GPGPU Composition with OCaml

Mathias Bourgoin
Sorbonne Universités, UPMC Univ Paris 06,
UMR 7606, LIP6,
4 Place Jussieu, F-75005 Paris, France
Mathias.Bourgoin@lip6.fr

Emmanuel Chailloux
Sorbonne Universités, UPMC Univ Paris 06,
UMR 7606, LIP6,
4 Place Jussieu, F-75005 Paris, France
Emmanuel.Chailloux@lip6.fr

Abstract

GPGPU programming promises high performance. However, to achieve it, developers must overcome several challenges. The main ones are : write and use hyper-parallel kernels on GPU, manage memory transfers between CPU and GPU, and compose kernels, keeping individual performance of components while optimizing the global performance. In this article, we propose to study the composition by distinguishing the location where it is done : kernel composition on the GPU, kernel generation by the CPU, and overall composition. To achieve it, we use the SPOC library, developed in OCaml. SPOC offers abstractions over the Cuda and OpenCL frameworks. It proposes a specific language, called Sarek, to express kernels and different parallel skeletons to compose them. We show that by increasing the level of abstraction to handle kernels, programs are easier to write and that some optimizations (*via* kernel generation and transfers scheduling) become possible. Thus, we win on both sides : expressiveness and efficiency.

Categories and Subject Descriptors C.1.4 [*Computer System Organization*]: Processor Architectures - Parallel Architectures; D.1.3 [*Software*]: Programming Techniques – Concurrent Programming; D.3.4 [*Programming Languages*]: Processors – Code Generation, Runtime environment

General Terms Design, Languages, Performance

Keywords GPGPU, DSL, OCaml, parallel skeletons, composition

1. Introduction

GPGPU programming consists in the use of specific devices, GPUs, that are dedicated to graphics rendering, for general purpose computations. Thus, it demands to compose software using two kinds of subprograms : *(i)* a host program that will run on the CPU and manage memory transfers and computations, and *(ii)* GPGPU kernels that are small programs describing intensive computations that will run on the GPU.

Both programs demand to be composed internally as well as globally. In order to improve the composition of GPGPU programs, we propose to use a high-level multi-paradigm language : OCaml.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ARRAY'14, June 12-13th 2014, Edinburgh, United Kingdom.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ACM 978-1-4503-2937-8/14/06...$15.00.
http://dx.doi.org/10.1145/2627373.2627379

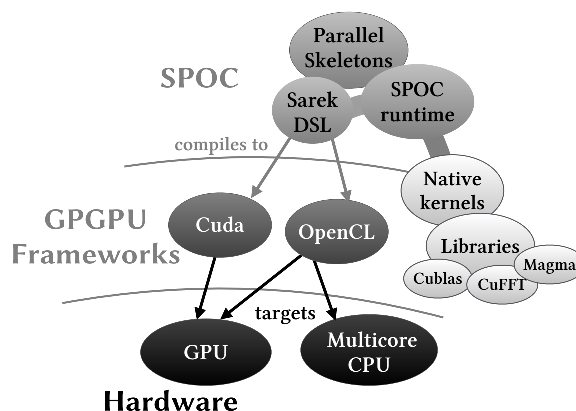

Figure 1: GPGPU Composition with OCaml

Using a dedicated library, SPOC, it is possible to use several compositions schemes to handle both host and kernel programs.

This article is constructed through a bottom-up direction. As presented in figure 1, we will first introduce GPGPU programming by describing how hardware is handled by common frameworks (Cuda [1] and OpenCL [2]). Then, we will present the SPOC library. First, in section 2.1, we will discuss our choice of OCaml as a composition language, and how its high-level features improve composition while keeping programs efficient. Then, we will describe in section 2.2 the SPOC runtime. It abstracts memory transfers and manages GPGPU kernels, through interoperability with external Cuda/OpenCL kernels and bindings with high performance libraries, as well as through a Domain Specific Language (DSL) embedded into OCaml, Sarek. SPOC and Sarek are both targeting high performance while offering abstractions and portability. Furthermore, both are built taking into account the evolution of computers and supercomputers into very heterogeneous architectures and provide seamless use of these systems. We will discuss, in section 3, how both can be used to compose programs sequentially, as its is commonly done in High Performance Computing (HPC) software, and will present an example of HPC software built with them. In order to improve composability and increase efficiency, we will describe, in section 4, how to build higher-order constructions upon SPOC. Specifically, we will present how parallel skeletons can be translated to composition of GPGPU kernels with automatic optimizations. As Sarek is embedded into OCaml, we will discuss how OCaml functions can transform GPGPU kernels to serve specific purposes, and how these transformations offer

automatic generation of complex composition algorithms, mixing GPGPU kernels with host composition. Finally, we will compare our approach with several related works in section 5, before concluding and presenting our future work.

2. GPGPU Programming with OCaml

Commonly, GPUs and CPUs are dedicated devices and possess their own memory. To produce efficient code, programmers have to optimize both their GPGPU kernels in order to benefit from the highly parallel architectures, as well as the host program to take into account the scheduling of transfers and kernel launches by the host program. Currently, the GPGPU programming model (and frameworks) can target multiple kinds of hardware architectures, from GPUs to multi-core CPUs and specific accelerators like FPGA and the Cell processor.

To write GPGPU software, two main frameworks exist : Cuda and OpenCL. Cuda is developed by Nvidia. It is proprietary and can only be used directly (without any kind of translator) with Nvidia devices. OpenCL is a standard offered by the Khronos Group. Many hardware vendors (including Nvidia, AMD or Intel) offer an OpenCL implementation for their products (multi-core CPUs as well as GPUs). Both frameworks are based on the stream processing that considers kernels as computations taking a stream as an input and producing another stream as an output. Cuda and OpenCL are built around C libraries used for the host part, and C subsets to express kernels. While being very similar, both are incompatible and difficult to compose together. Computers are now heterogeneous architectures that can easily combine a multi-core CPU (that can be targeted with OpenCL), that embeds a GPU (OpenCL) and often provides a dedicated GPU (compatible with Cuda or OpenCL) for intensive graphics rendering. Portability and heterogeneity of systems becomes an issue for programmers. To compose efficient software with low-level and verbose frameworks is very difficult and error-prone. Thus, we propose to use a high-level programming language to do the composition.

2.1 OCaml as a composition language

An important part of GPGPU programming comes from the composition of transfers and computations. OCaml [3] is a language developed at Inria. It is a multi-paradigm (functional, imperative, object, modular) language that features a state of the art memory manager. It can be compiled to efficient native code for performance as well as to virtual machine byte-code for portability. These features makes it an excellent candidate for GPGPU composition. In particular, it offers sequential programming, that is the common practice for HPC well as the common paradigm for GPGPU kernels and libraries. Besides, its other paradigms can help improve expressiveness, through the development of higher order algorithmic constructions to handle specific GPGPU algorithms.

Using OCaml, we developed the SPOC library that offers to use GPGPU programming *via* Cuda and OpenCL and provides several abstractions over memory transfers as well as devices management.

2.2 SPOC : Stream Processing with OCaml

SPOC [4, 5], is built upon Cuda and OpenCL. It has been developed focusing on performance but also portability and heterogeneity. Besides, as transfers scheduling and composition are difficult but mandatory tasks to achieve high performance, it provides automatic transfers as well as several ways of expressing GPGPU kernels, through interoperability or *via* the use of an OCaml DSL, Sarek.

Portable and Heterogeneous. To provide portability, SPOC unifies both Cuda and OpenCL APIs. SPOC is dynamically linked to those APIs, thus it is possible to compile or use a software built with SPOC on systems with any kind of GPU devices (compatible with Cuda or OpenCL). Besides, SPOC automatically detects every devices compatible with it at runtime. Associated with a common API, this can be used to handle multiple GPUs (from any framework) indifferently and conjointly. This eases the expression of complex programs dedicated to very heterogeneous architectures, such as supercomputers or current personal computers.

Vectors and lazy transfers. In order to abstract transfers scheduling, SPOC introduces a specific monomorphic data set to OCaml: vectors. SPOC keeps tracks of vectors location (on CPU or GPGPU memory) during program execution, and thus, can automatically 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 required) 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.

3. Kernel Composition

As for transfers, it is mandatory to compose kernel internally and globally. Thanks to OCaml, kernels can be composed sequentially, and new abstractions can be built to compose them.

3.1 Expressing kernels

SPOC provides two main solutions to express GPGPU kernels. The first one is to use interoperability with Cuda/OpenCL kernels. This eases code reuse as well as helps write bindings with existing high performance libraries such as Cublas [6] or Magma [7]. The second one is to use Sarek [4], a DSL built into OCaml dedicated to GPGPU kernels. Sarek is based on the C subsets of Cuda and OpenCL but offers type inference, with static type checking as well as an OCaml-like syntax for more consistency with the host program. Sarek kernels are written within the OCaml program (in the same file as the host program), and can even be embedded into OCaml functions. They are compiled in two steps. The first takes place at compile-time. It uses a Camlp4 [8] Ocaml syntax extension to type-check the kernel and generate OCaml code. This code embeds the kernel internal representation (KIR) as well as the representation of an external kernel, in order to make Sarek kernels compatible with external ones. SPOC handles both indifferently, linking external kernels to their source file and internal kernel to their KIR. The second step takes place at runtime and compiles KIR into actual Cuda or OpenCL kernels depending on the device where the kernel is loaded. This provides a way to write full GPGPU program from OCaml while providing portability and heterogeneity.

The figure 2 shows a simple kernel computing a vector addition written in Sarek, Cuda and OpenCL. Kernels are elementary operations that are mapped to the GPU computations unit to provide global computations. Here, multiple computation units will compute an elementary addition, producing the overall vector addition. Programmers can express a virtual 3D layout of their mapping, mimicking a 3D grid layout composed of blocks of elementary computation units. These layouts are dependent of the kernel and must be optimized depending on the computation units layout on the hardware used.

3.2 Sequential and functional composition

Why use lazy transfers? Using OCaml as a kernel composition language, it becomes possible to express composition with its multiple paradigms. However GPGPU kernels are imperative procedures that work through side effects making it difficult to predict which input value will be modified and thus provide efficient functional kernel composition. This leads to imperative designs, composing kernels sequentially. In order to make sure that every vector

Sarek

```
let vec_add = kern a b c n ->
 let i = thread_idx_x +
          block_idx_x * block_dim_x  in
   if i < n then  c.[<i>] <- a.[<i>] + b.[<i>]
```

Cuda

```
__global__ void vec_add(const float* a,
                        const float* b,
                        float* c,
                        int N) {
     int i = blockDim.x * blockIdx.x + threadIdx.x;
     if (i >= N) return ;
     c[i] = a[i] + b[i];}
```

OpenCL

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

Figure 2: Vector addition in Sarek, Cuda and OpenCL

needed by a kernel is present on the device running the kernel, it is mandatory to transfer all vectors on the device. As mentioned earlier, kernels are procedures and in order to make sure that every vector potentially modified by the kernel can be accessed by the CPU, a common automatic solution [9] consists in transferring back every input vectors to the CPU memory after kernel execution. It ensures that modified vectors are correctly transferred. But, this solution may trigger many unnecessary transfers especially when applying several kernels to the same vectors, and thus reduces performance. SPOC provides on-demand transfers. This offers efficient sequential composition and ensures that unused vectors stay on their current location. Thus, it limits costly transfers. Most high performance libraries are built to be used sequentially. Using OCaml imperative paradigm is thus helpful to develop simple bindings to this libraries.

Case study. To check that our solutions, with OCaml to compose kernels and high performance library calls, are relevant, we ported a real size numerical application from Fortran and Cuda to OCaml. We chose the PROP software of the 2DRMP suite [10] that simulates the scattering of electrons in ions at intermediate energies. 2DRMP is a high performance software targeting multiple architectures, from sequential computers to GPGPU clusters and supercomputers. The suite has been awarded the UK Research Councils' HEC Strategy Committee HPC Prize (2006) for machine utilization. This ensured us to work with state of the art HPC software. PROP mainly uses HPC libraries (Cublas and Magma) for its GPGPU computations. We provided bindings for OCaml to a subset of each library. Several GPGPU kernels were also translated from Cuda to Sarek. PROP being a part of We only translated the computation part to OCaml, keeping I/O in Fortran [11]. Our program mixes OCaml, Sarek, Fortran, calls to HPC libraries and a bit of C code to glue OCaml and Fortran together. While looking rather complex, and keeping the overall layout of the program, that is sequential calls to Cublas and Magma functions, the final code is 30% shorter than the original one, and benefits from features of OCaml such as automatic memory management, and static type checking. Most of the code reduction comes from the removal of transfers scheduling from the host program, SPOC managing it automatically.

Table 1 presents the results we obtained, compared to the original version of the program. We compared two versions of our

PROP Version	Time	Speedup
Fortran	15m51s	1
SPOC + Sarek	19m55s	0.8
SPOC + Cuda kernels	16m58s	0.93

Test machine : 1 Intel Core i7-3770 CPU + 1 Nvidia Tesla C2070 GPU
data sets : local $\mathfrak{R}$-matrix : 383×383 – global $\mathfrak{R}$-matrix : 7660×7660

Table 1: PROP results

program, one using Sarek to describe GPGPU kernels, while the other uses untouched Cuda kernels through interoperability. Results show that using SPOC offers very good performance as we achieve 80% of the hand-tuned Fortran program performance when using Sarek and increases it to 93% when using external kernels.

4. Functional composition

As discussed previously, using GPGPU kernels directly as well as through simple bindings to HPC libraries makes it difficult to compose computations other than sequentially. This is due to kernels being procedures with no outputs. In order to improve composition of the overall program and increase reusability, we propose algorithmic constructs built upon SPOC and Sarek : transformation functions that heavily modify Sarek KIR in order to provide specific computations, and algorithmic skeletons that offer functional composition of kernels and helps provide automatic optimization

4.1 Automatic composition from transformation functions

Using SPOC, it is possible to declare external kernels or describe internal ones using Sarek. Sarek gives access to KIR from the host program. Thus, we can write functions, from our composition language, OCaml, transforming KIR to match specific computations. For instance, a *map* (figure 3) transformation can transform an otherwise not executable kernel into a correct one. From a kernel generated *via* Sarek, *map* transforms scalar computations (`'a -> 'b`) that cannot be run on a GPU into vector ones (`'a vector -> 'b vector`). Taking a vector as parameter it automatically computes the host composition to produce the correct output vector.

```
val map : ('a -> 'b) kernel ->
          'a vector -> 'b vector
```

Figure 3: Type of *map* transformation

This kind of constructs can be used to rewrite the vector addition mentioned previously as simply as :

```
let vec_add a b = map2 (kern x y -> x + y) a b
val map2 : ('a -> 'b -> 'c) kernel ->
           'a vector -> 'b vector ->'c vector
```

To describe how the transformation is done, let's look at the sorting algorithm. Sorting vectors efficiently in parallel is difficult, especially with most GPUs that cannot call kernels from kernels or use recursion. Sorting combines GPGPU kernels with host composition to correctly schedule each step of the overall computation.

It is possible to write this kind of computations by hand with SPOC and Sarek. But using Sarek, it is possible to transform the elementary comparison to generate the needed code to compute a global sorting.

Figure 4 shows how *sort* transforms a scalar comparison kernel into the sequential host composition of a GPGPU kernel computing over vectors. Given the comparison function, *sort* injects it within a

```
sort (kern a b -> a - b) vec1
val sort : ('a -> 'a -> int) kernel ->
           'a vector -> unit
```

Injection into sort kernel

```
let bitonic_sort = kern v j k ->
 let open Std in
 let i = thread_idx_x + block_dim_x*block_idx_x in
 let ixj = Math.xor i j in
 let mutable temp = 0. in
 if ixj >= i then (
    if (Math.logical_and i k) = 0  then (
    if v.[< i >] - v.[< ixj >] > 0 then
     (temp := v.[<ixj>];
      v.[<ixj>] <- v.[<i>];
      v.[<i>] <- temp))
     else if v.[< i >] - v.[< ixj >] < 0 then
       (temp := v.[<ixj>];
        v.[<ixj>] <- v.[<i>];
        v.[<i>] <- temp);)
```

Host composition
↓

```
while !k <= size do
  j := !k lsr 1;
  while !j > 0 do
    run bitonic_sort (vec1,!j,!k) device;
    j := !j lsr 1;
  done;
  k := !k lsl 1 ;
done;
```

Figure 4: Overall composition generation *via sort* transformation

more complex predefined code. Here, we are using a simple Bitonic sorting algorithm [12].

Transformations makes using GPGPU easier by abstracting most of it : managing devices, transfers, kernels and the overall composition. Besides, transformations offer some polymorphism compared to kernel. The type of the final kernel depends on the transformation and its parameters.

4.2 Parallel skeletons

Skeletons [13] are algorithmic constructs based on common design patterns, that can be parameterized to adapt their behavior. They are commonly used in parallel programming to abstract parallel complexity while helping access high performance, in particular with Ocaml, for instance with the CamlP3L [14] library. By building skeletons, we can associate GPGPU kernels to inputs and outputs, making it possible to compose them. Thus, it becomes possible to compose GPGPU computations described with external Cuda and OpenCL kernels as well as with Sarek.

We introduce a skeleton data structure associating

- a 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. The `('a,'b,'c) skeleton` type is parameterized by `'a` the environment type, `'b` the input type and `'c`, the output type. The 'a type includes kernel globals, inputs and outputs and thus contains `'b` and `'c`.

SPOC offers two kinds of 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.

```
(* 'a : environment, 'b : input, 'c : output *)
val MAP : 'a kernel ->  'b vector -> 'c vector
     -> ('a,'b,'c) skeleton
val REDUCE : 'a kernel ->  'b vector -> 'c vector
     -> ('a,'b,'c) skeleton
val run : ('a,'b,'c) skeleton -> 'a -> 'c vector
```

Figure 5: Simplified types of skeletons constructions

Skeletons offer inputs and outputs making it possible to compose them. While looking similar, skeletons and transformations (for instance `MAP` and *map*) are different and serve different purposes (figure 5 presents skeletons constructions types). Skeletons can use existing kernels within composition while transformations generate complex code from simple sub-kernels. Transformations can generate kernels (without host composition) that can be used within composition skeletons.

4.3 Composition and automatic optimizations

Compositions explicitly describe the relations between skeletons and data. A common composition is `PIPE`. It composes two skeletons, taking the output of the first one as the input of the second one. `PIPE` explicitly expresses the dependencies between two computations and the data they handle, and thus, helps express complex algorithms. Besides, using this functional style helps us provide automatic optimizations, as it automatically gives us the data dependency graph of our programs. Indeed, using `PIPE`, we can easily start the transfers of the vectors needed by the second skeleton while computing the first one. This provides automatic overlapping of transfers by computations, and thus, improves the overall performance.

```
val PIPE : ('a,'b,'c) skeleton ->
           ('d,'c,'e) skeleton ->
           ('a,'b,'e) skeleton
```

Many algorithms can benefit from the use of `PIPE`, for instance, 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 each iteration, the vector b_k is multiplied by the matrix A and normalized. This example 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

This computation can be expressed through the use of `MAP` and `REDUCE` skeletons that can be composed with `PIPE`. Figure 6 shows the code of this program using skeletons. Beside transfer overlapping, piping skeletons also reduces the number of times

```
...
while (!norm > 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;
```

Figure 6: Power Iteration with skeleton compositons

SPOC will compute the checks and transfers optimizations needed for each run.

Version	Time (s)	Speedup
Sequential OCaml	637	1
OCaml + SPOC	101	6.31
OCaml + SPOC + Skeletons	81	7.86
OCaml + SPOC + Composition	74	8.61

Test machine : 1 Intel Core i7-3770 CPU + 1 AMD Radeon HD6950 GPU

Table 2: Results using skeleton composition

Table 2 shows performance results for this algorithm comparing four versions of it : using sequential OCaml, translating computations to GPGPU kernels, using GPGPU kernels within skeletons and composing GPGPU skeletons.

Here, using GPGPUs already improves performance. When using Skeletons, SPOC automatically optimizes the layout of the computations to maximize parallelism on the many computing units of our GPU. Using `PIPE`, SPOC can overlap some transfers with computations, and thus, improve performance.

This kind of overlapping could, for example, also be implemented in the `MAP` skeleton to provide some kind of multi-buffering for large enough data sets, computing over some part of the vector while transferring another part.

5. Related Works

GPGPU programming is a complex field but it can offer very high performance. To harness it, multiple solutions have emerged, some based on high-level languages and libraries, while others target scheduling and kernel composition.

5.1 High-level GPGPU programming

Most languages currently provides bindings to Cuda or OpenCL API to use GPGPU programming, including high level programming language. This section presents some libraries and tools providing higher-level features and compares them with our solutions. Most keep the same approach : provide DSLs dedicated to GPGPU kernels, with a runtime library to handle transfers and composition.

Accelerate [15] is a Haskell library that features a DSL to describe GPGPU computations. As its multi-core counterpart REPA, Accelerate is based on array computations. Similarly to SPOC skeletons and transformations, Accelerate computations take array (or vectors) as inputs and produce arrays as outputs. As with Sarek, Accelerate computations are transformed to an internal representation that is compiled to Cuda, OpenCL or REPA code, using specific plugins. Accelerate also allows to use external CUDA code via an FFI [16] making it very similar to SPOC. However, comes from the possibility, for Sarek to describe computations that are not vector operations.

FSCL [17], for F#, is composed of two projects *FSCL.Compiler* and *FSCL.runtime*. The first is a F# compiler that produces OpenCL code. FSCL features several high-level improvements overs classic OpenCL such as generic types, automatic array length or record and structs. In association, the runtime library offers to access OpenCL kernels from .NET. This approach is very similar to ours, providing a solution dedicated to kernels as well as another to compose kernels with high-level languages and tools. The main difference comes mostly from the compiler that offers higher-level features than Sarek but only targets OpenCL. As SPOC, FSCL.runtime offers specific parallel collections that are automatically managed.

Several tools exist for the Python language. Most are based on NumPy, the main numerical library for Python. For instance, **Parakeet** [18] and **Copperhead** [19] use subsets of the Python language to declare numerical computations that can be compiled just-in-time to multi-core or GPGPU kernels. Both offer data-parallel operators (like map and reduce) and use specific data-types. **Numba** [20] is another Python JIT compiler that compiles specific python expressions to parallel LLVM code. Its commercial version offers compatibility with Cuda.

Aparapi [21] is a library for Java targeting OpenCL. It describes kernels by inheriting specific Java classes in a similar way than with Java threads. Aparapi offers automatic memory management and device detections.

RustGPU [22] describes GPGPU kernels in Rust, that are compiled to Cuda kernels. It, then, offers to use those as external kernels *via* simple bindings.

With **Harlan** [23], developers can compose kernels with the Scheme language. It describes them as operations over vectors directly within Scheme. It also features specific combinators such as reduce.

Thrust [24] is a C++ library that targets Cuda. It offers abstraction overs classic C++ Cuda Code by providing vector combinators very similar to SPOC skeletons.

All describe kernels with their host language. Their kernel DSL or compiler offer languages very similar to their host language, and thus, increase the overall program consistency. However, they often offer either skeletons and vector combinators or kernel DSLs and most of the time cannot use external GPU kernels. SPOC also features multiple and unified backends and thus can be used in heterogeneous systems.

5.2 Kernel composition libraries

High-level tools exist for GPGPU programming, but few are focused on improving composition *via* automatic scheduling of GPGPU computations. Several libraries and compiling tools focus on this specific area, but are mainly targeting C/C++.

StarPU [25] and **XKaapi** [26] are C and C++ libraries that specifically target heterogeneous systems. Both describe tasks and the dependencies between them. Their runtimes automatically dispatch these tasks on several computation units, automatically scheduling data transfers and computations.

Par4All [27] is an automatic parallelizing and optimizing source to source compiler. From already data-parallel C or Fortran, it optimizes memory and computations dispatch overs parallel devices. Through static analysis, it can automatically schedule computations and transfers, offering optimizations such as transfers overlapping. This approach is very different from the others and takes place at compile-time.

6. Conclusion and Future Work

High performance GPGPU programming and numerical computations are commonly done in an imperative way. However, func-

tional or object compositions can improve the expression of complex algorithms. Using OCaml, we developed the SPOC library for GPGPU programming. By increasing abstraction, and offering several ways of composing GPGPU kernels, SPOC and Sarek, with OCaml, make GPGPU more accessible and safe. They provide static type checking, as well as automatic memory management, including transfers, to the overall program. To go further, Sarek offers direct transformation of small GPGPU sub-kernels into complete kernels associated with host composition code. To make composition possible, SPOC provides tools to include GPGPU kernels within skeletons. Skeletons improve code reuse and expressiveness. They explicitly describe relations between a kernel and its required data. Besides, SPOC directly uses this information to provide automatic optimizations when mapping kernels to GPGPU computing units. Skeletons can be composed. With compositions, the relations between skeletons can be explicitly expressed. Thus, SPOC can define an optimized automatic scheduling of transfers and computations. Skeletons, as well as transformations, also help to describe polymorphic, reusable and complex computations.

To increase composability, and thus, improve GPGPU programming, we intend to improve Sarek and our skeleton and transformation library. Sarek is a playground for code transformation and generation. We plan to use it to offer more transformations and them with skeletons composition. This will provide mechanisms to directly transform kernels by splitting/merging them and automatically generate synchronizations, reducing the number of kernel call when possible and keeping as much computation as possible on the GPU. To improve automatic composition, we also intend to add automatic cost evaluation to Sarek. Thus, we could provide skeletons dedicated to heterogeneous architectures automatically dispatching computations were they will be handled the best.

Another domain that could benefit from our approach is web programming. Web applications are becoming very demanding. To make the development of compute intensive applications such as multimedia applications possible, it becomes mandatory to have access to high performance tools. Thus, we intend to adapt ours to web environments, with the use of the WebCL [28] framework for JavaScript, associated with the existing js_of_ocaml [29] compiler that compiles OCaml byte-code to JavaScript.

References

[1] Nvidia, *Cuda C Programming guide*, 2012.

[2] K. O. W. Group, "OpenCL 1.2 specifications," 2012.

[3] X. Leroy, D. Doligez, A. Firsch, J. Garrigue, D. R. Remy, and J. Vouillon, "The OCaml System Release 4.01 : Documentation and User's Manual," tech. rep., Inria, 2013. `http://caml.inria.fr`.

[4] M. Bourgoin, E. Chailloux, and J.-L. Lamotte, "Spoc: GPGPU Programming Through Stream Processing with OCAML," *Parallel Processing Letters*, vol. 22, pp. 1–12, May 2012. `http://www.algo-prog.info/spoc`.

[5] M. Bourgoin, E. Chailloux, and J.-L. Lamotte, "Efficient Abstractions for GPGPU Programming," *International Journal of Parallel Programming*, pp. 1–18, 2013.

[6] NVIDIA, "Cublas Library," 2012. `http://developer.nvidia.com/cublas`.

[7] S. Tomov, R. Nath, P. Du, and J. Dongarra, "MAGMA Users Guide," 2011.

[8] J. Donham and N. Pouillard, "Camlp4 and Template Haskell," in *ACM SIGPLAN Commercial Users of Functional Programming*, pp. 6:1–6:1, ACM, 2010.

[9] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU communication management and optimization," *ACM SIGPLAN Notices*, pp. 142–151, 2011.

[10] N. S. Scott, M. P. Scott, P. G. Burke, T. Stitt, V. Faro-Maza, C. Denis, and A. Maniopoulou, "2DRMP: A Suite of Two-Dimensional R-matrix Propagation Codes.," *Computer Physics Communications*, pp. 2424–2449, 2009.

[11] M. Bourgoin, E. Chailloux, and J.-L. Lamotte, "Retour d'expérience: portage d'une application haute-performance vers un langage de haut niveau," in *la Conférence d'informatique en Parallélisme, Architecture et Système (ComPas)*, 2013.

[12] P. Kipfer and R. Westermann, "Improved gpu sorting," in *GPU gems*, vol. 2, pp. 733–746, 2005.

[13] M. Cole, "Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming," *Parallel computing*, pp. 389–406, 2004.

[14] R. Di Cosmo, Z. Li, M. Danelutto, S. Pelagatti, X. Leroy, P. Weis, and F. Clément, *CamlP3l 1.0: User Manual*, 2010. `http://camlp3l.inria.fr/`.

[15] M. Chakravarty, G. Keller, S. Lee, T. McDonell, and V. Grover, "Accelerating Haskell array codes with multicore GPUs," in *Proceedings of the sixth workshop on Declarative aspects of multicore programming (DAMP)*, pp. 3–14, ACM, 2011.

[16] R. Clifton-Everest, T. McDonell, M. Chakravarty, and G. Keller, "Embedding foreign code," in *Practical Aspects of Declarative Languages* (M. Flatt and H.-F. Guo, eds.), vol. 8324 of *Lecture Notes in Computer Science*, pp. 136–151, Springer International Publishing, 2014.

[17] G. Cocco, "FSCL F# to OpenCL Compiler and Runtime," 2013. `http://www.gabrielecocco.it/fscl/`.

[18] A. Rubinsteyn, E. Hielscher, N. Weinman, and D. Shasha, "Parakeet: A just-in-time parallel accelerator for python," in *The 4th USENIX Workshop on Hot Topics in Parallelism*, USENIX, 2012.

[19] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: compiling an embedded data parallel language," in *ACM SIGPLAN Notices*, vol. 46, pp. 47–56, ACM, 2011.

[20] T. Oliphant, "Numba python bytecode to LLVM translator," in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2012.

[21] A. INC, "Aparapi," 2013. `http://code.google.com/p/aparapi/`.

[22] E. Holk, M. Pathirage, A. Chauhan, A. Lumsdaine, and N. D. Matsakis, "GPU Programming in Rust: Implementing High-Level Abstractions in a Systems-Level Language," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW)*, pp. 315–324, 2013.

[23] E. Holk, W. E. Byrd, N. Mahajan, J. Willcock, A. Chauhan, and A. Lumsdaine, "Declarative parallel programming for gpus.," in *Internation Conference on Parallel Computing (PARCO)*, pp. 297–304, 2011.

[24] J. Hoberock and N. Bell, "Thrust: C++ Template Library for CUDA," 2009. `http://code.google.com/p/thrust/`.

[25] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, 2009.

[26] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1299–1308, 2013.

[27] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. O. McMahon, F.-X. Pasquier, G. Péan, P. Villalon, *et al.*, "Par4all: From convex array regions to heterogeneous computing," in *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, 2012.

[28] T. Aarnio and M. Bourges-Sevenier, "WebCL 1.0 specification," *Khronos WebCL Working Group*, 2014.

[29] J. Vouillon and V. Balat, "From Bytecode to JavaScript: The js_of_ocaml Compiler," *Software: Practice and Experience*, 2013.