Int J Parallel Prog
DOI 10.1007/s10766-013-0261-x

# Efficient Abstractions for GPGPU Programming

**Mathias Bourgoin · Emmanuel Chailloux · Jean-Luc Lamotte**

Received: 2 March 2013 / Accepted: 27 July 2013


**Abstract** General purpose (GP)GPU programming demands to couple highly parallel computing units with classic CPUs to obtain a high performance. Heterogenous systems lead to complex designs combining multiple paradigms and programming languages to manage each hardware architecture. In this paper, we present tools to harness GPGPU programming through the high-level OCaml programming language. We describe the SPOC library that allows to handle GPGPU subprograms (kernels) and data transfers between devices. We then present how SPOC expresses GPGPU kernel: through interoperability with common low-level extensions (from Cuda and OpenCL frameworks) but also *via* an embedded DSL for OCaml. Using simple benchmarks as well as a real world HPC software, we show that SPOC can offer a high performance while efficiently easing development. To allow better abstractions over tasks and data, we introduce some parallel skeletons built upon SPOC as well as composition constructs over those skeletons.

**Keywords** GPGPU · DSL · OCaml · Parallel skeletons · Parallel abstractions

## 1 Introduction

GPGPU programming proposes to use highly efficient but specific architectures to compute general software. This last decade, it has become so popular that sixty-two

M. Bourgoin (✉) · E. Chailloux · J.-L. Lamotte
Laboratoire d'Informatique de Paris 6 (LIP6-UMR 7606),
Université Pierre et Marie Curie (UPMC-Paris 6), Sorbonne Universités,
4 place Jussieu, 75005 Paris, France
e-mail: mathias.bourgoin@lip6.fr

E. Chailloux
e-mail: Emmanuel.Chailloux@lip6.fr

J.-L. Lamotte
e-mail: Jean-Luc.Lamotte@lip6.fr

Published online: 09 August 2013

systems of the top500 supercomputers list[1] use it (including the first one). Today, every personal computer sold is GPGPU compatible. However, by consisting in the use of different architectures (CPU and Accelerators) with different memory spaces, GPGPU programming can quickly become very hard to harness. Besides, it generally implies the use of low-level APIs that demand explicit memory and devices management. GPGPU programming is mainly possible through the Cuda [14] and OpenCL [12] frameworks. Both provide C/C++ libraries with C extensions to express GPU computations. To help programmers by simplifying GPGPU software development while ensuring more safety, we propose to use high-level programming languages. We developed a library for the OCaml language (SPOC—*Stream Processing with OCaml*, proposed in [4]) binding both Cuda and OpenCL frameworks while offering automatic management of data transfers between CPU and GPUs. We also developed parallel skeletons, and composition constructs based on our library. Our main goal is to offer abstractions while keeping performance (even for high-performance computing (HPC) applications). This relies on abstracting GPGPU frameworks by unifying both OpenCL and Cuda APIs. We also aim at providing abstraction over C-like languages provided by both frameworks in order to offer more expressivity and connection with the host language. Thus, we developped a new domain specific language, Sarek (Stream ARchitecture using Extensible Kernels), to express GPGPU kernels, which represent the main contribution of this paper.

Section 2 presents the SPOC library for GPGPU programming with OCaml. Section 3 shows how to express and use GPGPU kernels with SPOC through external Cuda/OpenCL kernels and introduces the Sarek language embedded in OCaml. It also presents benchmarks and results using SPOC, including results from a real size HPC application. Section 4 discusses the use of parallel skeletons libraries for GPGPU programming and how we implemented one with SPOC. Finally, Sect. 5 presents related works before concluding on our contribution and presenting future work in Sect. 6.

## 2 GPGPU Programming with OCaml

To allow GPGPU programming with OCaml we first developed SPOC, an OCaml library. SPOC allows to use Cuda or OpenCL kernels with OCaml and allows to create specific data sets useable by these kernels. SPOC also manages transfers between CPU memory and devices memory automatically. This section briefly presents GPGU programming and the SPOC library.

### 2.1 GPGPU Programming

GPUs are highly parallel architectures. They commonly are dedicated hardware with their own memory, linked to their hosts through PCI-E link. To ease programming, current frameworks only offer to use the *Stream Processing* paradigm. Stream processing

[1] http://www.top500.org/list/2012/11/

is related to SIMD (single instruction, multiple data). It implies the definition of simple kernel programs computing over single elements that are then applied to streams of elements. Dedicating one computing unit to each computation allows to easily benefit from the highly parallel GPU architecure.

To generalize GPGPU architectures, both frameworks (virtually) represent GPGPU with

- computation units, which we will call threads,
- blocks of threads and
- a grid of blocks.

The grid is mapped to the (hardware) device's multi-processors, each multi-processor running one or several blocks. Besides, devices are considered to have a dedicated memory, implying the copy of data from the CPU memory. Device memory is, moreover, divided into several categories:

- global memory (accessible by all threads in the grid),
- shared memory (shared inside a block),
- local memory (local to a thread) and
- specific memory banks depending on hardware.

Both Cuda and OpenCL offer a C/C++ API useable from classic host programs associated with a C subset language to express GPGPU kernels. Programmers have to manually handle GPGPU devices (which we simply call devices). This means to explicitly schedule kernel launches as well as memory transfers to and from CPU memory to device's (through verbose APIs). As the PCI-E link offers low bandwidth compared to that of the GPGPU memory (from 3 × to 24×—cf. Table 1), it is mandatory to manage transfers efficiently in order to prevent bottlenecks and achieve high performance.

GPGPU programming (and the Stream Processing paradigm) implies many levels of parallelism and memory. It also demands to explicitly manage data transfers as well as task scheduling. High level programming languages (such as OCaml) with high-level libraries can help tackle this challenge by abstracting some of this hardware related properties while improving expressivity and productivity.

## 2.2 SPOC, OCaml Host Library

### *2.2.1 OCaml*

OCaml [11] is a high-level general purpose programming language developed by Inria. It is a multiparadigm language: functional, imperative and object-oriented. It is strongly and statically typed and provides type inference. It offers automatic memory management through an efficient incremental garbage collector (GC). The OCaml distribution offers two compilers. One that produces portable bytecode (OCaml virtual machines exists for many architectures) while the other produces efficient native code. OCaml is also fully interoperable with the C language. SPOC benefits directly from several of these aspects. As an OCaml library using the C libraries from both Cuda and OpencL frameworks, it obviously uses the OCaml interoperability with C.

**Table 1** Theoretical memory bandwidth

| | Bandwidth (GB/s) | Speedup/PCI-E 2.0 | Speedup/PCI-E 3.0 |
|---|---|---|---|
| PCI-E (one way) 2.0 | 8.0 | 1.0 | 0.5 |
| PCI-E (one way) 3.0 | 16.0 | 2.0 | 1.0 |
| **CPU** Intel i7-3970X | 51.2 | 6.4 | 3.2 |
| **CPU** AMD FX-8350 | 37.0 | 4.6 | 2.3 |
| **GPU** NVIDIA GTX-690 | 384.0 | 48.0 | 24.0 |
| **GPU** Radeon HD 7970 | 264.0 | 33.0 | 16.5 |

SPOC targets GPGPU programming, and thus high performance, making it necessary to use a programming language already efficient for sequential computations, which native OCaml compilers provide. In addition, with SPOC, we aim to ease GPGPU programming while improving software reliability and productivity. Static type-checking improves reliability by detecting many programming errors at compile time, while the memory manager guarantees data consistency (no memory leak or dangling pointers) and limits memory usage during execution. Finally, SPOC also benefits from the multiple paradigms of OCaml as it uses modular, functional and sequential programming but also uses object oriented programming to manage kernels.

#### *2.2.2 SPOC*

SPOC is a GPGPU programming library for OCaml. It was developed focusing mainly on offering abstractions over current GPGPU frameworks, while keeping performance and portability. It first unifies both frameworks in a single API. A single program written with SPOC can use Cuda or OpenCL devices indifferently. It also allows the use of multiple GPUs using the same framework or not. This offers easy hybrid architectures management.

To offer automatic transfers, SPOC introduces a specific data set : *Vectors*. Vectors are based on OCaml bigarrays. They contain information on their position: on the host or devices memory. This allows SPOC, on specific occasions (like read/write on a vector or the launch of a kernel) to check and move them if needed. When a kernel needs a vector, SPOC transfers it to the device memory. When a vector used on a device is reused by the CPU, SPOC waits for the kernel to end before transferring back the vector. If unused, vectors aren't transferred. Furthermore, vectors are managed by the OCaml memory manager. Its garbage collector can delete them, from host as well as devices memory.

## 3 GPGPU Kernels and Ocaml

### 3.1 Using SPOC with External Kernels

SPOC allows the use of external kernels written in Cuda C/C++ or OpenCL C99. It provides an OCaml syntax extension to declare such kernels in a way similar to

```
1  __kernel void vec_add(__global const float * a,
2                        __global const float * b,
3                        __global float * c, int vector_size){
4      int nIndex = get_global_id(0);
5      if (nIndex < vector_size)
6          c[nIndex] = a[nIndex] + b[nIndex];}
```

OpenCL Code

```
1  kernel vector_add :
2   Spoc.Vector.vfloat32 -> Spoc.Vector.vfloat32 -> Spoc.Vector.vfloat32 ->
3    int -> unit = "kernel_file" "vec_add"
```

OCaml Code

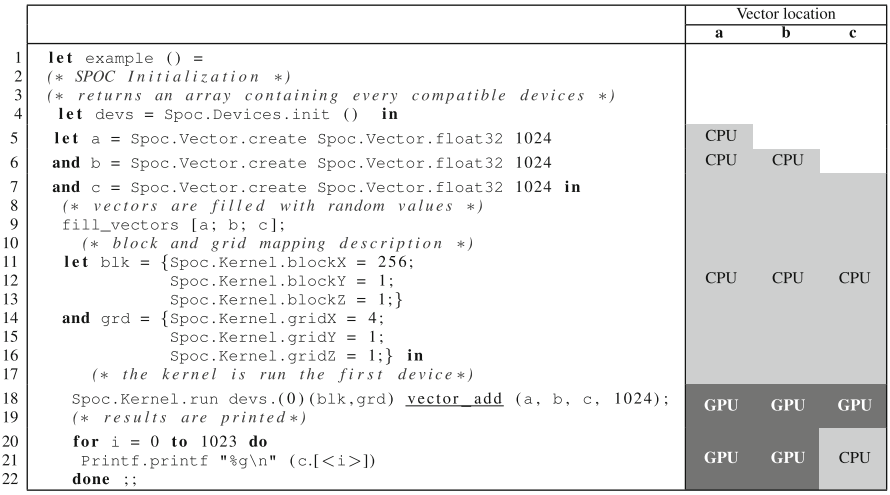
**Fig. 1** Kernel declaration with SPOC

OCaml existing external C functions declaration. This allows OCaml to type-check kernel parameters. Our syntax extension generates for each kernel an OCaml class, with a specific execution method and instantiates it. It also offers a compilation method able to load and compile the external kernel source code dynamically before being run. SPOC caches compiled kernels to avoid unnecessary re-compilation. Figure 1 presents an OpenCL kernel performing a vector addition with the OCaml code needed to use it. As presented earlier, it contains only the computation code for one element of the vector, using *get_global_id()* to get the index of the vector to compute. This figure also shows the necessary OCaml code to use such a kernel. It consists of the keyword *kernel* followed by its name as it will be used from the OCaml program, followed by its type (consisisting of the types of its parameters associated with its return type, which is always unit as GPGPU kernels always work through side effects). The two strings are the path to the source file containing the kernel and the name of the kernel function inside it. SPOC will load the source file and compile the correct kernel. Parameter types must be translated. In this example, *float* * vectors are translated to *Spoc.Vector.vfloat32*, to be compatible with SPOC and OCaml.

The OCaml code Fig. 2 presents a function using this kernel. It also shows the position of each vector at runtime. Vectors are automatically allocated on CPU memory at creation (line 5–7). Kernels are launched through the call of the *Kernel.run* (line 18) function that ensures that every vector needed for the computation has been transfered before actually running the kernel. In the same way, reading a vector triggers the check and potential transfers before performing the actual reading.

### 3.2 A DSL to Express Internal Kernels

SPOC can use external kernels (written in Cuda C/C++ or OpenCL C99). While this allows to use hand tuned kernels and existing code (including heavily optimized libraries), this also means using two different languages. Besides, as kernels are read and compiled dynamically, it's difficult for us to provide efficient kernels transformation or static optimizations implying the knowledge of both sides of the program (CPU OCaml code and GPGPU kernels). To work around this limitation, we extend OCaml with a domain specific language (DSL) to express GPGPU kernels.




```
1  let example () =
2  (* SPOC Initialization *)
3  (* returns an array containing every compatible devices *)
4   let devs = Spoc.Devices.init ()  in
5   let a = Spoc.Vector.create Spoc.Vector.float32 1024
6   and b = Spoc.Vector.create Spoc.Vector.float32 1024
7   and c = Spoc.Vector.create Spoc.Vector.float32 1024 in
8    (* vectors are filled with random values *)
9    fill_vectors [a; b; c];
10     (* block and grid mapping description *)
11   let blk = {Spoc.Kernel.blockX = 256;
12              Spoc.Kernel.blockY = 1;
13              Spoc.Kernel.blockZ = 1;}
14   and grd = {Spoc.Kernel.gridX = 4;
15              Spoc.Kernel.gridY = 1;
16              Spoc.Kernel.gridZ = 1;} in
17     (* the kernel is run the first device*)
18    Spoc.Kernel.run devs.(0)(blk,grd) vector_add (a, b, c, 1024);
19    (* results are printed*)
20    for i = 0 to 1023 do
21     Printf.printf "%g\n" (c.[<i>])
22    done ;;
```

**Fig. 2** Simple example with dynamic transfers

### *3.2.1 Language Description*

Using SPOC as a basis, we propose a language embedded into OCaml allowing to express GPGPU kernels (that we call Sarek). It's based on two elements: a syntax extension and an OCaml library. The extension (built using the Camlp4 OCaml preprocessor) allows kernel definitions, generating at compile-time a kernel internal representation AST (which we call "*Kir*"). The library transforms and compiles, at runtime, *Kir* to Cuda/OpenCL kernels.

*Sarek.* To allow the building of optimized kernels and to give programmers as many knowledge on the generated kernels as possible we decided to create a DSL very close to Cuda C and OpenCL C99. This allows programmers to efficiently predict their kernels performance and makes specific code tuning from the DSL possible. Furthermore, as errors are very difficult to detect in GPGPU kernels (when appearing at runtime), it also helps understanding them by avoiding heavy transformation of the code. Sarek is a statically and strongly typed imperative language. Figure 3 presents its grammar. A kernel is an OCaml expression whose subexpressions are included in the given grammar. It is not currently possible to define and use functions but it is possible to use predefined functions accessible through predefined modules (mainly GPGPU globals used to handle parallelism and synchronization, associated with mathematic functions, as well as binary operators for specific data types). Sarek allows side effects but has, contrary to Cuda/OpenCL frameworks, the possibility to return values (which will be stored in SPOC Vectors). Control flow consists on the alternative `if-then-else` associated with `while` and `for` loops. Sarek is strongly typed and binary operators are differentiated by the type of their arguments. GPGPU performance varies tremendously when using simple or double precision, so, Sarek allows (and demands) users to explicitly specify, through these operators, their data types. Sarek currently limits itself to floats and integers computations. It doesn't allow user defined types and can

| | | |
|---|---|---|
| Kernel | $K ::=$ **kern** $id* -> expr$ | kernel |
| Expressions | $expr ::= expr; expr$ | Sequence |
| | \| **let open** $Uid$ **in** $expr$ | Local Module |
| | \| **if** $cond$ **then** $expr$ [**else** $expr$]? | Alternative |
| | \| **for** $id = expr$ **to** $expr$ **do** $expr$ **done** | For Loop |
| | \| **while** $cond$ **do** $expr$ **done** | While Loop |
| | \| $id$ **.** [$< expr >$] | Vector Access |
| | \| $expr\ binop\ expr$ | Binary Operation |
| | \| $id$ **:=** $expr$ | Affectation |
| | \| $id$ **.** [$< expr >$] $< - expr$ | Vector Affectation |
| | \| $id\ expr\ \ldots\ expr$ | Function Call |
| | \| $id$ | Variable |
| | \| $c$ | Constants |
| Ident | $id ::= [a-zA-Z0-9\_]+$ | Variable name |
| | \| $Uid\ .\ id$ | Module Access |
| Module Ident | $Uid ::= [A-Z][a-zA-Z-0-9\_]*$ | Module name |
| | \| $Uid\ .\ Uid$ | Embedded Module |

**Fig. 3** Part of Sarek grammar

**Fig. 4** Sarek vector addition

```
1  let vec_add = kern a b c ->
2    let open Std in
3    let idx = global_thread_id in
4     c.[<idx>] <- a.[<idx>] + b.[<idx>]
```

only compute over single elements or SPOC vectors. Besides, Sarek being compiled at runtime can use OCaml globals that will be translated into their corresponding value during compilation.

Figure 4 presents the vec_add example (presented in OpenCL in Fig. 1) written with Sarek. The module gives access to globals such as `global_thread_id`. Here, the kernel works through side effects. The syntax is close to OCaml's. Types are inferred, allowing the generation of a typed AST needed to later generate correct Cuda or OpenCL code.

*Static Code Generation.* We propose a Camlp4 OCaml extension transforming at compile-time Sarek to *Kir*. Camlp4 is the OCaml preprocessor offered with the OCaml distribution. It offers tools for syntax and allows to modify the concrete syntax of the language, helping develop syntax extensions for OCaml. This generation is done in four steps (Fig. 5):

- type checking,
- generating OCaml Code,
- generating *Kir* and
- generating a spoc_kernel to receive our future kernel source code

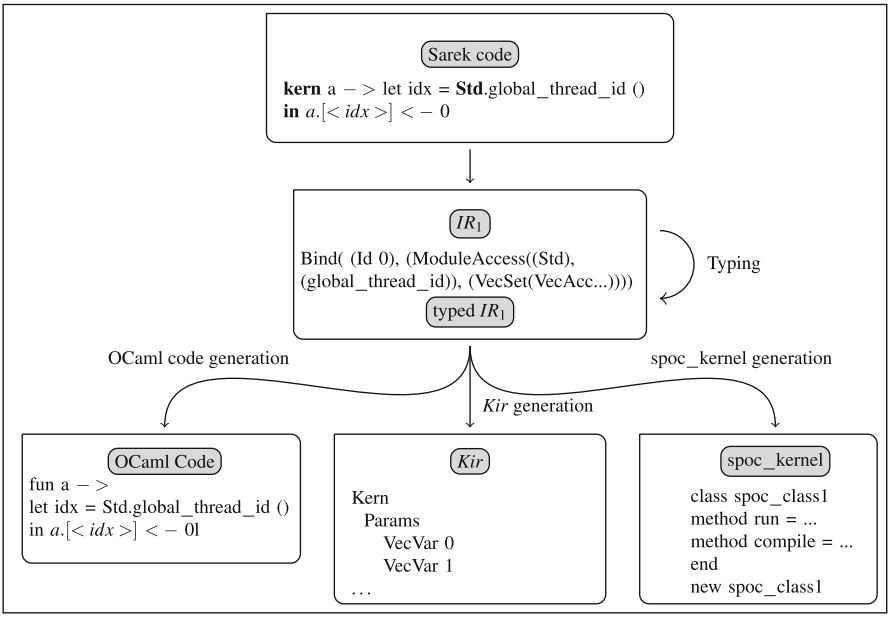



**Fig. 5** Kernel static compilation

As OCaml loses type information after compilation, we have to store it in our internal representation to generate correct Cuda/OpenCL code at runtime. OCaml preprocessing takes place before OCaml typing, ensuring that generated code will later be correctly type-checked by the compiler. Thus, we have to type-check Sarek code ourselves. Sarek being a rather simple language with currently no polymorphism, types are easily inferred. From this, we generate two separates functions, one corresponding to the kernel code, written in OCaml, as if running on the CPU. This allows us to benefit from OCaml type checker to recheck our programs while allowing us to build (in a future work) an OCaml GPGPU simulator that could use these functions as kernels, and that we could use as a default backend when using SPOC and Sarek on architectures incompatible with OpenCL/Cuda. We then generate the *Kir* code. Finally, this extension generates a spoc_kernel (in the same fashion as the previous SPOC extension) but does not associates this kernel with external source files. Association will be made, at runtime, with generated source coming from the *Kir* compiler.

*Dynamic Kernel ompilation.* At compile-time, the OCaml extension takes Sarek and generates *Kir* that is then compiled to Cuda/OpenCL at runtime. Figure 6 presents this mechanism. When the `gen` function is called, the *Kir* AST is parsed to generate Cuda and OpenCL source code. Despite its syntax, Sarek (as well as *Kir*) is very close to Cuda/OpenCL C, allowing direct translation. Currently, compilation does not optimize kernels, and generates an equivalent code for both Cuda and OpenCL targets. Figure 7 shows the OpenCL and Cuda code generated from the vec_add kernel in Fig. 4.

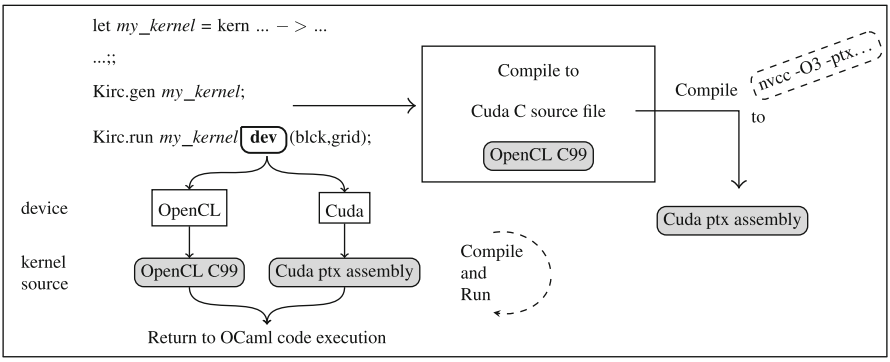



**Fig. 6** Dynamic Kernel compilation

*OpenCL*

```
__kernel void spoc_cl_0 ( __global int* spoc_var0,  __global int* spoc_var1,
                __global int* spoc_var2 ) {
  int spoc_var4;
  spoc_var4 = get_global_id (0) ;

  spoc_var2[spoc_var4] = spoc_var0[spoc_var4] + spoc_var1[spoc_var4];
}
```

*Cuda*

```
__global__ void spoc_cu_0 ( int* spoc_var0, int* spoc_var1,
                int* spoc_var2 ) {
  int spoc_var4;
  spoc_var4 = blockIdx.x * blockDim.x +threadIdx.x;
  spoc_var2[spoc_var4] = (spoc_var0[spoc_var4] + spoc_var1[spoc_var4]);
}
```

**Fig. 7** Cuda/OpenCL code generated from Sarek vector addition (Fig. 4)

As SPOC can only dynamically compile Cuda assembly, *Kirc* compiles its generated Cuda C source to assembly using the NVidia compiler. The OpenCL source code stays untouched as SPOC can use it directly. The spoc_kernel associated with our AST is then updated with the newly generated source code. When the kernel is used, *Kirc* only calls the execution method of the spoc_kernel. This will check the running device and compile the corresponding source code before running it and return to the OCaml program. When generating the kernel, *Kirc* will switch OCaml float/integers globals from the code with their corresponding values.

Sarek is currently a first attempt at embedding kernel expression into OCaml. It is limited to using global and local memory on devices. It could offer shared memory access as well as with higher level constructs to ease kernel development and offer stronger linking with the OCaml CPU code while providing higher performance. However, it already proves usefull to express any kind of kernels as shared memory can be simulated by global memory. Besides, it offers type-safety and static checking which helps a lot increasing productivity by detecting many errors at compile-time. Sarek also helps with portability, coupled with SPOC, it allows to write one single program, in OCaml, which can run on any GPGPU architecture.

**Table 2** Benchmarks using SPOC

| 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 |

## 3.3 Benchmarks

### *3.3.1 Small Examples*

This section presents performance benchmarks using SPOC for GPGPU:

- Mandelbrot generates a picture associated with a Mandelbrot set computation. Graphics are done by the CPU (using the OCaml Graphics module) while computation take place on the device.
- Matmult computes a (naive) matrix mutliply over two $2{,}000 \times 2{,}000$ matrices.

For each example we used multiple devices, a Nvidia Tesla C2070, a Geforce GTX 680, an AMD Radeon HD 6950 and an Intel CPUs (i7-3770). Cuda and OpenCL frameworks were provided through Cuda-5.0, AMD APP 2.8, Intel SDK (2012). Our Operating System is Ubuntu Linux 12.10 with Linux 3.7.0 kernel. Sequential results were measured on the i7-3770 processor using OCaml (on a single CPU core).

Table 2 presents both running times and speedups over sequential computations. For each benchmark, the first line presents the results obtained using external kernels (written in Cuda or OpenCL), and the second the line the results achieved with Sarek. It shows that using SPOC allows to efficiently use GPGPU programming as it offers high level of performance with high speedups over sequential OCaml computations. Furthermore, using internal kernels slightly reduces performance as it implies an additional compilation phase (which is currently not optimized) but still largely outperforms sequential computation. Moreover, the speedups obtained using OpenCL on CPU shows that using OpenCL with SPOC could prove efficient for CPU computations, especially for OCaml users, as Inria's distribution of OCaml can currently not benefit from multicore architectures.

### *3.3.2 Real World Use Case*

To check that our approach is scalable and can be used efficiently in HPC software, we've used SPOC to translate a known HPC software using GPGPU from Fortran and Cuda to OCaml and SPOC. The ported application (`PROP`) comes from the 2DRMP suite [15]. It obtained the HPC prize for Machine Utilization, awarded by the UK Research Councils HEC Strategy Committee in 2006, which ensures us to work on realistic, efficient and optimized code. It simulates the scattering of electrons in H-like atoms at intermediates energies. This software computes over large matrices which grow during runtime. This implies many reallocations on the GPU and means heavy usage of GPU memory. Being situated in the center of the 2DRMP

suite, PROP handles data produced by a previous program and generates new data for other ones. This implies a complex and large part of the code dedicated to I/O and data management.

The 2DRMP suite works on sequential architectures as well as on high performance clusters or supercomputers. 2DRMP has been optimized for shared-memory architectures but also for distributed-memory ones. PROP has been modified multiple times to benefit from GPGPU architectures.

PROP is written in Fortan and uses the BLAS and LAPACK libraries for the scattering computation of the input matrix. *CAPS-Entreprise* made a first modification by modifying the scattering equation to handle larger matrices. Indeed, for matrix multiplication, GPGPU performance grow with the matrices size. Using the HMPP compiler, CAPS developped a new version of PROP where matrix multiplications took place on the GPGPU (using Cuda). A second modification was made to limit transfers between CPU and GPGPU by moving every computations on the GPGPU. This work gave birth to a version of PROP performing all computations on the GPU[10] . This application heavily relies on the Nvidia Cublas[13] and the Magma[19] library, with few external Cuda kernels. It uses a C-glue to bind the Fortran code and the Cuda library.

To translate PROP from Fortran and Cuda to OCaml, we binded the necessary functions from Cublas and Magma before translating the computation part of the program, leaving I/O in Fortran. Library binding implied using SPOC vectors with external kernels and C funtions coming from the libraries. In order to handle small computations on matrices without transalting them back to the CPU, PROP uses small kernels written in Cuda. We developped two versions of the translation. One, using these kernels externally from SPOC, the other using Sarek to express those kernels, removing every Cuda code from the program.

Benchmarks (Table 3) show that our solution reaches **93** % of the performance of the hand tuned Fortran code while automatically handling transfers. This heavily reduces code size but also allows developers to focus on computations and kernels. This shows that SPOC can indeed be used to efficiently manage GPGPU architectures and software, even for very demanding HPC software. Using Sarek implies more dynamic compilation (which is still unoptimized) and achieves **80** % of the original performance. Howevers, this offers static typing of kernels and free portability through

**Table 3** Benchmarks comparing PROP running time using Fortran and SPOC

| Running device | Running time | Speedup/Fortran | | |
|---|---|---|---|---|
| | | CPU 1 | CPU 4 | GPU |
| Fortran CPU 1 core | 4,271.00 s (71 m11 s) | 1.00 | 0.51 | 0.22 |
| Fortran CPU 4 core | 2,178.00 s (36 m18 s) | 1.96 | 1.00 | 0.44 |
| Fortran GPU | 951.00 s (15 m51 s) | 4.49 | 2.29 | 1.00 |
| *OCaml GPU* | *1,018.00* s (*16* m*58* s) | *4.20* | *2.14* | *0.93* |
| *OCaml (+ Sarek) GPU* | *1195.00* s (*19* m*55* s) | *3.57* | *1.82* | *0.80* |

Results achieved using OCaml with our solution are italicized

OpenCL (as soon as OpenCL replacements for Cublas and Magma will be bound to the program).

## 4 Kernel Composition

While SPOC allows to use GPGPU programming with OCaml, it is still necessary to manually schedule GPGPU tasks to build complex programs. To ease the expression of complex algorithms it is common to use algorithmic skeletons. In this section, we will present how we build parallel algorithmic skeletons based on SPOC and Sarek and how to compose them.

### 4.1 Parallel Skeletons

Parallel algorithmic skeletons are programming constructs based on common patterns hiding most of the parallel programs complexity. Using some parameters they automatically manage synchronization or task scheduling for the programmer. Using high-level features of OCaml, we propose to use SPOC to build GPGPU skeletons simplifying software designs and offering automatic optimization of those skeletons [5].

In the following section we define two well known skeletons `Map` and `Reduce`. `Map` takes a kernel and a vector as parameters, and returns a vector. Each element of the returned vector is the kernel applied to the corresponding element of the input vector. `Reduce` also takes a kernel and a vector as parameters. `Reduce` returns a vector containing only one value. This value is computed by recursively combining elements of the input vector using the kernel to do the combination. True reduction needs a lot of synchronization between threads, and cannot fully benefit from all the computation units of GPGPU devices.

#### *4.1.1 Using External Kernels*

As explained in Sect. 3.1, SPOC allows to use external kernels. They take parameters and modify them through side effects. To allow composition and achieve automatic management of kernels and data, inputs and outputs are mandatory. We introduced a data structure associating:

- an external kernel
- an execution environment (consisting of the kernel parameters)
- an input (included in the execution environment)
- an output (included in the execution environment)

Inputs and outputs have to be SPOC vectors.

To run those skeletons we provide two functions:

- *run* which runs the skeleton on one device
- *par_run* which tries to divide computation by running the skeleton on a list of target devices

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

**Fig. 8** Type of map transformation

Using external skeletons limits our possibilities as they cannot be modified (OCaml sees the source code as a simple string). However, skeletons still help developers by automatically handling blocks and grid mappings that are hard to define if the running architecture is unknown or very heterogenous.

#### *4.1.2 Using Internal Kernels*

As we presented Sect. 4, using SPOC kernels to build skeletons and compositions was made difficult by many aspects of spoc_kernels like the fact they only work through side effects and cannot easily be modified (as they basically come from external source code). Using Sarek, we can solve those problems. Indeed, compilation taking place at runtime, we can provide functions transforming *Kir* to map a Sarek kernel to a skeleton.

To provide map and reduce skeletons using Sarek, we built two functions transforming *Kir* before compilation. For instance, map transforms a code working on a single element into one computing over a vector. It simply replaces every access to the scalar parameter to an access to a vector field. The kernel stays mostly unchanged, using local (thread) memory to compute the new scalar element before copying it in a field of a new vector. Figure 8 shows the type of the map transformation function. It takes a kernel generated *via* Sarek : a spoc_kernel (which corresponds to an object instantiated with the class generated by spoc extension) and a kirc kernel (associating the OCaml generated code with the *Kir* AST)). It returns a new couple. Scalar computations $('a -> 'b)$ are transformed into vector ones $(('a,'d)vector -> ('b,'c)vector)$. Vector types $(('a,'b)$ and $('b,'c))$ are defined by coupling the OCaml type of its elements with the abstract type corresponding to their C type. The spoc_kernel generated is compatible with map construct using external kernels.

### 4.2 Example

Figure 9 presents three versions (which will be detailed later) of a more complex example than in Fig. 2: the iteration loop of a power iteration algorithm (computing the largest eigenvalue of a given matrix). This algorithm is described by the iteration

$$b_{k+1} = \frac{A \times b_k}{||A \times b_k||}$$

At every iteration, the vector $b_k$ is multiplied by the matrix $A$ and normalized. These examples use three very simple kernels :

- *kern_init* which computes the matrix-vector multiply
- *kern_divide* which simply divides one data set by another
- *kern_norm* which computes the norm of an input vector

**(a)**

```
(**** OCaml program using the SPOC library ****)
open Kernel
...
let block = {blockX = 256; blockY = 1; blockZ = 1;}
and grid = {gridX = (Vector.length vn)/256; gridY = 1; gridZ = 1;}
while (!norm > eps && !iter < max_iter) do
  incr iter;
  maximum.[<0>] <- 0.;
  run dev (block,grid) kern_init (vn, v, a, n);
    (* Tools.fold_left max 0. v recursively computes the *)
    (* maximum of the vector v  (computes on CPU) *)
  maximum.[<0>] <- Tools.fold_left max 0. vn;
  run dev (block, grid) kern_divide (vn, maximum, n);
  run dev (block, grid) kern_norm (vn, v, v_norm, n);
  norm := Tools.fold_left max 0. v_norm;
done;
```

**(b)**

```
(**** Modifications to use skeletons ****)
open Compose
...
while (!norm > eps && !iter < max_iter) do
  incr iter;
  maximum.[<0>] <- 0.;
  vn := (run (map kern_init vn (vn, v, a, n )) dev vn);
  maximum := (run (reduce spoc_max maximum (vn, maximum,n)) dev vn);
  vn := (run (map kern_divide vn (vn, maximum, n)) dev vn);
  v_norm := (run (map kern_norm v_norm (vn, v, v_norm, n)) dev vn);
  max2 := (run (reduce spoc_max max2 (v_norm, max2, n)) dev v_norm );
  norm := max2.[<0>];
done;
```

**(c)**

```
(**** Modifications to use skeleton composition ****)
open Compose
...
while (!norme > eps && !iter < max_iter) do
 incr iter;
 max := (run (pipe
  (map k_init vn (vn, v, a, n ))
   (reduce spoc_max max (vn, max,n))) dev vn);
 norm := (run (pipe
  (pipe
     (map k_divide vn (vn, max, n))
     (map k_norm v_norm (vn, v, v_norm, n)))
      (reduce spoc_max max2 (v_norm!, max2, n))) dev vn ).[<0>];
done;
```

**Fig. 9** *Power iteration* examples. **a** Power iteration. **b** Power iteration with skeletons. **c** Power iteration with skeleton compositons

Using `Map` and `Reduce` we rewrote this example (Fig. 9b). Each kernel run is translated to a `Map` skeleton while each maximum computation is translated to a `Reduce` skeleton. Both skeletons automatically select blocks and grid size according to the input vector size and to the running device features. Furthermore, while we

previously used the CPU to compute the maximum of a vector, we now use the `Reduce` skeleton that computes on the GPU. Our implementation currently uses only one thread to compute on the device in order to avoid synchronization. This provides a low kernel performance but can improve overall efficiency as it also avoids unnecessary transfers by keeping data on the device.

### 4.3 Skeleton Composition

Having introduced common skeletons, it is now possible to provide composition constructs over skeletons. We propose two skeleton compositions : `Pipe` and `Par`.

- `Pipe` takes two skeletons and returns a skeleton computing both skeletons sequentially using the first sekeleton output as the input of the second one.
- `Par` takes two skeletons and returns a skeleton that will run both in parallel if the hardware allows it.

In our example, many computations use previous results as input. It is thus possible to use the `Pipe` composition (Fig. 9c). Piping skeletons allows SPOC to start transferring data needed by the second skeleton while computing the first one, overlapping transfers with computations. Besides, it also reduces the number of times SPOC will compute the checks and transfers optimizations needed for each run.

### 4.4 Benchmarks

Table 4 presents the running time obtained using external kernels for the three different versions, for 10,000 iterations of the power iteration algorithm. (a) shows the basic algoritmh while (b) uses map and reduce skeletons and (c) uses the pipe composition. Results show that composing can help developers (as describe in the previous section) while it also allows us to provide more automatic optimizations. Using GPGPU kernels increases performance over sequential OCaml computations. Using skeletons increases performance, mostly because the reduction takes place on the GPU, avoiding transfers. However, while it slightly improves performance, we do not heavily benefit from the pipe composition as the overlapping of transfers by computation is here limited to the transfer of a single value. Furthermore, the reduction skeleton, although still sequential (while computed on the GPU), improves only slightly performance (mainly because it avoids unnecessary transfers).

## 5 Related Works

Many high-level languages now provide bindings GPGPU frameworks. Some goes further by offering higher abstractions over those frameworks. Most are compatible with only Cuda or OpenCL and some are still in development. So, it was difficult to compare their performance with ours. In this section, we present some of these approaches and their specificities.

*Directives.* The HMPP [8] workbench provides directives for C and Fortran to declare and call codelets that can run on accelerators. This method allows developers to keep

**Table 4** *Power iteration* results

(a) Results with SPOC

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

(b) Results with skeletons

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

(c) Results with skeleton composition

| Device | Framework | Time (s) | Speedups | | |
|---|---|---|---|---|---|
| | | | Ocaml | SPOC (a) | Skeletons (b) |
| 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 |

their source compatible with CPU-only systems while the HMPP compiler will generate specific code for GPGPU systems. OpenACC [7] is a standard providing directives highly compatible with HMPP directives.

*Skeletons.* SkePU [9] and SkelCL [16] are both C++ libraries automatically handling vectors and transfers similarly as SPOC (with lazy transfers). Both provide data-parallel skeletons that can compute over those vectors. They benefit from the compatibility between C++ and Cuda/OpenCL and avoid complex translations between different languages. StarPU [2] offers a C++ runtime to manage and schedule tasks. It provides tools to generate parallel tasks over heterogeneous hardware while offering the possibility to hand-tune kernels if needed. Libraries like Magma and SkePU are usable with StarPU.

*DSL.* Obsidian [17](Haskell) or Copperhead [6](Python) are embedded languages to describe operations on arrays. They offer a set of combinators that are translated to Cuda and run on the device. Obisidian offers numerous constructions including filters modifying vector sizes. This kind of combinators are very difficult to efficiently parallelize on GPGPU, and so limit performance gains. ScalaCL [3](Scala) and Accelerator [18](.Net) provides specific data sets with automatic translation of specific operations to GPGPU kernels. Aparapi [1] is a Java library converting Java bytecode to OpenCL. It provides a Kernel class whose method *run* must be overridden to express the kernel to run.

## 6 Conclusion

### 6.1 Contribution

In this paper, we described SPOC a library to manage GPGPU kernels and memory with OCaml. This library offers automatic transfers between host and devices. SPOC offers to use external kernels and existing libraries with OCaml. We also introduced Sarek to express kernels from the OCaml program. While currently very simple, it already makes expressing various programs possible and offers similar performance as with external kernels. Both our small examples as well as a real size HPC software, show that it already offers a high level of performance. Finally we presented parallel skeletons using SPOC external and internal (Sarek) kernels. Using skeletons helps simplifying GPGPU programming by enabling automatic management of many low-level hardware-related optimizations. We also provided composition over those skeletons and showed through an example how they allow to explicitly describe the relations between kernels and vectors associated with them. This allows higher optimizations based on these relations.

### 6.2 Future Work

To provide higher level abstractions and offer simple tools for GPGPU programming, we intend to improve both Sarek and our skeleton library. Currently, Sarek can only manage vectors and float/int values. To provide more expressivity, while giving more flexibility to developpers, custom (record and variant) type definition will be added to the language. As presented, Sarek can use functions coming from predefined modules. We intend to provide a syntax extension to express those external functions and modules.

Currently, Sarek does not help ensuring safety (besides being strongly and statically typed) and we also intend to offer generation of safe kernel, producing (less efficient) kernels with memory bound checking and exception handling. With GPGPU frameworks, kernels are seen as a whole and only few errors are reported to the CPU. Moreover, these errors contain very little information. For example, errors produced by a thread on a device can not be traced back to their origin. Offering safer kernels with precise exception handling could thus greatly improve debugging.

Sarek is a playground for code transformation and generation. We plan to use it to provide more skeletons and compositions. Currently, compositions do not heavily transform kernels. We intend to provide mechanism such as kernels merging/splitting to easily express relations and synchronizations between kernels.

**Acknowledgments** The authors thank Stan Scott and the "High Performance and Distributed Computing" department of Queen's University of Belfast (United Kingdom) for the 2DRMP library and the program `PROP` as well as Rachid Habel from Telecom SudParis for sharing his knowledge on `PROP`. This work is part of the OpenGPU project and is partially funded by SYSTEMATIC PARIS REGION SYSTEMS & ICT CLUSTER (http://opengpu.net/).

## References


1. AMD: Aparapi. http://code.google.com/p/aparapi/
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput. Pract. Experience Special Issue: Euro-Par **2009**(23), 187–198 (2011)
3. Beck, R., Larsen, H., Jensen, T., Thomsen, B.: Extending scala with general purpose GPU programming. Technical report, Adlborg University, Departement of Computer Science (2011)
4. Bourgoin, M., Chailloux, E., Lamotte, J.L.: SPOC : GPGPU programming through stream processing with OCaml. Parallel Process. Lett. **22**(2), 1–12 (2012)
5. Bourgoin, M., Chailloux, E., Lamotte, J.L.: High level GPGPU programming with parallel skeletons. In: Patterns for Parallel Programming on GPUs. Saxe-Coburg Publications (to appear). ISBN 978-1-874672-57-9
6. Catanzaro, B., Garland, M., Keutzer, K.: Copperhead: compiling an embedded data parallel language. SIGPLAN Notices **46**(8), 47 (2011)
7. Cray Inc. CAPS Enterprise, N., Group, T.P.: OpenACC 1.0 specification (2011)
8. Dolbeau, R., Bihan, S., Bodin, F.: HMPP: a hybrid multi-core parallel programming environment. In First Workshop on General Purpose Processing on Graphics Processing Units (2007)
9. Enmyren, J., Kessler, C.W.: SkePU: a multi-backend skeleton programming library for multi-GPU systems. In: Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications, HLPP '10, pp. 5–14. ACM (2010)
10. Fortin, P., Habel, R., Jezequel, F., Lamotte, J., Scott, N.: Deployment on gpus of an application in computational atomic physics. In: Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), pp. 1359–1366. IEEE (2011)
11. Leroy, X., Doligez, D., Firsch, A., Garrigue, J., Remy, D.R., Vouillon, J.: The OCaml system release 4.00: documentation and user's manual. Technical report, Inria (2012). http://caml.inria.fr
12. Munshi, A., et al.: The OpenCL Specification (2012). http://www.khronos.org/opencl
13. Nvidia, C.: Cublas library (2012). http://developer.nvidia.com/cublas
14. Nvidia, C.: Cuda C Programming Guide (2012). http://docs.nvidia.com/cuda/index.html
15. Scott, N., Scott, M., Burke, P., Stitt, T., Faro-Maza, V., Denis, C., Maniopoulou, A.: 2DRMP: a suite of two-dimensional R-matrix propagation codes. Comput. Phys. Commun. **180**(12), 2424–2449 (2009)
16. Steuwer, M., Kegel, P., Gorlatch, S.: SkelCL-a portable skeleton library for high-level GPU programming. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), pp. 1176–1182. IEEE (2011)
17. Svensson, J.: Obsidian: GPU Kernel programming in Haskell. Technical report 77L, Computer Science and Enginering, Chalmers University of Technology and Gothenburg University (2011)
18. Tarditi, D., Puri, S., Oglesby, J.: Accelerator: using data parallelism to program GPUs for general-purpose uses. ACM SIGARCH Comput. Archit. News **34**(5), 325–335 (2006)
19. Tomov, S., Nath, R., Du, P., Dongarra, J.: Magma users guide. ICL, UTK (2009)