Retour d'expérience : portage d'une application haute-performance vers un langage de haut niveau

ComPAS/RenPar 2013

Mathias Bourgoin - Emmanuel Chailloux - Jean-Luc Lamotte

16 Janvier 2013

Our Goals

Globally

- Allow GPGPU programming with the OCaml language
- Use OCaml to offer abstractions for GPGPU programming
- **Test our approach with a real HPC use case**

Specifically

- Translate a known HPC software from Fortran+Cuda to OCaml
- Check performance and memory usage
- Advantages of using a high level language

Real World HPC Use-Case

PROP

- Included in the 2DRMP[a] suite
- Simulates scattering of electrons 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

PROP : GPGPU Implementation

First modification (*Caps-Entreprise*)

- Matrix multiplication ported to Cuda using the HMPP Compiler
- 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
- Cublas and Magma library to perform computations
- C glue to bind Fortran code with Cuda

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 For Scientists

Advantages

- Interactive toplevel : easy to test and perform simple computations
- Static type checking
- Functionnal programming
- First-class array and list literals
- Algebraic data types with pattern matching : efficient tree manipulation
- Parametric polymorphism
- Efficient garbage collection
- Fast compilation
- Easy to learn (especialy for “non programmer” scientists)

Drawbacks

- Garbage collector prevents threads from running in parallel
- Boxing/Unboxing of numeric data
- No operator overloading

SPOC (*Stream Processing with OCaml*)

- Abstract both Cuda/OpenCL frameworks
- Abstract memory and transfers
- Use OCaml type-checking to ensure kernels type safety

SPOC: Abstracting frameworks

Our choice

- **Dynamic linking**
- **Unify** both frameworks

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 explicitely 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

Kernels

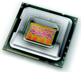
Type-Safe Kernel Declaration

- Static arguments types checking (compilation time)
- Spoc dynamically compiles kernels from source (.ptx / .cl)

Example

```
kernel vector_add: Vector.vfloat64 -> Vector.vfloat64 ->
                   Vector.vfloat64 -> int -> unit = "my_file" "kernel_add"
```

A Little Example



CPU RAM

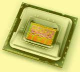
GPU0 RAM

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 v3 v1 v2 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 dev.(0) (block,grid) k;
  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

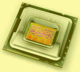
GPU0 RAM

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 v3 v1 v2 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 dev.(0) (block,grid) k;
  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

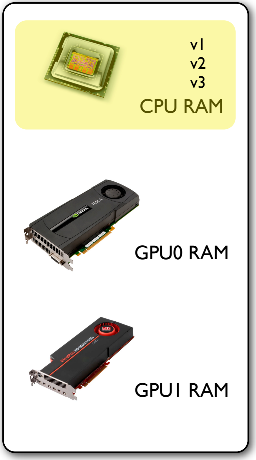
GPU0 RAM

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 v3 v1 v2 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 dev.(0) (block,grid) k;
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[\%d] = \%f; " i v3.[<i>]
  done;
```

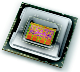
A Little Example



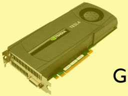
Example

```
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 v3 v1 v2 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 dev.(0) (block,grid) k;
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[\%d] = \%f; " i v3.[<i>]
  done;
```

A Little Example



CPU RAM



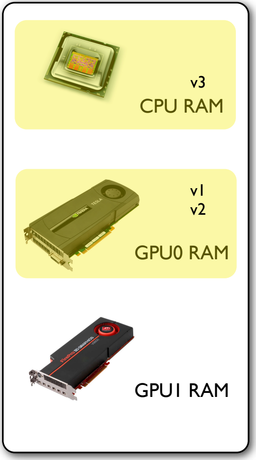
v1
v2
v3
GPU0 RAM

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 v3 v1 v2 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 dev.(0) (block,grid) k;
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[\%d] = \%f; " i v3.[<i>]
  done;
```

A Little 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 v3 v1 v2 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 dev.(0) (block,grid) k;
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[\%d] = \%f; " i v3.[<i>]
  done;
```

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 and Cuda
- Huge reduction of the code
- No more transfers!!

Benchmarks : Efficiency

Data Set	Size of the local $\mathcal{R}$-matrix	Size of the global final $\mathcal{R}$-Matrix	Number of scattering energies
Small	90x90	360x360	6
Medium	90x90	360x360	64
Large	383x383	7660x7660	6

Figure : Data sets

Small	Running Time	Speedup /Fortran	Speedup /OCaml
GPU Fortran	1m49s	100%	198%
GPU OCaml	3m36s	50%	100%

Medium	Running Time	Speedup /Fortran	Speedup /OCaml
GPU Fortran	10m70s	100%	241%
GPU OCaml	26m58s	41%	100%

Figure : Performances

Benchmarks : Efficiency

Data Set	Size of the local $\mathcal{R}$-matrix	Size of the global final $\mathcal{R}$-Matrix	Number of scattering energies
Small	90x90	360x360	6
Medium	90x90	360x360	64
Large	383x383	7660x7660	6

Figure : Data sets

Small	Running Time	Speedup /Fortran	Speedup /OCaml
GPU Fortran	1m49s	100%	198%
GPU OCaml	3m36s	50%	100%

Medium	Running Time	Speedup /Fortran	Speedup /OCaml
GPU Fortran	10m70s	100%	241%
GPU OCaml	26m58s	41%	100%

Figure : Performances

Benchmarks : Efficiency

Data Set	Size of the local $\mathcal{R}$-matrix	Size of the global final $\mathcal{R}$-Matrix	Number of scattering energies
Small	90x90	360x360	6
Medium	90x90	360x360	64
Large	383x383	7660x7660	6

Figure : Data sets

Large	Running time	Speedup / Fortran			Speedup / Ocaml
		CPU 1	CPU 4	GPU	
CPU 1 core	71m11s	100%	51%	22%	28%
CPU 4 core	36m18s	196%	100%	43%	55%
GPU OCaml	19m55s	357%	182%	80%	100%
GPU Fortran	15m51s	449%	229%	100%	126%

Figure : Performances

Benchmarks : Memory Footprint

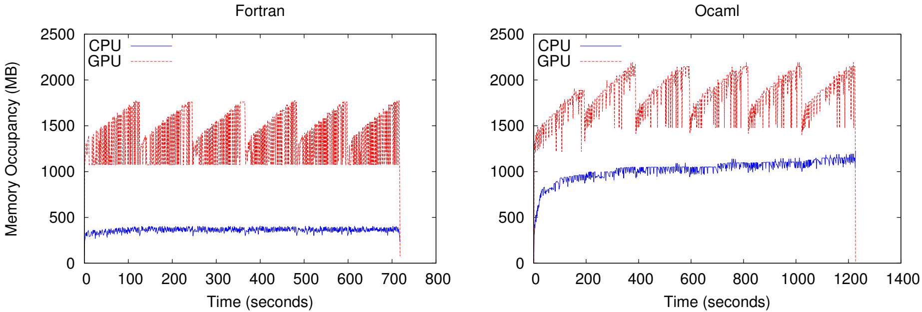


Figure : Memory occupancy (Large Case)

Conclusion

Advantages

- **No more transfers**
- Smaller, readable code
- Automatic memory management
- Type safety
- Good performance
- Small memory occupancy (on the GPGPU)

Limitations

- Not on par with hand-tuned Fortran performance
- Increase in CPU memory occupancy

Future Work

SPOC: Increase performance

- Allow usage of precompiled kernels
- Use of Weak (freeable) pointers for CPU space

`PROP`: Increase portability

- Complete code translation (I/O)
- Use SPOC to make `PROP` compatible with OpenCL
- Allow the use of hybrid clusters

Increase abstraction

- Skeleton library based on SPOC to ease complex algorithm implementation
- DSL akin to OCaml to describe GPGPU kernels

Thanks

SPOC

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