Profiling High Level Heterogeneous Programs

Using the SPOC GPGPU framework for OCaml

Mathias Bourgoin[1], Emmanuel Chailloux[2], and Anastasios Doumoulakis[3]

[1] Univ. Orléans, INSA Centre Val de Loire,
LIFO EA 4022,
FR-45067 Orléans, France
Mathias.Bourgoin@univ-orleans.fr

[2] Sorbonne Universités, UPMC Univ. Paris 06,
UMR 7606, LIP6,
F-75005 Paris, France
Emmanuel.Chailloux@lip6.fr

[3] Sorbonne Universités, UPMC Univ. Paris 06
F-75005 Paris, France
Anastasios-Louis.Doumoulakis@etu.upmc.fr

Abstract. Heterogeneous systems are widespread. When neatly used, they enable an impressive performance increase. However, they typically demand developers to combine multiple programming models, languages and tools into very complex programs that are hard to design and debug. Writing correct heterogeneous programs is difficult, achieving good performance is even harder. To help developers, many high-level solutions have been developed, in particular for GPGPU programming (combining a CPU host with one or many GPUs). Most of the time, these solutions generate part of the code that will actually be executed and insulate the programmer from low-level architectural details. They help build efficient and safer programs, but they leave programmers clueless when facing complex bugs or when trying to optimize their application further. In this paper, we propose to enable profiling of heterogeneous applications built using such tools. We present our solution for the SPOC GPGPU framework for the OCaml programming language. Using this framework, we show how it is possible to provide relevant information to the developer focusing on the different parts of GPGPU programs: firstly, by instrumenting the high-level runtime host library of the framework, second, by benefiting from the dynamic generation of co-processor subprograms (GPGPU kernels) in this kind of frameworks to automatically instrument and measure computations, in order to provide developers understandable feedback on their programs. The solution we present, using simple examples, enables profiling of high-level systems in very heterogeneous systems that can combine a CPU host with numerous very different co-processors.

1 Introduction

Most computers (from supercomputers to smart devices) are heterogeneous. They combine several very different computing devices, each with its set of features and dedicated to specific computations. A common example is a multi-core CPU associated with a parallel co-processor, often a GPU. Heterogeneous programming implies using these multiple different architectures in one program conjointly. It aims at benefiting from each architecture specific features to improve the overall program, most of the time focusing on performance. Each associated device in a heterogeneous system comes with its own programming paradigms, set of languages, compilers, libraries and tools. This leads to mixing completely different subprograms using very low-level APIs. As such, targeting heterogeneous systems is difficult and error-prone, using it to improve performance is even harder.

With the growing interest in general purpose GPU programming (GPGPU) (using the CUDA and OpenCL frameworks), multiple tools have appeared to make heterogeneous programming (with GPUs) simpler. They combine language constructs (from libraries to completely new languages), with compilers and runtime libraries to provide high-level abstractions that help harness GPGPU programs. Two kinds of solutions have been developed: *(i)*first, compiler directives (such as with OpenMP 4.0[4] or OpenACC[5]) that make possible to specify parts of programs (mostly loops) as targets for automatic transformations to GPGPU computations, *(ii)* second, domain specific languages (DSLs) to describe computations dedicated to the co-processors (for instance GPGGPU kernels) that are embedded into general purpose programming languages with dedicated programming constructs to compose these computations. Some are based on statically typed environments to benefit from static type safety to detect at compile-time many errors that would be otherwise difficult to correct when triggered at runtime, such as Aparapi[6] for Java, Accelerate [6] for Haskell or FirePile [16] for Scala. Others are mostly focused on abstractions to improve productivity, offering specific constructs that automatically handle co-processors, such as Parakeet [19] and Copperhead [5] for Python, NT2 [9,14] and Thrust[7] for C++. Most of these solutions automatically handle communications between devices (host/co-processor(s)). Besides, they often generate, statically or dynamically, parts of the complex code targeting co-processors. They improve productivity and provide easier programming than with basic low-level tools offered by hardware vendors. However, as intended, they hide most of the actual complex code that runs on the hardware. This forbids programmers from further optimizing their code or look for complex bugs. Indeed, while most of these high-level solutions produce code that is compatible with low-level debuggers or profilers, it quickly becomes very difficult to tie the low-level generated code with the original source code written in the first place.

[4] OpenMP. `http://www.openmp.org/`
[5] OpenACC. `http://www.openacc.org/`
[6] Aparapi. `http://code.google.com/p/aparapi`
[7] Thrust: C++ Template Library for CUDA. `http://thrust.github.io/`

In this paper, we propose to tackle this issue by improving profiling of heterogeneous programs written with a generative high-level set of tools for the high-level programming language, OCaml. Using this framework, we show the main features that are necessary to improve profiling of heterogeneous programs and that could be implemented in other solutions as well. The next section presents the SPOC library and the Sarek DSL that were previously developed for OCaml to target and abstract GPGPU programming. We show how SPOC unifies both OpenCL and CUDA frameworks, abstracts memory transfers and device management, while Sarek helps easily express GPGPU kernels that are efficient, portable, safe and extensible (through meta-programming like techniques). In Section 3, we describe how we propose to profile and debug OCaml programs that are using SPOC and Sarek. This represents the primary contribution of this paper. First, we focus on the host (CPU) part, and then, on the kernel part of heterogeneous programs. The kernel profiling system is based on the Sarek DSL and thus, provides a unified solution that is independent of both CUDA and OpenCL low-level frameworks. The overall profiling solution, similarly to the SPOC framework, is portable and compatible with very heterogeneous systems that combine a CPU host with multiple different co-processors (compatible with CUDA or OpenCL). Section 4 presents related works. Finally, Section 5 concludes and discuss future works.

2 High-Level Heterogeneous Programming with OCaml

To make heterogeneous programming (using GPGPU frameworks) simpler and safer, we previously developed a set of tools based on the OCaml high-level programming language. OCaml is developed by Inria [13]. It is a multi-paradigm (functional, imperative, object, modular) language. It can be compiled to efficient native code for performance as well as to dedicated virtual machine byte-code for portability. It features an efficient and customizable memory manager. OCaml is a general purpose programming language with an emphasis on expressiveness and safety. It is a statically typed language with type inference which helps rule out many programming errors. However, it is also interesting to note that as with many high-level programming languages and frameworks, OCaml eases the overall development of complex programs, but does not offer many tools for profiling and debugging. Profiling is handled differently for byte-code and native code.[8] Byte-code profiling gives developers feedbacks on the number of calls of different parts of the programs while native profiling shows the actual running time of different parts of the programs. The native solution may sound appealing, but is in fact difficult to use correctly as the code written by the programmer is intertwined (in the profiling output) with the OCaml runtime library calls (such as those from the garbage collector).

[8] `https://ocaml.org/learn/tutorials/performance_and_profiling.html#Profiling`

Using OCaml, we developed the open-source[9] SPOC (Stream Processing OCaml) library [2]. It is based on the CUDA and OpenCL frameworks. It offers several abstractions over memory transfers as well as device management. SPOC only focuses on the host program, but can interoperate with native GPGPU kernels (written using the OpenCL and CUDA C subsets). Beside, we developed a domain specific language, Sarek (Stream ARchitecture Extensible Kernels), dedicated to describing kernels. It is these kernels, that are easier to write than the native ones, but difficult to tie to what actually runs on the co-processor, that we will focus on profiling in the next sections.

2.1 Host Programming

To provide portability, SPOC unifies both CUDA and OpenCL APIs. SPOC automatically detects all devices compatible with it at runtime. Associated with a common API, this can be used to indifferently and conjointly handle multiple co-processors (from any framework). This eases the expression of complex programs dedicated to very heterogeneous architectures. SPOC introduces a specific data set to OCaml: vectors. Vectors keep information about their current location in the system (on host or co-processor memory). Thus, SPOC can automatically trigger transfers when needed. In particular, SPOC checks that every vector used by a GPGPU kernel is present in the co-processor memory (and triggers transfers if required) before launching the computation. Similarly, when the host reads or writes in a vector, SPOC checks its location and transfers it if needed.

2.2 Programming Kernels

SPOC provides two solutions to express GPGPU kernels. First, one can use interoperability with native CUDA and OpenCL kernels. This eases code reuse and helps with writing bindings with existing high performance libraries. The second solution is to use Sarek, a DSL built into OCaml that is dedicated to GPGPU kernels. It is an expression oriented language with an ML-like imperative core. Sarek is based on the C subsets provided by OpenCL and CUDA and provides specific primitives to handle co-processors computation units efficiently. Sarek kernels are compiled in two steps. First, at compile time, we use a Camlp4 [8] OCaml syntax extension to type check Sarek kernels and generate an internal representation that is directly embedded into the host program. Then, at runtime, we provide OCaml functions that can translate the internal representation into actual native kernels that can be executed on GPGPU co-processors. Fig. 1 shows the Sarek kernel used for a simple machine learning example. The native profiling kernel generated from this one and the output our profiling tool provides are respectively presented in Fig. 3 and Fig. 4 and discussed in Section 3.2. This kernel comes from a program (based on a benchmark by P. Tomson[10]) that uses the k-Nearest Neighbors algorithm. From a training

[9] Open-source distribution: `http://www.algo-prog.info/spoc`

[10] http://philtomson.github.io/blog/2014/05/29/comparing-a-machine-learning-algorithm-implemented-in-f-number-and-ocaml/

```
let compute = kern trainingSet data res setSize dataSize ->
  let open Std in
  let computeId = thread_idx_x + block_dim_x * block_idx_x in
  if computeId < setSize then (
    let mutable diff = 0 in
    let mutable toAdd = 0 in
    let mutable i = 0 in
    while(i < dataSize) do
      toAdd := data.[<i>] - trainingSet.[<computeId*dataSize + i>];
      diff := diff + (toAdd * toAdd);
      i := i + 1;
    done;
    res.[<computeId>] <- diff)
  else
    return ()
```

Fig. 1: Sarek kernel used in a k-Nearest Neighbors machine learning program

set of 5000 pictures of a handwritten digits and a validation set of 500 labeled examples, it compares each of the 500 validating samples against the 5000 training examples and returns the label with the closest match. As OCaml, Sarek offers type inference with static type checking as well as an OCaml-like syntax for more consistency with the host program. The kernel here is named `compute` and is recognized by the use of the keyword `kern`. GPGPU co-processors have several kinds of memory with different properties (shared/local, different bandwidth, etc.). As with CUDA/OpenCL, it is mandatory to manually optimize the use of these different memories to achieve high performance. The parameters of our simple kernel are stored in the co-processor global memory. Local values can be declared using the `let in` construction, with the `mutable` keyword to explicitly express mutability. Mutable variables (including vectors) stored in global and local memory can be modified using the `<-` and `:=` syntax respectively. As with OpenCL and CUDA kernel-dedicated C subsets, Sarek provides primitives and constants to manage the many computational units of the GPU (such as `thread_idx_{xyz}`, `block_idx_{xyz}`, `block_dim_{xyz}`). An issue may arise in this kernel if `dataSize` (that is a constant declared in the host program) is smaller than the picture represented in `data` (that is taken from the validation set). In this case, the kernel will run without error, but will produce a wrong result, that may be difficult to debug. Profiling may help in this matter as we will see in the next sections. As native kernels are generated at runtime using "simple" OCaml functions, it is also possible to transform kernels to change their behavior. For instance, we built several array oriented constructs (Map/Reduce-like skeletons) that abstract GPGPU programming even further and offer several automatic optimizations [3].

OCaml, SPOC and Sarek help expressing programs that can benefit from GPGPU co-processors. Together, they provide a portable solution that can target either Cuda or OpenCL compatible co-processors. Besides, this solution can be used in very heterogeneous systems combining multiple different devices (each

one compatible with any of the two frameworks). For instance, it can be used in a system associating a multi-core CPU (being the host, as well as seen as an OpenCL co-processor), an integrated GPU (compatible with an OpenCL implementation that can be different from the CPU one) and several dedicated accelerators (for instance CUDA-compatible GPUs or some FPGAs). CPU implementations of OpenCL can make use of either the CPU cores themselves or the integrated graphics. SPOC and Sarek unify the programming of such systems and automatically generate the code targeting each accelerator making it very suitable for very heterogeneous systems. However, they hide most of the GPGPU programming part. This builds a gap between the code written by the programmer and the code that is actually executed. For instance, while kernels produced from Sarek can be used with profilers/debuggers provided by hardware vendors, it quickly becomes very difficult to tie the output of such tools (based on the OpenCL/CUDA C subsets or assembly languages) with the source code of the program (that the programmer has written in Sarek). Moreover, in very heterogeneous systems, each accelerator will need a specific profiler/debugger to handle the part of the program it executes, making it a very complex and cumbersome task. In the next section, we present our solution to profile heterogeneous programs using OCaml, SPOC and Sarek.

3 Profiling

There is a tradeoff to make between control over the software and high-level abstractions. SPOC is on the high-level side, which brings simplicity, efficiency and straightforwardness. But programming at a higher level must not mean a systematic loss in performance, this is why we need a way to monitor performance and provide the programmer with useful information about the execution of the program he wrote. Fig. 2 shows how we implemented our solution[11] to profile heterogeneous programs written in OCaml with SPOC and Sarek. At compile-time, it only consists in linking the program with a modified SPOC runtime library. At run-time, this library changes the behavior of SPOC programs (as is shown here for a vector addition program) to enable profiling of both the host part and the GPGPU kernels as is described in this section.

3.1 Host part

Here, we present the profiling system for the host code, corresponding to SPOC code described in section 2.1. One important aspect to consider is the impact that profiling can have on the execution of a program, it has to be minimal despite the fact that we need to collect precise information to help with debugging, this means that we must be especially cautious when dealing with asynchronous functions. The SPOC library uses asynchronous transfers and kernel launches

[11] SPOC experimental `profile` branch:
https://github.com/mathiasbourgoin/SPOC/tree/profile

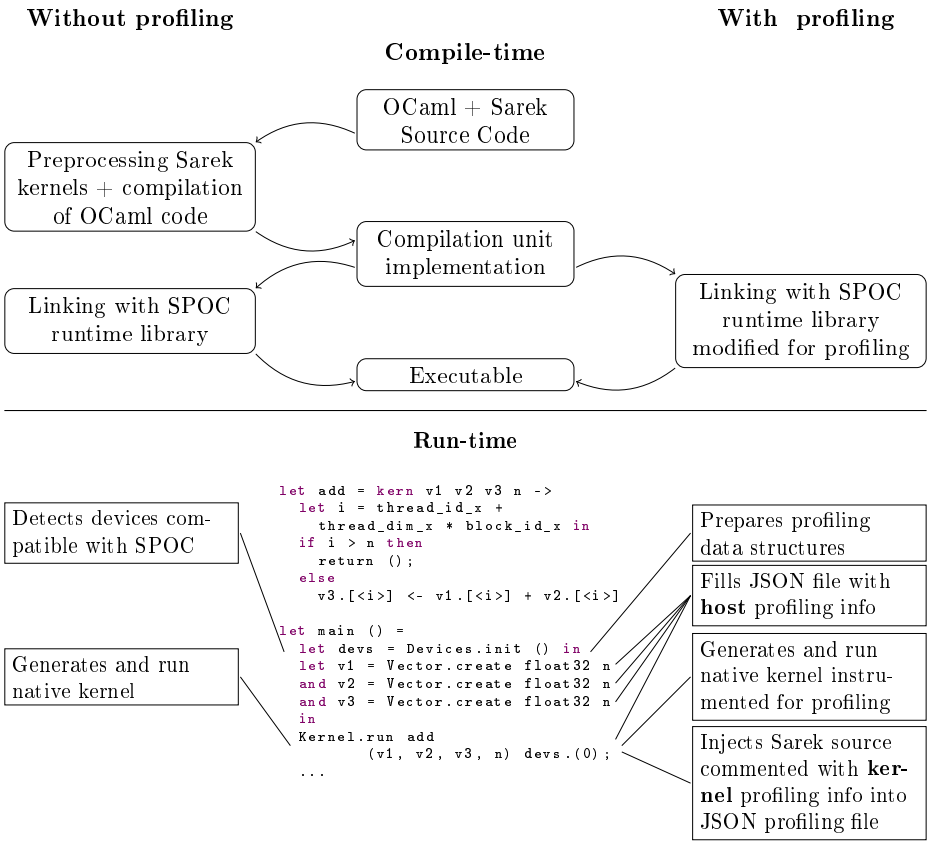

Fig. 2: Compilation/Execution of SPOC programs with/without profiling

to improve efficiency, but they are harder to trace. It is also necessary that we have no major differences between OpenCL and CUDA API calls for profiling, as we want our profiling to give similar feedback on all platforms that can run SPOC programs regardless of the framework used. In particular, we want to be able to know the state of the memory of the host at any point in time during the execution. To achieve that, we collect the following information:

- The list of GPGPU devices, detected by SPOC, along with all the data associated with them (name, VRAM amount, clock frequency, etc...)
- The allocations and de-allocations of SPOC vectors in memory (whether it be CPU or GPU memory)
- The transfers of vectors between the CPU and the GPUs (including the direction of the transfer, the GPU involved in each transfer, the duration, the size, etc...)
- The compilation, loading and execution times for every kernel involved in the computation.

All the relevant data is written to a JSON file during runtime. It is easily done by copying SPOC internal representation of elements, such as vectors, to the file containing the profiling data. When dealing with asynchronous calls we use the queue mechanisms provided by OpenCL and CUDA to have accurate information about asynchronous events such as the duration of a vector transfer. After the SPOC program is done running we can put together the state of the memory at any point during the execution and the JSON format makes it possible to make a profiling visualizer that can help with debugging programs and optimizing memory transfers. A visualizer can show on a timeline the different events taking place in the program. It could present a timeline per co-processor on the system and show the events associated with this device: memory transfers, compilation and launch of kernels. It could also present a timeline per SPOC vector to visualize their transfers during their lifetime. This kind of tool can help the user understand the behavior of their program in the automated environment of SPOC, in particular when using high-level algorithmic constructs (*à la* Map/Reduce) that are offered in the SPOC tool-set. Besides, it can also be useful for manual optimizations, allowing the user to use explicit primitives to trigger specific events at a desired time. For instance, users could trigger transfers as soon as possible to increase performance, instead of letting SPOC trigger them only when needed (at the latest). Moreover, it is compatible with the whole SPOC library, which means that it can be used to profile programs running on CUDA or OpenCL (from any hardware dependant implementation) devices indifferently and conjointly (in very heterogeneous systems).

3.2 Kernel part

To describe GPGPU kernels, SPOC offers two solutions: interoperability with native CUDA/OpenCL kernels, as well as the Sarek DSL. Native kernels are seen as black boxes associated with a specification (mainly name and types of inputs/outputs). As such, it is only possible to use the low-level tools provided by hardware vendors to profile such kernels. Using Sarek, it is possible to use those tools as well, but as we stated before, it quickly becomes complex to link the output of such tools to the code written in the first place. In the rest of this section, we will only focus on Sarek kernels. We propose to use transforming functions on Sarek kernels to instrument the generated native kernels in order to get profiling information during the execution of a kernel and link back this information to the Sarek source code.

Sarek provides portability and can be used on any CUDA or OpenCL-compatible device. Thus, to stay portable, we cannot use specific architectures counters to profile kernels and must implement our own counters in a way that stays compatible with all frameworks and architectures. While it limits the scope of our profiling counters, it still allows us to provide information on a kernel behavior. Moreover, as for the host part, it provides a portable solution that can also be used in very heterogeneous systems that combine multiple devices normally profiled with dedicated (and incompatible) profiling tools. Our current solution provides three kinds of counters:

– control flow counters that measure the total number of visits of every constructs: loops, branches (conditional expressions, pattern matching), functions,
– memory counters that measure the total number of read/writes in the different types of memory available on the co-processor (global, local, etc.),
– counters for the number of floating operations within the kernel.

These counters are global values (shared by all threads participating in the kernel execution) that are incremented using dedicated atomic operations. Fig. 3

```
__kernel void spoc_dummy (
                            __global unsigned long  * profile_counters,
                            __global int* trainingSet, __global int* data,
                            __global int* res, int setSize, int dataSize ) {
  int computeId;
  int diff;
  int toAdd;
  int i;
  computeId = ((get_local_id (0)) +
                           ((get_local_size (0)) * (get_group_id (0)))) ;
  if ( computeId < setSize ){
    spoc_atomic_add(profile_counters+3, 1); // control if
    spoc_atomic_add(profile_counters+0,1); // global mem store
    diff = 0 ;
    toAdd = 0 ;
    i = 0 ;
    while (i < dataSize){
      spoc_atomic_add(profile_counters+1,2); // global mem load
      spoc_atomic_add(profile_counters+2, 1); // control while
      toAdd = (data[i] - trainingSet[((computeId * dataSize) + i)]) ;
      diff = (diff + (toAdd * toAdd)) ;
      i = (i + 1);} ;
    res[computeId] = diff;;
  }
  else{
    spoc_atomic_add(profile_counters+4, 1); // control else
    return  ;
  }
}
```

Fig. 3: Generated OpenCL profiling kernels

presents the OpenCL kernel generated for profiling from the kernel in Fig. 1. Gray lines show the code added for profiling. The possible bug that could be seen if `dataSize` is smaller than the picture represented in `data` (as told in section 2.2) can be found more easily with profiling counters. The iterations of the while loop will be too small compared to the size of `data`. The profiling OpenCL kernel is generated at runtime, using the kernel internal representation embedded in the OCaml host program. The total number of counters needed for the kernel is computed during this generation. A SPOC vector is then created to store these counters and is passed as the first parameter of the kernel when launched for profiling. The kernel is run normally, incrementing counters during

its execution. At the end of its execution, SPOC brings back the profiling counters vector in host memory and uses it to format a profiling output for the user. In order to make things as easy as possible for the user, the output is formatted as Sarek code corresponding to the kernel, commented with information taken from the counters that have been updated during the profiling phase. The output is not exactly the Sarek source code written originally. Indeed, the kernel internal representation is the only available representation of the kernel at runtime. It has been created at compile-time from the original Sarek source and some information (that has become unnecessary at this point) has been lost (in particular function names and variable kinds (mutable/constant)) and instructions in the kernel may have been reordered. A correct execution of the profiling kernel from

```
(* Profile Kernel *)
kern trainingSet data res setSize dataSize ->
(** ### global_memory stores : 5000 **)
(** ### global_memory loads : 7840000 **)

let mutable computeId = (thread_idx_x + (block_dim_x * block_idx_x)) in
  if (computeId < setSize) then
  (**  ### visits : 5000 **)
    let mutable diff = 0 in
    let mutable toAdd = 0 in
    let mutable i = 0 in
    while i < dataSize do
    (**  ### visits : 3920000 **)
      toAdd := (data.[<i>] - trainingSet.[<((computeId * dataSize) + i)>]);
      diff := (diff + (toAdd * toAdd));
      i := (i + 1);
    done;
    res.[<computeId>] <- diff;
  else
  (**  ### visits : 120 **)
    return ()
```

Fig. 4: Output of profiling Sarek kernel

Fig. 3 produces a commented Sarek kernel as shown in Fig. 4. Comments in the gray lines present the profiling counters, as they are formatted in the profiling output. The number of visits in the while loop helps the user to check that its parameter were correct (here that `dataSize` is not smaller than the length of the vector `data`. It can also be used for optimizations. For instance, a high number of global memory loads/stores may point out that the user should use faster memory. Besides this simple Knn program, or solution as been tested with multiple examples from our SPOC simple examples/benchmarks (such as Matrix multiplication, fractals, monte-carlo pi computations). It has also been tested with simple examples on multi-devices systems with expected results. However, it still needs to be tested in larger programs that use numerous very different kernels and vectors, in particular in very heterogeneous systems.

4 Related Work

Heterogeneous programming is based on low-level frameworks specialized for the co-processors they target (most of the time, each hardware vendor offers its own framework). For GPGPU programming CUDA is dedicated to NVIDIA architectures and OpenCL is a standard shared by multiple vendors (including NVIDIA, AMD, Intel, and ARM). NVIDIA provides a profiling tool for its architecture that is compatible with CUDA, *nvprof* [15]. It traces all CUDA host API calls (memory transfers, kernels launches) and show specific GPU metrics (percent of memory operations, control flow operations etc) that can help understand and optimize CUDA kernels. This information is also available in a graphical tool, the *NVIDIA Visual Profiler* as well as plugins for several integrated development environment (such as Eclipse or Visual Studio). In the same vein, OpenCL frameworks providers offer dedicated tools. Most of the time, they are compatible with any OpenCL runtime for the host part but only with the hardware they provide for the kernel part (for instance, AMD *APP (Accelerated Parallel Processing) Profiler* [17], Intel *System Analyzer and Platform Analyzer* [7] or *ARM DS-5 Streamline Profiling tool*[12]). Besides GPGPU systems, other heterogeneous platform vendors provide dedicated tools for their hardware. For instance, FPGA vendors, such as Altera or Xilinx, offer specific tools for their devices (for instance the *Nios II Embedded Evaluation Kit* and the *Xilinx Platform Studio*). As stated before, these tools can provide precise metrics, but each one is only compatible with one set of devices and they are difficult to tie to high-level solutions that tend to hide the actual code that manages the co-processors. Moreover, in very heterogeneous systems, it quickly becomes necessary to use several of these profiling tools together (one for each kind of device). This demands to use them one at a time as they cannot interoperate. Each profiler will profile a different run of the program, focusing only on what is executed on the dedicated hardware. This makes profiling very complex to achieve in such systems and confirms the need for generic tools that can target multiple frameworks and hardware accelerators used in parallel in heterogeneous applications.

High-level programming frameworks that are still using low-level subprograms to target co-processors exist for heterogeneous programming. This is typically the case for StarPU [1] and XKaapi [12] that automatically schedules computations in heterogeneous systems. With such tools, it is easier to tie the actual running code with the profiling output of low-level tools. However, as the framework handles scheduling automatically with specific strategies (that can be customized by the user) they also provide profiling features for further optimization and debug.[13]

High-level programming languages are hard to profile with classic low-level tools (as we have seen for OCaml in section 2). Specific tools are being developed

[12] `https://developer.arm.com/products/software-development-tools/ds-5-development-studio/streamline`

[13] StarPU profiling: `http://starpu.gforge.inria.fr/files/doc/starpu-1.1.2/html/HowToOptimizePerformanceWithStarPU.html`
XKaapi profiling: `http://kaapi.gforge.inria.fr/#!perfcounter.md`

to tackle this issue. In OCaml, an important property that is automatically managed and difficult to trace is memory management. The *OCaml Memory Profiler* [4] focuses on memory and help optimize programs that either tends to allocate too much memory or spend too much time in the garbage collector. This tool could be used in association with SPOC and our own solution for profiling as SPOC vectors are automatically managed by the OCaml memory manager to provide feedbacks on the overall program (not only the GPGPU part).

An important part of our contribution is to provide information that is easy to tie back to the code that was written by the user. Another solution is explored with the Leo framework [11] that integrates the Dandelion [18] high-level GPGPU framework with GPU Lynx [10] that is a dynamic instrumentation toolchain for GPGPU frameworks. Leo proposes to use the feedback of automatically generated profiling kernels (as with our solution) to automatically optimize the kernels. This solution could also be explored in future experiments, with SPOC and Sarek instead of directly providing the profiling output to the user to help him understand, optimize and debug his code.

5 Conclusion and Future Work

Heterogeneous programs are complex to write. High-level tools can manage lots of complex low-level mechanisms that are mandatory to benefit from the multiple, different devices in the system. By doing so, they hide a large part of the actual program to the user which makes debugging and optimizing more difficult. Simple profiling tools can help achieve a better understanding of such programs. As we have seen, even with a high level of abstraction it is possible to provide users with relevant information on their programs, from the profiling of asynchronous automatically triggered events (in particular communications between devices) to the obtaining of specific metrics in intensive computations (that run on co-processors). Using SPOC and Sarek, it is possible to provide useful feedback, first, by instrumenting the SPOC runtime library for the host part and, then through use of generative functions that transform GPGPU kernels to increment profiling counter during their execution. This solution is fully compatible with the SPOC toolset. This means that it is compatible with either CUDA or OpenCL frameworks, as well as with very heterogeneous systems combining multiple different devices managed by any of these two frameworks.

Our future work includes improving the level of information provided in our tool. This can be done internally, by adding more counters. In particular, it would be interesting, in GPGPU kernels to have information relative to specific threads instead of concerning the overall kernel. In this matter we intend to provide the maximum and minimum of each thread counter (which can be very useful to exhibit specific and unexpected behaviors). To go further, we intend to provide developers with a way of specifying their own counters and only compute those.

As with SPOC and Sarek, this kind of solution is particularly adapted for generative high-level frameworks where most of the actual complex low-level code that runs on co-processors is generated at run-time.

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009 23, 187–198 (2009)
2. Bourgoin, M., Chailloux, E., Lamotte, J.L.: Efficient Abstractions for GPGPU Programming. International Journal of Parallel Programming pp. 1–18 (2013)
3. Bourgoin, M., Chailloux, E.: High-Level Accelerated Array Programming in the Web Browser. In: ACM SIGPLAN International Workshop on LIbraries, Languages and Compilers for Array Programming. pp. 31–36 (Jun 2015)
4. Bozman, Ç., Mauny, M., Le Fessant, F., Gazagnaire, T.: Profiling the Memory Usage of OCaml Applications without Changing its Behavior. In: OCaml 2013. Boston, United States (Sep 2013), `https://hal.inria.fr/hal-01095305`
5. Catanzaro, B., Garland, M., Keutzer, K.: Copperhead: Compiling an embedded data parallel language. SIGPLAN Not. 46(8), 47–56 (Feb 2011)
6. Chakravarty, M., Keller, G., Lee, S., McDonell, T., Grover, V.: 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)
7. Corporation, I.: Profiling OpenCL Applications with System Analyzer and Platform Analyzer (2014)
8. Donham, J., Pouillard, N.: Camlp4 and Template Haskell. In: ACM SIGPLAN Commercial Users of Functional Programming. pp. 6:1–6:1. ACM (2010)
9. Esterie, P., Falcou, J., Gaunard, M., Lapresté, J.T., Lacassagne, L.: The Numerical Template toolbox: A Modern C++ Design for Scientific Computing. Journal of Parallel and Distributed Computing 74(12), 3240–3253 (Jul 2014)
10. Farooqui, N., Kerr, A., Eisenhauer, G., Schwan, K., Yalamanchili, S.: Lynx: A Dynamic Instrumentation System for Data-parallel Applications on GPGPU Architectures. In: Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software. pp. 58–67. ISPASS '12, IEEE Computer Society (2012)
11. Farooqui, N., Rossbach, C.J., Yu, Y., Schwan, K.: Leo: A Profile-driven Dynamic Optimization Framework for GPU Applications. In: Proceedings of the 2014 International Conference on Timely Results in Operating Systems. pp. 5–5. TRIOS'14, USENIX Association (2014)
12. Gautier, T., Lima, J.V., Maillard, N., Raffin, B.: 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 (May 2013)
13. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml System Release 4.03 : Documentation and User's Manual. Tech. rep., Inria (Apr 2016), `http://caml.inria.fr`
14. Masliah, I., Baboulin, M., Falcou, J.: Metaprogramming dense linear algebra solvers. Applications to multi and many-core architectures. In: 13th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2015) (Aug 2015)

15. NVIDIA: CUDA Profiler Users Guide (Version 8.0) (2016)
16. Nystrom, N., White, D., Das, K.: Firepile: Run-time Compilation for GPUs in Scala. In: Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering. pp. 107–116. GPCE '11, ACM (2011)
17. Purnomo, B., Rubin, N., Houston, M.: ATI Stream Profiler: A Tool to Optimize an OpenCL Kernel on ATI Radeon GPUs. In: ACM SIGGRAPH 2010 Posters. pp. 54:1–54:1. SIGGRAPH '10, ACM, New York, NY, USA (2010)
18. Rossbach, C.J., Yu, Y., Currey, J., Martin, J.P., Fetterly, D.: Dandelion: A Compiler and Runtime for Heterogeneous Systems,. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. pp. 49–68. SOSP '13, ACM (2013)
19. Rubinsteyn, A., Hielscher, E., Weinman, N., Shasha, D.: Parakeet: A Just-In-Time Parallel Accelerator for Python. In: The 4th USENIX Workshop on Hot Topics in Parallelism. USENIX (2012)