Host side solution

M. Bourgoin *et al.*, HLPGPU 2012

M. Bourgoin *et al.*, PPL, 2012

SPOC: Abstracting frameworks

Our choice

- **Dynamic linking**.
- The Cuda implementation uses the Cuda Driver API instead of the Runtime Library (lower level API, does not need the cudart library which is only provided with the Cuda SDK).

Compilation doesn't need any specific hardware
(no need of a Cuda/OpenCL compatible Device) or SDK.

Allows

- development **for multiple architectures from a single system**;
- executables to use **any OpenCL/Cuda Devices conjointly**;
- distribution of a **single executable for multiple architectures**.

SPOC: Abstracting Transfers

Automatic Transfers

Vectors automatically move from CPU to Devices

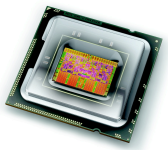
- When a CPU function uses a vector, SPOC moves it to the CPU RAM
- When a kernel uses a vector, SPOC moves it to the Device Global Memory
- Unused vectors do not move
- SPOC allows users to explicitly force transfers

OCaml memory manager

Vectors are managed by the OCaml memory manager

- **Automatic allocation(s)**
- The GC **automatically frees** vectors (on the CPU as well as on Devices)
- Allocation failure during a transfer triggers a collection

A Little Example



CPU RAM

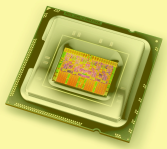
GPU0 RAM

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 = 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 Little Example



v1
v2
v3
CPU RAM

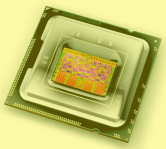
GPU0 RAM

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 Little Example



v1
v2
v3
CPU RAM

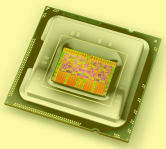
GPU0 RAM

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 Little Example



v1
v2
v3
CPU RAM

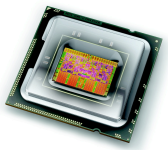
GPU0 RAM

GPU1 RAM

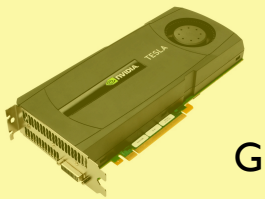
Example

```
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 Little Example



CPU RAM



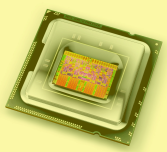
v1
v2
v3
GPU0 RAM

GPU1 RAM

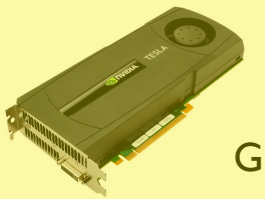
Example

```
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 Little Example



v3
CPU RAM



v1
v2
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;
```

Overview

How to express kernels

What we want

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

Two Solutions

A DSL for OCaml : Sarek

- Easy to express
- Easy transformation from OCaml
- Safer

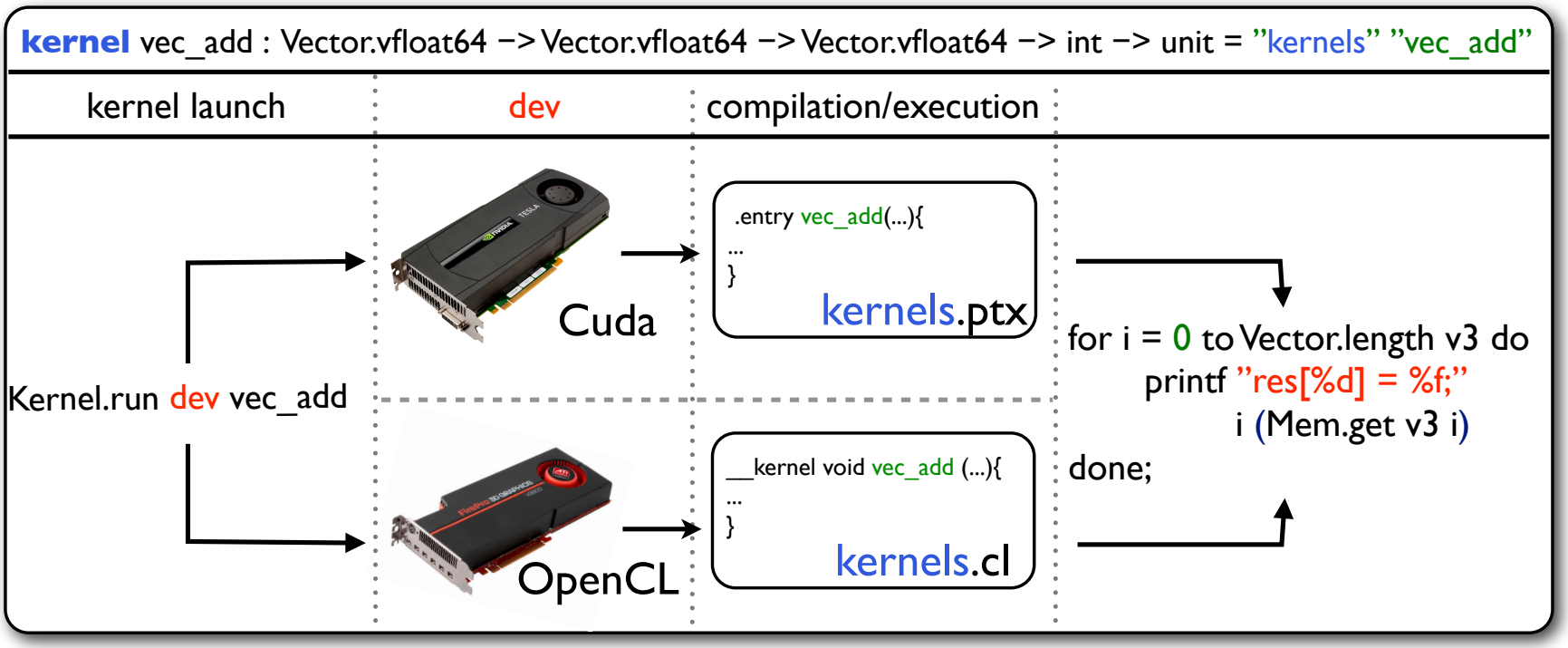
Interoperability with Cuda/OpenCL kernels

- Higher optimisations
- Compatible with current libraries
- Less safe

External Kernels

Type-Safe Kernel Declaration

- Static arguments types checking (compilation time)
- Kernel.run compiles kernel from source (.ptx / .cl)



Errors

Type-Checking: Error at compile-time

```
kernel vector_sum: Vector.vfloat64 -> unit = "my_file" "kernel_sum"
let v = Vector.create Vector.float32 1024 in
  Kernel.run device (block, grid) vector_sum v;
```

Type-Checking : Correct

```
kernel vector_sum: Vector.vfloat64 -> unit = "my_file" "kernel_sum"
let v = Vector.create Vector.float64 1024 in
  Kernel.run device (block, grid) vector_sum v;
```

Exceptions

SPOC raises OCaml exceptions when

- Kernel compilation/execution fails
- Not enough memory on devices

Sarek : Stream ARchitecture using Extensible Kernels

Sarek Vector Addition

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

OpenCL Vector Addition

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

M. Bourgoin *et al.*, HLPP 2013

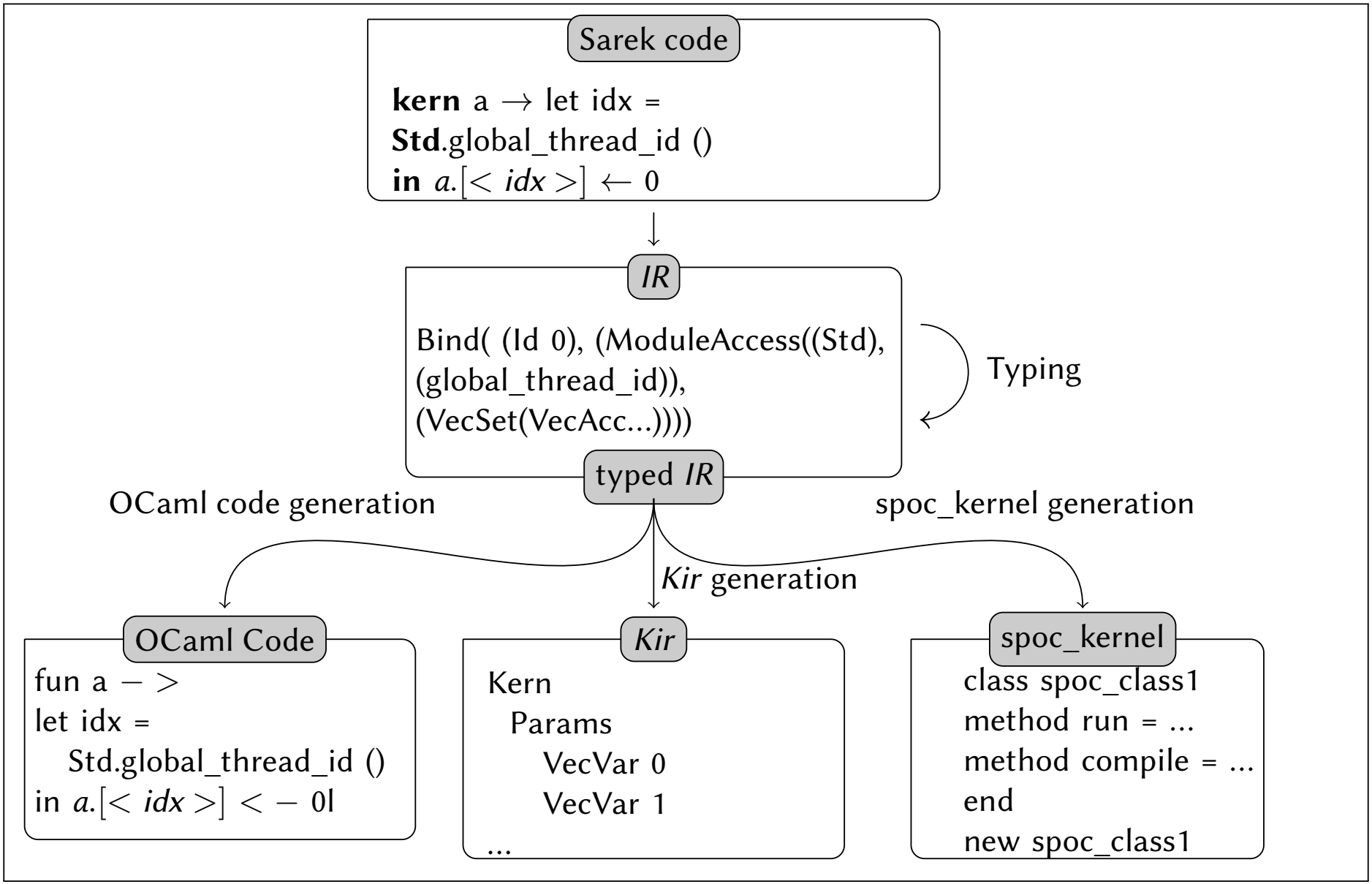
Sarek Vector Addition

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

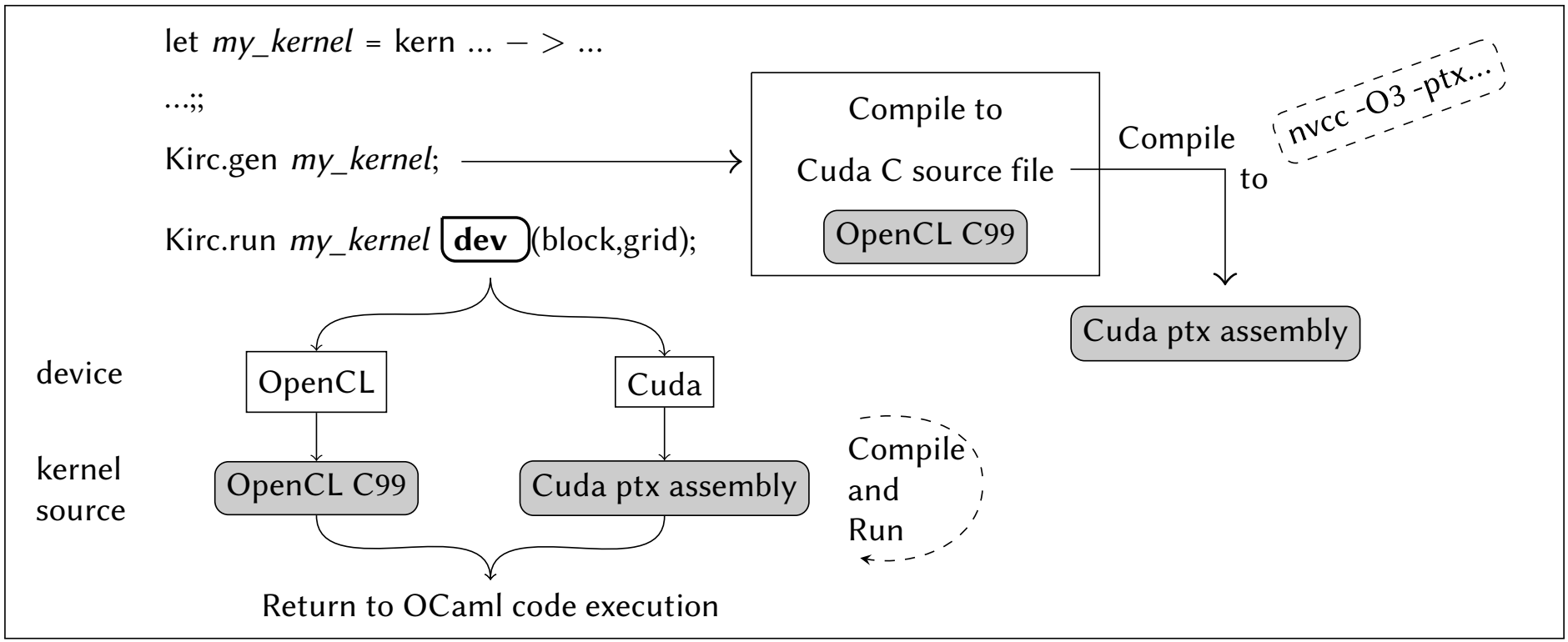
Sarek offers

- ML-like syntax
- Type inference
- Static type checking
- Static compilation to OCaml code
- Dynamic compilation to Cuda and OpenCL

Sarek Static Compilation



Sarek Dynamic Compilation



Vector Addition

SPOC + Sarek

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

Overview

Kernel Composition

Composition

Compose multiple kernels to express algorithms

Benefits

- Ease programming
- Allow new automatic optimizations

Problems

To be composable, kernels must have an input/output

Parallel Skeletons Using External Kernels

Using External kernels

Describe Skeletons as :

- an external kernel
- an execution environment
- an input
- an output

Two running functions:

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

M. Bourgoin *et al.*, OpenGPU Workshop, 2012

Skeletons

Two kinds of Skeletons

- $map : kernel \rightarrow env \rightarrow vector \rightarrow skeleton$
- $reduce : kernel \rightarrow env \rightarrow vector \rightarrow skeleton$

Skeleton Composition

- $pipe : skeleton \rightarrow skeleton \rightarrow skeleton$
- $par : skeleton \rightarrow skeleton \rightarrow skeleton$

Running Function

$run : skeleton \rightarrow device \rightarrow vector \rightarrow vector$

Parallel Skeletons Using Sarek

Using Sarek

Skeletons are functions transforming Kir AST :
Example:
map (kern a -> b) =>
Scalar computations $('a \rightarrow 'b)$ are transformed
into vector ones $(('a, 'c)\ vector \rightarrow ('b, 'd)\ vector)$.

Currently

Sarek skeletons generates spoc_kernels compatible
with external kernel skeletons

```
val map : ('a -> 'b) kirc_kernel -> spoc_kernel ->
  spoc_kernel * (('a, 'c) vector -> ('b, 'd) vector) kirc_kernel
```

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
```

Skeletons

```
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[i])))))
           (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
```

Skeletons

```
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[i])))))
           (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
```

Skeletons

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

Benefits

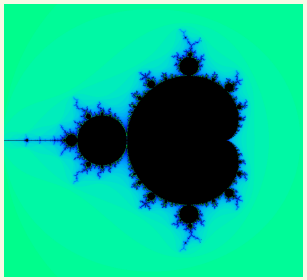
Benefits

- Explicitly describe relation between kernels/data
- Automatic blocks/grids mapping on GPUs
- Optimize data location (GPUs/CPU)
- Optimize automatic transfers

Overview

Benchmarks : Toy Examples

Mandelbrot



Computation handled through SPOC
Graphics handled through OCaml graphics (by the CPU)

Matmult

Naive matrix multiply
Over two 2000×2000 matrices

Using unoptimised kernels (non vectorized, no shared memory, etc)

Results : Toy Examples

Sample / Device	OCaml Sequential (s)	C2070 Cuda (s)		GTX 680 Cuda (s)		AMD6950 OpenCL (s)		i7-3770 (Intel OpenCL) (s)	
$\text{Mandelbrot}_{\text{ext}}$	474.5	5.9	×80.4	4.0	×118.6	4.9	×96.8	6.0	×79.1
$\text{Mandelbrot}_{\text{Sarek}}$		7.0	×67.8	4.8	×98.8	5.6	×84.7	7.2	×65.9
$\text{Matmult}_{\text{ext}}$	85.0	1.3	×65.4	1.7	×50.0	0.3	×283.3	4.8	×17.7
$\text{Matmult}_{\text{Sarek}}$		1.7	×50.0	2.1	×40.5	0.3	×283.3	6.2	×13.7

Using Sarek offers very high performance for data parallel programs
Using external kernels allows to achieve higher optimizations

Results : Using Composition

Power Iteration : Using SPOC

SPOC Device	Theoretical GFLOPS(DP)	Framework	Time (s)	Speedup
i7-3770	32 (1 core)	OCaml (1 Thread)	637	×1
Tesla C2070	515	SPOC (Cuda)	150	×4.24
Radeon HD 6950	562.5	SPOC (OpenCL)	101	×6.31

Using SPOC already improves performance

Results : Using Composition

Power Iteration : Using Skeletons

Skeletons	Framework	Time (s)	Speedups	
Devices			OCaml	SPOC
i7-3770	OCaml	637	-	-
Tesla C2070	Cuda	135	×4.72	×1.11
Radeon HD 6950	OpenCL	81	×7.86	×1.25

Skeletons help increase performance *via* automatic grid/bloc mapping

Results : Using Composition

Power Iteration : Composing Skeletons

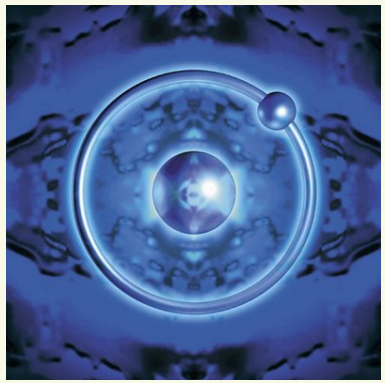
Composition	Framework	Time (s)	Speedups		
Devices			OCaml	SPOC	Skeletons
i7-3770	OCaml	637	-	-	-
Tesla C2070	Cuda	133	×4.79	×1.13	×1.02
Radeon HD 6950	OpenCL	74	×8.61	×1.36	×1.09

Pipe composition allows transfer overlapping by computation, increasing performance even further

Real-world Example

PROP

- Included in the 2DRMP[ab] suite
- Simulates e^- scattering in H-like ions at intermediates energies
- PROP Propagates a $\mathcal{R}$-matrix in a two-electrons space
- Computations mainly implies matrix multiplications
- Computed matrices grow during computation
- Programmed in Fortran
- Compatible with sequential architectures, HPC clusters, super-computers



[a] NS Scott, MP Scott, PG Burke, T. Stitt, V. Faro-Maza, C. Denis, and A. Maniopoulou. 2DRMP : A suite of two- dimensional R-matrix propagation codes. Computer Physics Communications, 2009

[b] HPC prize for Machine Utilization, awarded by the UK Research Councils' HEC Strategy Committee, 2006

PROP : GPGPU Implementation

First modification (*Caps-Entreprise*)

- Matrix multiplication ported to Cuda using the HMPP Compile
- Propagation equation modified to handle bigger matrices
- Lower tranfers/computation ratio but still many computations made by the CPU

Second modification (*LIP6*)

- Reduce transfer by performing all propagation computation on the GPGPU
- Overlaps transfers with computations over different sections of the $\mathcal{R}$-matrix
- Uses Cublas and Magma library to perform computations
- Uses a C glue to bind Fortran code with Cuda

Porting PROP In OCaml

Our approach

- Translation of the computing part only (leaving I/O and initialisation to Fortran)
- Binding of a subset of the Cublas and Magma Library for OCaml (using SPOC)
- C glue between Fortran and OCaml

Result

- A program mixing Fortran, C, OCaml using Sarek
- Dramatic reduction of the code size
- No more transfers!!

Results: PROP

Running Device	Running Time	Speedup / Fortran		
		CPU 1	CPU 4	GPU
Fortran CPU 1 core	4271.00s (71m11s)	1.00	0.51	0.22
Fortran CPU 4 core	2178.00s (36m18s)	1.96	1.00	0.44
Fortran GPU	951.00s (15m51s)	4.49	2.29	1.00
OCaml GPU	1018.00s **(16m58s)**	4.20	2.14	0.93
OCaml (+ Sarek) GPU	1195.00s **(19m55s)**	3.57	1.82	0.80

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

Ready for the real world...

Overview

Using SPOC with Multicore CPUs?

Why?

OCaml cannot run parallel threads...
Multiple "solutions" have been considered :

- New runtime/GC $\Rightarrow$ OC4MC[a] experiment ?
- Automatic forking $\Rightarrow$ ParMap?
- Extension for distributed computing $\Rightarrow$ JoCaml?
- Probably many other solutions (new compiler?, parallel virtual machine?, etc)

[a] M. Bourgoin, P. Wang *et al.*, IFL 2009

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

Overview

Conclusion

SPOC : Stream Processing with OCaml

- OCaml library
- Unifies Cuda/OpenCL
- Offers automatic transfers
- Is compatible with current high performance libraries

Sarek : Stream ARchitecture using Extensible Kernels

- OCaml-like syntax
- Type inference
- Easily extensible via OCaml code

Skeletons and Composition

- Ease programming
- Allow automatic optimization

Conclusion

Results

- Great performance
- Great for both GPU and CPU
- Nice playground for further abstractions

Future Work

SPOC

- CPU optimization : no more copy
- Test on new architectures : ARM SOC, Xeon Phi

Sarek

- Custom types, Function declarations, Recursion, Exceptions, ...
- Allow shared memory access
- Optimize code generation
- Auto tuning for different GPGPU architectures

Skeletons and Composition

- Use Sarek to provide more skeletons
- e.g., Allow kernel splitting/merging

Thanks

Emmanuel Chailloux
Jean-Luc Lamotte

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

For more information
mathias.bourgoin@lip6.fr