Automatic Parallelization of a Gap Model using Java and OpenCL

Jonathan Passerat-Palmbach, Arthur Forest, Julien Pal, Bruno Corbara, David R.C. Hill

UMR 6158 CNRS

Automatic Parallelization of a Gap Model using Java and OpenCL

Jonathan Passerat-Palmbach*† ‡ § ,
Arthur Forest †,
Julien Pal †,
Bruno Corbara ¶ ‖ ,
David R.C. Hill † ‡ §

Originally published in: European Simulation and Modeling Conference
2012 — October 2012 — pp 24-31
ISBN: 978-90-77381-73-1

Abstract: *Nowadays, scientists are often disappointed by the outcome when parallelizing their simulations, in spite of all the tools at their disposal. They often invest much time and money, and do not obtain the expected speed-up. This can come from many factors going from a wrong parallel architecture choice to a model that simply does not present the criteria to be a good candidate for parallelization. However, when parallelization is successful, the reduced execution time can open new research perspectives, and allow to explore larger sets of parameters of a given simulation model. Thus, it is worth investing some time and workforce to figure out whether an algorithm is a good candidate to parallelization. Automatic parallelization tools can be of great help when trying to identify these properties. In this paper, we apply an automatic parallelization approach combining Java and OpenCL on an existing Gap Model. The two technologies are linked with a library from AMD called Aparapi. The latter allowed us to study the behavior of our automatically parallelized model on 10 different platforms, without modifying the source code.*

Keywords: Automatic Parallelization; Simulation; Java; OpenCL; Aparapi; Manycore

* This author's PhD is partially funded by the Auvergne Regional Council and the FEDER
† ISIMA, Institut Supérieur d'Informatique, de Modélisation et de ses Appplications, BP 10125, F-63173 AUBIÈRE
‡ Clermont Université, Université Blaise Pascal, LIMOS, BP 10448, F-63000 CLERMONT-FERRAND
§ CNRS, UMR 6158, LIMOS, F-63173 AUBIÈRE
¶ Clermont Université, Université Blaise Pascal, LMGE, BP 10448, F-63000 Clermont-Ferrand
‖ CNRS, UMR 6023, LMGE, F-63177 AUBIÈRE

RESEARCH CENTRE
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France

1 INTRODUCTION

At the manycore era, the keyword is parallelization. While designing parallel applications from scratch can be quite tricky, parallelizing sequential applications often involves a large refactoring. Moreover, the awaited speed-up is not magically obtained. Depending on the parallel-likeness of the application, results can be very disappointing. Then, the question is: how much time shall we invest on trying to parallelize a given application?

Many works have tackled automatic parallelization over the past few years. Consequently, a number of tools are now available to help developers harness parallel cores horsepower. Depending on your favorite programming language and underlying platform, a large set of tools is often available.

The questions of the hardware accelerator can also be complicated to address, since different devices will support different software technologies. However, since 2008, the OpenCL standard [Khronos OpenCL Working Group, 2011] is trying to fill the gap between parallel platforms by offering a single language and a multi-platform support. The programming language supports C99 constructs and is compiled on-the-fly at runtime to exploit a wide range of devices, from CPU to GPU and FPGAs.

Still, the OpenCL highly verbose API (Application Programming Interface) makes it not very attractive to a wide range of developers. Hopefully, high-level libraries can be used to hide this complexity, and even automatically transform high-level code into OpenCL. Moreover, such libraries permit a better integration of OpenCL into applications already developed in C++ or Java, for example.

In this paper, we study the benefits of parallelizing part of a Gap Model simulation written in Java using OpenCL. To do so, we generate OpenCL code through Aparapi, an AMD Java library automatically producing parallel OpenCL code from a raw Java source. This process will help us determine whether such an automatic parallelization approach is relevant when dealing with legacy simulation models. Along the following pages, we will successively:

- Describe our gap model and its aims;
- Present automatic parallelization tools such as OpenCL and Aparapi;
- Detail the implementation of the parallelization of our gap model;
- Comment the results obtained thanks to this automatic parallelization approach.

2 GAP MODEL

The work in this paper relies on a gap model described in this section. Gap models study the dynamics of forests, and particularly the trees that fall, and progressively regrow over the years. Considering a precise square area of the forest, we will look at the gap dynamics (their appearance and progressive disappearance). Gaps are the "empty" areas formed by fallen trees and the neighbors they have carried away while collapsing. Gap models are among the most widely used in forest modeling. While studies tackling forest dynamics are legion [Acevedo et al., 1995, Chave, 1999, Bugmann, 2001, Gourlet-Fleury et al., 2004], for this simulation we have chosen to stick with a simplified gap model, with emphasis on the resulting light distribution. The model

supporting this work is intended to become the base of a more ambitious multi-scale multi-agent simulation model, this is why we focused on parallelizing it, so that it does not slow down the future simulation.

The simulation model that we aim at implementing represents the long term dynamics of Ant Gardens (AGs) and of some of their inhabitants in a tropical rainforest. An AG is a complex arboreal suspended structure including an ant nest and symbiotic plants growing on it. In French Guiana, AGs are initiated by different species of ants that incorporate the plant seeds in the humus-rich carton of their nest [Orivel and Leroy, 2011]. AGs are installed on a more or less sun-exposed site depending on the light preferences of ant species. Among the plants growing on AGs is a water-holding one (a tank-plant from the bromeliad family) that harbours various aquatic organisms from microorganisms to vertebrates (batrachian tadpoles) among which many insect larvae. The tanks of these bromeliads harbour different communities depending on the ant species inhabiting their resident AG, partly due to canopy openness and resulting incident light [Céréghino et al., 2010].

The whole AG model will help us study the consequences of human activities on the forest. Indeed, when Man builds a road, the latter creates a huge edge to a degree comparable to an artificial linear gap. The edge effect that results is very important. Man encourages the development of species accustomed to this type of environment. The anthropic action may favor particular ants and therefore particular aquatic insect larvae. Many studies (see the survey in [Kitching, 2000]) have shown that ecosystems that develop in bromeliads are home to many mosquito larvae. In French Guiana, some species of mosquitoes are vectors of "dengue" (due to arboviruses; some as dengue haemorrhagic fever are deadly if left untreated). The potential danger of such uncontrolled proliferation is evident, which explains the value of studying such a configuration by simulation.

Now, let us consider the gap model at the heart of this study. According to data provided by domain specialists, we know that in some studied area in French Guiana, 33 gaps per year appear over a $300m^2$ area, on average. This size can range from $20m^2$ to $20000m^2$, and it takes 20 to 25 years for a gap to structurally close completely, and this according to an exponential decay law. Indeed, in the early years, many young seedlings are taking advantage of the sunlight reaching the ground, and gradually as the trees grow, they close the gap surface, and reduce the amount of light reaching the ground. Finally, according to measurements made on two sites, 1.1% of forest area falls every year on the first site and 1.3% on the second. This means that statistically, half the forest area is affected by gaps in 69 years, and nearly all its surface (99%) is in 400 years.

In addition to the gaps dynamics, in our model we had to consider the intensity of the incident light over time, as it is directly related to the area exposed by gaps. The light model is recomputed annually, as it evolves with the gaps. To represent the illumination at a point, we must take into account both the direct exposure when it belongs to a gap, but also the gradient of light scattered from different points of the simulated area. A UML class diagram of our model is presented in Figure 1.

From a more technical point of view, our gap model is developed in Java. This programming language enables us to run the simulation on any platform. We can also take advantage of several powerful third-party libraries available in Java, such as *JFreeChart* for instance, to display live statistics when running the simulation. Figure 2 shows off a screenshot of the graphical user interface displaying the evolution of the forest over the simulation time.

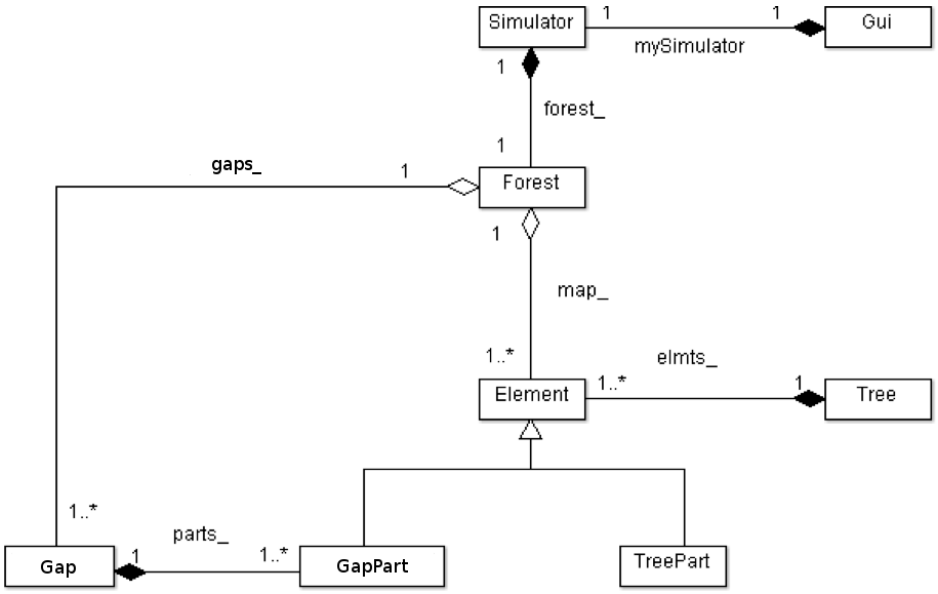


Figure 1: UML class diagram of the gap model

3 AUTOMATIC PARALLELIZATION

3.1 OpenCL

OpenCL is a standard proposed by the Khronos group that aims at unifying developments on various kinds of hardware accelerators like CPUs, GPUs and FPGAs. It provides programming constructs based upon C99 for writing the actual parallel code (called the kernel). Kernels are executed by several work-items that will be mapped onto different execution units depending on the target: for instance, GPUs will associate them to local threads. For scheduling purposes, work-items are then bundled into work-groups each containing an identical amount of work-items.

Kernels are enhanced by APIs (Application Programming Interface) used to control the device and the execution. At the time of writing, the latest version of the API is 1.2 [Khronos OpenCL Working Group, 2011] and was released in November, 2011. The execution of OpenCL programs relies on specific drivers issued by the manufacturer of the hardware they run on. The point is OpenCL kernels are not compiled with the rest of the application, but on-the-fly at runtime. This allows for specific tuning of the binary for the current platform.

On the other hand, OpenCL is a constrained and complicated API. OpenCL often leads to problems like bloated source code or risky constructs. Indeed, when kernel functions that execute on the hardware accelerator remain concise and thus fully expressive, host API routines result in verbose source code where it is difficult for an external reader or for a non-regular developer to determine the purpose of the application. On top of that, verbose constructs discourage developers from checking the results of each of their calls to the API. At the same time, the OpenCL API is very permissive with regards to the type of the parameters its functions accept. If a developer mistakenly swaps two parameters in an API call, it is very likely that the application will remain silent and not throw any error, unless the developer has explicitly taken care of properly handling the error codes this call returns. In a heterogeneous environment such as

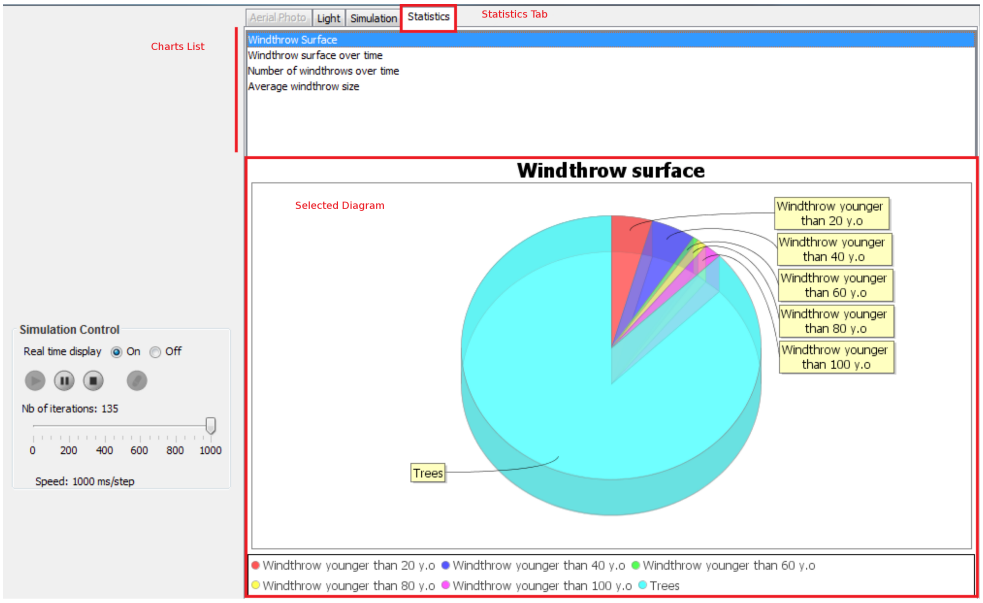


Figure 2: Screenshot of the statistics tab of the simulation's GUI

OpenCL, where the parallel part of the computation will be relocated on hardware accelerators, runtime errors that only issue error codes are not the easiest bugs to get rid of.

As a conclusion, although the OpenCL standard is a great tool offering portability in high performance computing, it does not meet the higher level APIs expectations awaited by parallel application developers to help them avoid common mistakes, and to produce efficient applications in a reasonable lapse of time. That is why we strongly encourage the use of third-party APIs to generate OpenCL code.

3.2 Automatic parallelization in Java

Several interesting tools running on top of the Java platform enable developers to easily take advantage of OpenCL in their application. Among the most relevant, we can cite JavaCL [Chafik, 2011a], ScalaCL [Chafik, 2011b] and Aparapi [AMD, 2011]. In this work, we have chosen to employ the latter: Aparapi. Because of its ability to transform Java code into OpenCL code, we judge it more suitable for a smooth integration within an already existing Java simulation. In this respect, a proof of concept was published by an AMD software engineer to demonstrate the features of Aparapi [Joshi, 2012]. The interested reader can refer to [Passerat-Palmbach and Hill, 2013] for a more complete survey detailing automatic parallelization toolkits related to OpenCL, and particularly JavaCL and ScalaCL.

At the time of writing, automatic parallelization approaches cannot be considered as magical tools that will make the most of any software at no cost. Especially when dealing with a cross-platform tool such as OpenCL, maximum performance cannot be achieved without any intervention from the developer. In our case, we consider automatic parallelization strategies as helpers to build prototypes to figure out the parallel-likeness of the application. Furthermore, when using OpenCL, the prototype can also help us determine which platform will be the best at parallelizing our simulation.

3.3 Aparapi

Aparapi [AMD, 2011] is a project initiated by AMD and recently freed under an MIT-like open source license. It intends providing a way to perform OpenCL actions directly from a pure Java source code. This process is entirely carried out at runtime and involves no upstream translation step. To do so, Aparapi relies on a Java Native Interface (JNI) wrapper of the OpenCL API that hides complexity to developers. Basically, Aparapi proposes implementing the operations to be performed in parallel within a *Kernel* class. The kernel code takes place in an overridden *run()* abstract method of the *Kernel* class that sets the boundaries of the parallel computation. At the time of writing, only a subset of Java features are supported by Aparapi, which means that *run()* can only contain a restricted amount of features, data types, and mechanisms. Concretely, primitive data types, except *char*, are the sole data elements that can be manipulated within Aparapi's kernels.

The process is set off when the *execute* method is called on a *Kernel* instance. Having converted the Java source code to OpenCL, Aparapi tries to set up the classical OpenCL tool-chain. Globally, it consists in reading the source, compiling, building the whole program and finally enqueuing the kernel to be executed on the target device. If this process fails, Aparapi falls the execution back to a Java Thread Pool (JTP) running on the CPU.

This short description reflects the expectations we had when sketching the identy kit of the tool we wanted to employ for building our automatically parallelized prototype. In fact, apart from the *run* method that must be provided, nothing more is expected from us. In the worst case, the OpenCL conversion will not collapse and the parallelism will be limited to the capabilities of the host CPU to execute parallel threads from a pool of workers.

4 PARALLELIZATION USING APARAPI

4.1 Profiling

Prior to any parallelization attempt, it is a good point to use a profiler in order to determine which part of the simulation model is the most time consuming. Indeed, due to the complexity of our model, it would have been difficult to design a fully parallel prototype. Thus, we needed a profiling step to figure out what were the critical parts of the model. We used the Netbeans profiler to obtain this information. Figure 3 shows the output of this tool on our model.

The screenshot reveals that the hot spot in our model is a method called *setBoundaries*, which is called by every instance of *Gap*, several times at each iteration. Thus, tackling the parallelization of this method that represents 70.5% of the global execution time should speed up the whole simulation. The problem now is to find an appropriate way to parallelize this method while ensuring its output remains the same and taking advantage of the parallel architectures at our disposal.

4.2 Implementation

Now that we have identified both the part of the algorithm to be parallelized and the tool that will be used to do so, let us describe the new way to set the boundaries of gaps in parallel.

The sequential version of *setBoundaries* is gap-based, and is called successively by every gap on the map. Let us recall that Aparapi cannot handle types other than primitives yet. Thus,

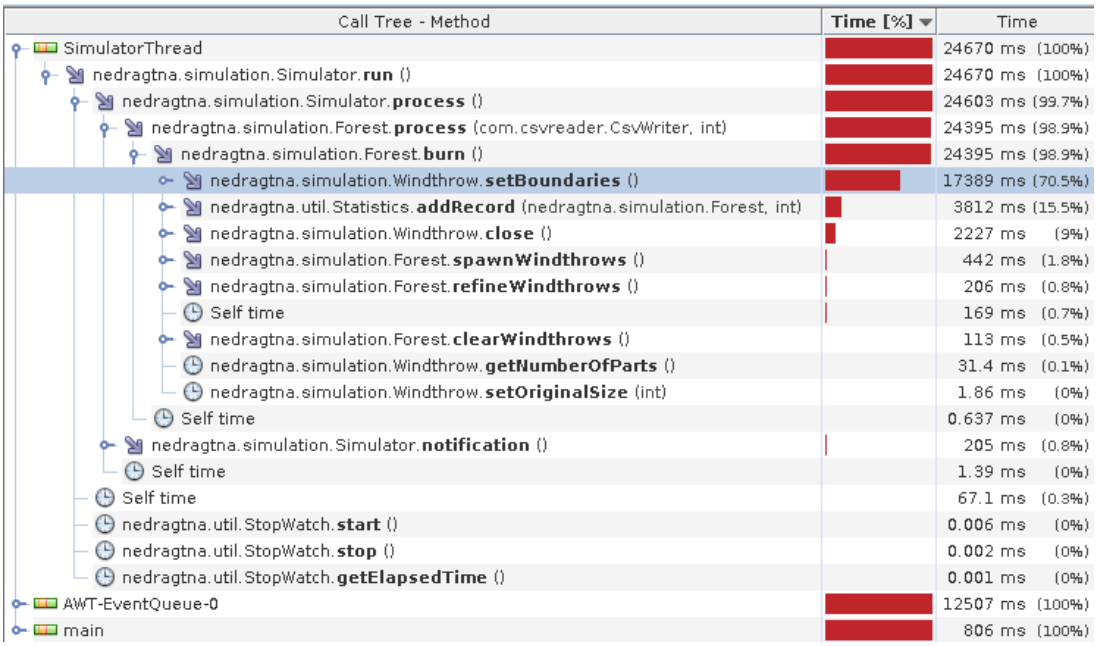


Figure 3: Netbeans Profiler output after 200 iterations of the sequential Gap Model

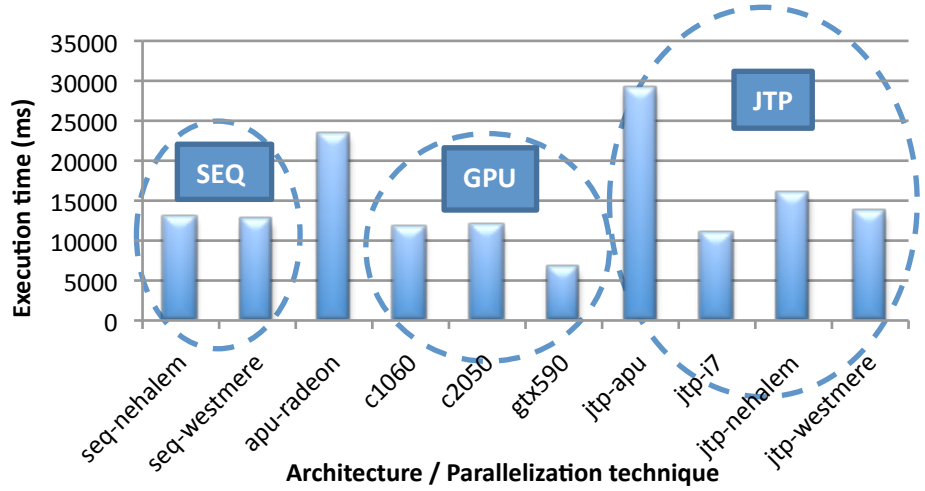
the *Gap* class or its *Forest* container cannot be directly used in the parallel version. The most efficient parallelizations are hardly those that perfectly match the original algorithm. Often, one has to rethink his algorithm in order to make it parallel.

We have chosen to represent the whole *Forest* as a map of boolean cells. A cell that contains part of a gap bears the *true* value, whereas all the other cells are set to *false*. This boolean representation allows us to turn the original algorithm containing lots of nested branches into a boolean expression that involves no branch at all. This way, we enable our parallel code to run faster when deployed on a GPU platform. In fact, the programming model of GPUs does not cope with heavy branching, and the resulting OpenCL code could lead to dramatic performance issues that would not reflect the actual computing capabilities of GPUs.

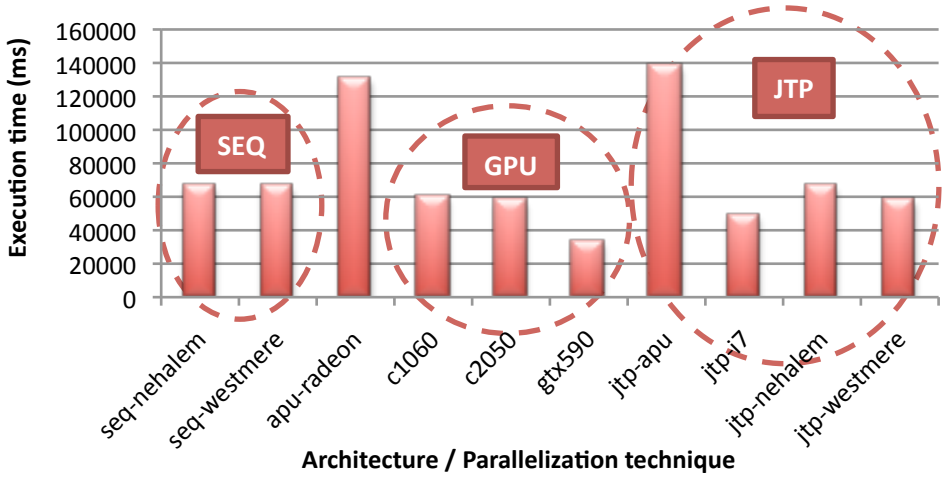
Along with this spatial parallelism, a computing element is assigned to each cell. It will translate differently following the underlying platform they run on. For instance, OpenCL-enabled GPUs will treat them as logical threads whereas they will be tasks when falling back to a JTP execution.

5 RESULTS

In this section, we compare the execution time of our gap model on various architectures. Actually, there are two main parameters that can lead to different performance for a given algorithm: changing the underlying platform and the size of the data to process. In our case, we compare no less than 10 different platforms, from the original sequential execution to OpenCL declinations generated automatically by Aparapi. We also consider the capability of these approaches to



(a) Small map (300,000 cells)



(b) Large map (1,500,000 cells)

Figure 4: Execution time for 10 different platforms running the simulation on a different map sizes

scale with the data by feeding the model with two different maps: a small one containing around 300,000 cells, and a large one that is about five times as big.

The first thing to notice in view of these two charts is that the size of the input data does not impact the parallelization techniques: the most efficient with a small map remain the same when dealing with a large map. It means that the automatic parallelization approach that we set up using Aparapi scales well with data.

Now regarding the platforms, three groups distinguish on both Figures 4(a) and 4(b): the sequential executions, the OpenCL-enabled GPU ones, and finally the OpenCL-disabled JTP executions. We studied sequential executions on two successive generations of Intel server processors: the cutting-edge Westmere and its predecessor, the Nehalem. The execution time on

these two kinds of architectures is roughly the same. The second group is formed by NVIDIA GPU platforms. Here again, we tested several generations and kinds of architectures: a Tesla C1060, a Tesla C2050 and a GeForce GTX 590. The two latter GPUs belong to a more recent architecture codenamed Fermi. Surprisingly, the less powerful GTX outperforms its two counterparts specifically designed to process High Performance Computing jobs. This is due to the host machine in which these devices are embedded. The Tesla GPUs are contained in machines with ECC-RAM memory, the latter kind of memory involves a noticeable overhead when accessing data. Given that our algorithm transfers the whole map at every simulation step, the two Tesla GPUs suffer from their host's weakness, in spite of their overwhelming computational power. Last but not least, the JTP runs are quite homogeneous on all the architectures but the AMD APU. As a matter of fact, this processor is a combination of two rather slow elements: the CPU part is far behind our Intel cores in terms of performance, in the same way that the embedded Radeon GPU cannot stand the comparison with NVIDIA products.

As a conclusion, the GPU parallelizations distinguish as the most efficient of our benchmark. They are consequently the ones that we will investigate further to parallelize our Gap Model. As long as all the efficient GPUs belong to the NVIDIA family of processors, we could even consider switching to a CUDA implementation that would make the most of these hardware accelerators, and offer maximum speed-up to our simulation.

A slight limitation must be noted when looking at these results. In this work, we have run the same parallel algorithm on every hardware architecture at our disposal. However, some platforms might be more efficient with a coarser parallelization grain. JTPs for instance would have fewer tasks to schedule on their worker threads, but this is out of the scope of this study, where we are looking for the best environment to run a given parallelized algorithm.

6 PERSPECTIVES

Although the Aparapi solution is efficient, it is not very convenient to use. Indeed, it still implies some makeshift developments to be integrated in Java applications. First of all, the way it is shipped is not fully satisfying because it needs additional work from the user to be inserted in a Maven build, for example.

Then, the major issue when integrating Aparapi in a Java development chain is that it can only handle primitives Java types. In our case, we have provided methods to convert the objects running our gap model to low-level primitive arrays. If Aparapi does not provide an automatic boxed-to-primitive type converter in its upcoming versions, we will create one in our future work. To do so, we plan to take advantage of the Java Reflection API, and especially of the *Javassist* library [Chiba, 1998] that enables defining classes at runtime.

We are confident enough in the success of this further development, since OpenCL is able to handle basic Object-Oriented features, such as encapsulation, as we would do it in C. Thus, it should be possible to generate Java code at runtime that maps high-level classes to basic constructions that can be understood by OpenCL.

On the pure performance point of view, we have presented our work as a prototype to figure out whether parallelization was suited for our simulation. Now that preliminary results have shown satisfying enough figures, we can spend more time on optimizing the parallel code for the architecture that showed the most promising results. To do so, we have slightly modified the

behavior of Aparapi so that it systematically outputs the resulting OpenCL code. This way, we can modify this code to benefit of some hardware dedicated optimizations. While being a cross-platform tool, OpenCL allows developers to harness the specificities of underlying platforms through a mechanism called *extensions*. In our case, it would be interesting to determine whether hardware-specific optimizations can increase the execution speed of our simulation again.

By doing so, we would however disable Aparapi in our application, since it is unable to read user-written kernels. Still, other Java libraries allow developers to integrate OpenCL kernels at no cost in their applications. We particularly think of JavaCL [Chafik, 2011a], which produces Java wrapper classes at compile time to launch OpenCL kernels from a Java source code.

7 CONCLUSION

In this paper, we have presented an automatic parallelization approach using OpenCL, applied to a gap model written in Java.

In order to pair Java and OpenCL, we have chosen a free library provided by AMD, called Aparapi, which automatically transformed our sequential Java code into parallel OpenCL code. This approach has shown to be satisfying enough to design a parallel prototype of our simulation. As we have seen in the results section, automatic parallelization does not allow to leverage the most of parallel architectures, but it gives precious hints about whether an application is worth being parallelized or not. In this way, OpenCL is a great tool combined to automatic parallelization because it allows developers to test various kinds of architectures without changing their code at all.

Automatic parallelization allowed us to get rid of a hot spot that used to slow down the simulation. Now that we have obtained these preliminary results, not only the simulation was sped up due a partial parallelization, but we are now able to target the architecture that appeared as the most efficient, in our case: the NVIDIA GPUs. Further development will consequently focus on leveraging this particular platform, from the parallel OpenCL code that we extracted thanks to our contribution to the Aparapi library.

ACKNOWLEDGMENTS

The authors would like to thank Yannick Le Pennec for his careful reading and useful suggestions on the draft. This work was partly funded by the Auvergne Regional Council.

References

[Acevedo et al., 1995] Acevedo, M., Urban, D., and Ablan, M. (1995). Transition and gap models of forest dynamics. *Ecological Applications*, 5(4):1040–1055.

[AMD, 2011] AMD (2011). Aparapi. `http://developer.amd.com/zones/java/aparapi/Pages/default.aspx`.

[Bugmann, 2001] Bugmann, H. (2001). A review of forest gap models. *Climatic Change*, 51:259–305.

[Céréghino et al., 2010] Céréghino, R., Leroy, C., Dejean, A., and Corbara, B. (2010). Ants mediate the structure of phytotelm communities in an ant-garden bromeliad. *Ecology*, 91(5):1549–1556.

[Chafik, 2011a] Chafik, O. (2011a). Javacl. `http://code.google.com/p/javacl/`.

[Chafik, 2011b] Chafik, O. (2011b). Scalacl. `http://code.google.com/p/scalacl/`.

[Chave, 1999] Chave, J. (1999). Study of structural, successional and spatial patterns in tropical rain forests using troll, a spatially explicit forest model. *Ecological Modelling*, 124:233–254.

[Chiba, 1998] Chiba, S. (1998). Javassist—a reflection-based programming wizard for java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, page 174.

[Gourlet-Fleury et al., 2004] Gourlet-Fleury, S., Cornu, G., Dessard, H., Picard, N., and Sist, P. (2004). Modelling forest dynamics for practical management purposes. *Bois et Forêts des Tropiques*, 280(2):41–52.

[Joshi, 2012] Joshi, S. (2012). Leveraging aparapi to help improve financial java application performance. Technical report, AMD.

[Khronos OpenCL Working Group, 2011] Khronos OpenCL Working Group (2011). The opencl specification 1.2. Specification 1.2, Khronos Group.

[Kitching, 2000] Kitching, R. (2000). *Food webs and container habitats: the natural history and ecology of phytotelmata*. Cambridge University Press.

[Orivel and Leroy, 2011] Orivel, J. and Leroy, C. (2011). The diversity and ecology of ant gardens (hymenoptera: Formicidae; spermatophyta: Angiospermae). *Myrmecological News*, 14:73–85.

[Passerat-Palmbach and Hill, 2013] Passerat-Palmbach, J. and Hill, D. (2013). Opencl: A suitable solution to simplify and unify high performance computing developments. ISSN 1759-3158.

8 BIOGRAPHY

Jonathan Passerat-Palmbach holds a French Diplôme d'Ingénieur, equivalent to a Master Degree in Computer Science from the ISIMA Computer Science & Modelling Institute. He is currently a PhD student at the LIMOS (ISIMA) - UMR CNRS 6158 of Blaise Pascal University of Clermont-Ferrand (France). His work is focused on high performance computing tools and stochastic discrete events simulation with manycore systems.

Arthur Forest & Julien Pal are two undergraduate students from ISIMA who highly contributed to the development of the simulation model in Java.

Bruno Corbara holds a PhD in Biology of Behavior from University Paris XIII. He is currently an Assistant Professor at Blaise Pascal University of Clermont-Ferrand (France) and researcher at the LMGE-UMR CNRS 6023. His work is mainly focused on tropical ecology (particularly in forest canopies), behavioral ecology of social insects (ants and wasps), ant-plant interactions and ecology of bromeliad tanks, with a recurrent interest in multi-agent models.

David R.C. Hill is currently Vice President and Professor at Blaise Pascal University. Since 1990, Professor Hill has authored or co-authored many technical papers and journal papers and

he has published various textbooks including recent free e-books from Blaise Pascal University Press.

UMR 6158 CNRS

RESEARCH CENTRE
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France

Publisher
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France
http://limos.isima.fr/