High-Performance GPGPU Programming with OCaml

Mathias Bourgoin, Emmanuel Chailloux, Jean-Luc Lamotte

Laboratoire d'Informatique de Paris - UMR 7606[1]
Université Pierre et Marie Curie - Sorbonnes Universités
4 place Jussieu, 75005 Paris, France
{Mathias.Bourgoin,Emmanuel.Chailloux,Jean-Luc.Lamotte}@lip6.fr

We present an OCaml GPGPU library with a DSL embedded into OCaml to express GPGPU kernels. The level of performance achieved is measured through different examples. We also discuss the use of GPGPU programming to increase the performance of multicore-CPUs software, written in OCaml.

GPGPU Programming

GPGPU (General Purpose Graphic Processing Unit) programming consists in using GPU devices to increase software performance. GPUs are manycore architectures combining a dedicated memory with several hundreds of computation units. GPGPU programming demands to describe a host program, running on the CPU, as well as computation kernels, running on the GPU. It is commonly handled through the use of the Cuda[4] or OpenCL[6] frameworks. Both are using the Stream Processing paradigm that is based on SIMD (Single Instruction Multiple Data) and simplifies the parallelism by describing parallel computations as the application of a simple series of operations over every element of a data stream, producing a new stream as a result.

GPGPU Programming with OCaml

To allow GPGPU programming with OCaml, we propose a library, SPOC (Stream Processing with OCaml) that unifies both OpenCL and Cuda frameworks and offers automatic data transfers between CPU and GPU memory. To express kernels, we offer a DSL embedded into OCaml: Sarek (Stream ARchitecture Extensible Kernel). We also provide interoperability between SPOC and current solutions to allow the use of existing high performance libraries.

SPOC. SPOC[2] allows GPGPU programming with OCaml. It is based on the Cuda and OpenCL frameworks, abstracting some of the low-level, verbose and error-prone boilerplate.

SPOC automatically detects, at runtime, every device compatible with Cuda or OpenCL on the system. It offers a unified API to handle them that allows to use any kind of device regardless of the framework they depend on. It also eases the development of multi-GPGPU software using different devices conjointly.

The memory bandwidth of the bus linking GPGPU devices and the CPU host is relatively small (10-20× that of the GPGPU itself). Thus, transfers can introduce bottlenecks in the whole program and demand to be optimized to achieve high performance. To ease GPGPU programming, SPOC manages transfers automatically. It introduces a vector data-set that is very similar to OCaml bigarrays that are monomorphic and can contain different kinds of integers, floats or booleans. Besides, SPOC always knows vectors location (on CPU or GPGPU memory) and is able to transfer them when needed. In particular, SPOC checks that every vector used by a GPGPU kernel is present in GPGPU memory (and triggers transfers if needed) before launching the computation. Similarly, when the CPU reads or writes in a vector, SPOC checks its location and transfers it if needed. Furthermore, SPOC uses the OCaml garbage collector to manage vectors and trigger transfers to the CPU memory when the GPU memory is full.

Sarek. To express kernels we propose a domain specific language, embedded into OCaml: Sarek. We've built it as a CamlP4 OCaml extension. It offers an OCaml-like syntax with type inference and static type checking. Current solutions are very difficult to debug, especially with OpenCL which commonly compiles kernels at runtime. Offering static type-checking already improves GPGPU programming by enabling error detection at compile-time. Otherwise, Sarek is very similar to the C subsets that are offered in Cuda or OpenCL. It is an imperative language used to express monomorphic elementary operations that will automatically be computed in parallel by the multiple computation units of GPGPU devices.

To use current high-performance GPGPU libraries as well as to allow developers to hand-tune their GPGPU kernels, we also permit to declare external GPGPU kernels somewhat similarly to the way OCaml already handles C externals.

```
1  open Spoc
2  let vec_add = kern a b c n ->
3    let open Std in
4    let idx = global_thread_id in
5    if idx < n then
6      c.[<idx>] <- a.[<idx>] + b.[<idx>]
7
8  let dev = Devices.init ()
9  let n = 1_000_000
10 let v1 = Vector.create Vector.float64 n
11 let v2 = Vector.create Vector.float64 n
12 let v3 = Vector.create Vector.float64 n
13
14 let block = {blockX = 1024;
15               blockY = 1; blockZ = 1}
16 let grid={gridX=(n+1024-1)/1024;
17           gridY=1; gridZ=1}
18
19 let main () =
20   random_fill v1;
21   random_fill v2;
22   Kirc.gen vec_add;
23   Kirc.run vec_add  (v1, v2, v3, n)
24            (block,grid) dev.(0);
25   for i = 0 to Vector.length v3 - 1 do
26    Printf.printf "res[%d] = %f; " i v3.[<i>]
27   done;
```

Kernel Code (Sarek)

Host Code (SPOC)

We present a simple example using SPOC and Sarek. The GPGPU kernel, written with Sarek, mainly consists in an elemental addition (line 6). It will be computed by a large number of threads, the code line 3-5 is used to ensure that no thread will try to access indexes out of the vectors boundaries.

The host program initializes the SPOC system (line 8). `Devices.init` returns an array of devices usable with SPOC. We then declare some vectors that will be used in the program (lines 9-12). Lines 14-17 consists in the declaration of a grid of blocks of threads that describes the layout of the GPGPU computation units we intend to run our kernel on. This is a virtual layout that will be mapped to the hardware one when the kernel is launched. Here, the grid will be composed a thread per element in our vectors. Running a kernel on it will allow *n* threads to compute the kernel code,

[1]Work partially funded by the OpenGPU Project : `http://opengpu.net`

resulting in a full vector addition. The Sarek code is compiled to Cuda/OpenCL dynamically (line 22). The kernel is then launched on a device (line 23-24), using the block and grid layout as parameters, as well as every parameter needed by the kernel to run correctly. Here all vectors ($v1$, $v2$ and $v3$) are automatically transferred to the device memory before the actual computation. Then the result (computed in $v3$) is printed by the CPU (lines 25-27). The CPU waits for the GPU to end its computation then transfers back $v3$ to be able to read its new values. $v1$ and $v2$ stay on the GPGPU memory, waiting for future uses or for the OCaml garbage collector to free them.

Benchmarks. To check the performance achieved by our solution we propose simple benchmarks as well as the translation of a HPC software from FORTRAN to OCaml.

We tested SPOC with two common data-parallel examples, Mandelbrot and Matrix Multiplication. We present the results achieved with multiple GPU devices compared to those sequential OCaml computation on an Intel Core-i7 3770.

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

For each sample, the first line shows the time and speedups obtained using SPOC and external kernels. The second line presents the results with Sarek. These show that using SPOC offers very high speedups over OCaml for intensive data-parallel programs.

To make sure that SPOC and Sarek are efficient with real-world programs, we ported the PROP software of the 2DRMP[5] suite from FORTRAN to OCaml. 2DRMP simulates the scattering of electrons in H-like ions at intermediates energies. It is a HPC (High Performance Computing) software awarded with the HPC prize for Machine Utilization, by the UK Research Councils' HEC Strategy Committee in 2006 that ensures us to work with state of the art HPC software. It is written in FORTRAN and heavily uses Cublas and Magma libraries for GPGPU computations. We bound those libraries to OCaml using SPOC before we porting the several kernels used in PROP into Sarek. We did not translate the I/O code, keeping it in FORTRAN.

Running Device	Running Time	Speedup
FORTRAN CPU 1 core	4271.00s (71m11s)	1.00
FORTRAN CPU 4 core	2178.00s (36m18s)	1.96
FORTRAN GPU	951.00s (15m51s)	4.49
OCaml (external kernels)	1018.00s **(16m58s)**	4.20
OCaml (Sarek)	1195.00s **(19m55s)**	3.57

Our translation is compared to different FORTRAN version of PROP. Using Sarek, our solution achieves 80% of the hand-tuned FORTRAN performance, while using external kernels, it grows to 93%. Our two versions offer a dramatic reduction of code size (30% less) mainly due to automatic memory management with automatic transfers. These very good results show that SPOC and Sarek can be used for OCaml programmers to increase the performance of their intensive data-parallel computations but also to ease the development of real HPC programs, as the performance is on par with that of low-level hand-tuned solutions.

SPOC for Multicores

OCaml is currently allowing concurrency, but without parallelism. As we have seen, SPOC and Sarek are useful to benefit from the GPGPU high performance with OCaml. Moreover, current CPUs are usable with the OpenCL framework, easing the efficient programming of multicore software. Thus, we propose to use SPOC and Sarek to benefit from multicore CPUs with OCaml.

In order to verify that SPOC proves being a good solution for multicore CPU computations, we tested simple benchmarks, comparing SPOC with OC4MC[1] and ParMap[3]. OC4MC is an effort to provide an alternative runtime/GC to OCaml allowing parallel threads. It provides a thread library compatible with the standard one, offering full compatibility with current OCaml programs. ParMap is an OCaml library providing Map/Fold constructs overs arrays and lists that offer auto-forking of OCaml code, providing automatic parallelism. OC4MC is a low level solution, demanding the development of complex programs using posix threads while ParMap offers a high-level solution that mainly targets data-parallelism. We compared SPOC with OC4MC through the classic matrix multiplication, and with ParMap *via* a program raising naively every elements of a vector to the power of 100. These programs show the behavior of each solution for data-parallel computations, where they should perform the best.

	OCaml	ParMap	OC4MC	SPOC + Sarek
Power	11s14	3s30	-	<1s
Matmul	85s	-	28s	6.2s

Each solution provides high speedups over OCaml sequential computations, however, SPOC offers the best performance. This is mainly due to the use of GPGPU programming frameworks in SPOC's core that benefit from every extensions of current CPUs (including SIMD ones) with automatic vectorization.

Conclusion

We presented SPOC, an OCaml library to allow GPGPU programming. SPOC offers automatic management of GPGPU devices as well as automatic transfers between CPU and GPGPU memory. With SPOC we offer Sarek, a DSL, embedded into OCaml to express GPGPU kernels. Sarek features an OCaml-like syntax with static type-checking, improving GPGPU software development. We tested our solution with multiple examples showing that it offers very high performance and can be used as a solution to develop HPC software. Furthermore, we compared SPOC with current efforts to provide parallel computations to OCaml, showing that SPOC can also be used to improve multicore CPU performance with OCaml.

Currently, Sarek manages vectors of float or int values. To provide more expressivity, while giving more flexibility to developpers, custom (record and variant) type definition will be added to the language. We also plan to provide interoperability between SPOC and OpenGL to be able to express 3D shaders with Sarek.

References

[1] Mathias Bourgoin, Benjamin Canou, Emmanuel Chailloux, Adrien Jonquet, and Philippe Wang. OC4MC: Objective Caml for Multicore Architectures. In *Draft Proceedings of the 21st Symposium on Implementation and Application of Functional Languages.*, 2009.

[2] Mathias Bourgoin, Emmanuel Chailloux, and Jean Luc Lamotte. SPOC : GPGPU Programming through Stream Processing with OCaml. *Parallel Processing Letters*, 2012.

[3] Marco Danelutto and Roberto Di Cosmo. A "minimal disruption" skeleton experiment: seamless map & reduce embedding in OCaml. *Procedia Computer Science*, 2012.

[4] NVidia. Cuda C Programming guide, 2012.

[5] 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.

[6] Khronos OpenCL WG. OpenCL 1.2 specifications, 2012.