J Supercomput (2011) 58:349–356
DOI 10.1007/s11227-011-0591-7

Automatic code generation for GPUs in llc

Ruyman Reyes · Francisco de Sande

Published online: 15 March 2011

Abstract `llc` is a C-based language where parallelism is expressed using compiler directives. In this paper, we present a new backend of an `llc` compiler that produces code for GPUs. We have also implemented a software architecture that eases the development of new backends. Our design represents an intermediate layer between a high-level parallel language and different hardware architectures.

We evaluate our development by comparing the OpenMP and `llc` parallelizations of three different algorithms. In every case, the probable performance loss with respect to a direct CUDA implementation is clearly compensated by a significantly smaller development effort.

Keywords GPU computing · CUDA · llc · OpenMP · Compiler

1 Introduction

The use of graphic processors (GPUs) [4] in High Performance Computing (HPC) has burst on the market as a new alternative for exploiting parallelism in many applications [7]. For some problems, GPU computing is a low-cost alternative that in many cases delivers results similar to or even better than those achieved with traditional hardware. Given the larger market for GPU technology, its effect on the HPC market is sure to be long-lasting.

From our point of view, the greatest obstacle to the adoption of GPU computing by end users of HPC is the lack of programmability of GPUs.

R. Reyes · F. de Sande (✉)
Departamento de Estadística, I.O. y Computación, Universidad de La Laguna, Avda. Astrofísico F. Sánchez, s/n, 38271 La Laguna, Tenerife, Canary Islands, Spain
e-mail: fsande@ull.es

R. Reyes
e-mail: rreyes@ull.es

In this paper, we address the problem of automatic code generation for GPUs from high-level languages. `llc` [3] is a parallel language where parallelism is expressed through the use of compiler directives that adhere to the OpenMP syntax. `llCoMP`, the `llc` compiler, is a source-to-source compiler that translates C code annotated with `llc` directives into high-level parallel code. For the time being, `llCoMP` has two different backends: one producing hybrid MPI+OpenMP code, and the new backend generating CUDA code.

The performance of the MPI and hybrid MPI+OpenMP code generated by `llCoMP` has been studied in previous papers [3, 8]. In this article, we focus our attention on the study of the new `llCoMP` CUDA backend. With our approach, the performance loss with respect to a direct CUDA implementation is clearly offset by a significantly smaller development effort.

The rest of this article is organized as follows. In Sect. 2, we expose the motivation for our work. The main ideas behind the translation performed by `llCoMP` are given in Sect. 3. In Sect. 4, we discuss some of the optimizations currently implemented in the new compiler backend. The experimental evaluation of the translation produced by the compiler is presented in Sect. 5. Finally, we offer some concluding remarks and propose future lines of work in Sect. 6.

2 Motivation

At the current time, HPC technology is witnessing some very rapid changes. The end of the Gigahertz race has expanded the number of computer architectures that are capable of achieving high performance. These profound changes in the hardware world are immediately followed by corresponding shifts at the software level. New tools and languages are clearly needed if we are to take advantage of the new hardware capabilities.

In the last decade, we have seen a proliferation of HPC-specific languages and tools, promoted not only by governments but also by academia and business. All of them aim to offer the maximum performance with the least programming effort. New languages hide architectural constrains and provide a highly expressive syntax, allowing parallelism to be expressed accurately.

It is well known that the introduction of a new language has two main drawbacks: HPC users need to learn a new programming language, and they cannot reuse their previous codes without some effort. An alternative approach consists of extending a widely known language (usually C or Fortran) by adding a minimum amount of constructs to exploit parallelism. One of the most successful instances of this approach is OpenMP. It was designed as a shared memory programming standard and has yielded great performance for these systems.

Another effect of the changes in the hardware layer has been the increase in heterogeneity in HPC architectures [1]. This new situation has partially left behind OpenMP and most other languages. Almost none of the current OpenMP implementations consider heterogeneous systems, or even support specific computational devices.

Each of these additional computational devices has its own programming interface and model. If we consider FPGA as an example, we observe that despite its increasing popularity, there is no common programming API for it, and the programmer

needs to develop specific code for each device and for each function that she wants to implement.

At the time of this writing, the OpenCL [5] standard represents an effort to create a common programming interface for heterogeneous devices, which many manufacturers have adopted. However, it is still in its infancy, and its programming model is not simple.

CUDA [6] is a more mature and widespread approach, although currently it only supports NVIDIA devices. CUDA offers a programming interface, mostly C with a small set of extensions. This framework allows HPC users to re-implement their codes using GPU devices. Although it is somewhat simple to build code using this framework, achieving a good performance rate is difficult and requires a huge coding and optimization effort to obtain the maximum performance from the architecture.

There is a division between the users who have a need for HPC techniques and the experts who design and develop the languages since, in general, the users do not have the skills necessary to exploit the tools involved in the development of the parallel applications. Any effort to narrow the gap between users and tools by providing higher level programming languages and increasing their simplicity of use is thus welcome.

In recent years, we have been working on a project that aims to combine simplicity for the user with reasonable performance and flexibility to migrate code across different architectures.

We propose to the HPC programmer a simple and well-known language that hides the hardware complexity with an OpenMP-like syntax. To the architecture designer, we present templates, representing the most common parallel patterns, where she can introduce optimized versions without too much effort. Our framework connects language and patterns, producing efficient parallel code for different architectures.

As an example of this idea, the code in Listing 1 is an implementation in `llc` of a Molecular Dynamics (MD) simulation. It employs an iterative numerical procedure to obtain an approximate solution whose accuracy is determined by the time step of the simulation.

The `llc` code is compiled by `llCoMP`, the `llc` compiler-translator, which produces an efficient high-level parallel code. Since all OpenMP directives and clauses are recognized by `llCoMP`, we have four versions with the same code: sequential, OpenMP, `llc`/MPI, and `llc`/MPI+OpenMP. We need only choose the proper compiler to obtain an executable for the desired platform.

3 The translation process

Nowadays, parallel architectures are changing so fast that we need a flexible framework that allows us to reuse our work when going from one architecture backend to another. In order to accomplish this task, we decided to use Python, due to its friendly syntax and modularity capabilities.

Reusing code from open source projects, we were quickly able to build a C frontend supporting OpenMP. The class hierarchy we designed allowed us to develop the CUDA backend in only a couple of months.

The code generation in `llCoMP` (like in the former version of the compiler) uses the *code pattern* concept. A *code pattern* is an abstraction that represents a specific

```
1     ...
2     #pragma omp parallel for default(shared) private(rij, d)
         reduction(+ : pot, kin)
3     #pragma llc result(f[i], nd)
4       for (i = 0; i < np; i++) {     /* Pot. energy and forces */
5         for (j = 0; j < nd; j++)
6           f[i][j] = 0.0;
7         for (j = 0; j < np; j++) {
8           if (i != j) {
9             d = dist(nd, box, pos[i], pos[j], rij);
10            pot = pot + 0.5 * v(d);
11            for (k = 0; k < nd; k++) {
12              f[i][k] = f[i][k] - rij[k] * dv(d) /d;
13            }
14          }
15        }
16        kin = kin + dotr8(nd, vel[i], vel[i]); /* kin. energy */
17      }
18    ...
```

Listing 1 Main loop of a molecular dynamic code simulation in `llc`

task in the context of the translation. `llCoMP` uses two kinds of code patterns: static and dynamic. The simplest code patterns are implemented using code templates, while the most complex cases require the implementation of a *Mutator* (a Python class capable of transforming the Internal Representation, IR).

A code template is a code fragment in the target language that will be modified according to certain input parameters. This code is interpreted and translated to the IR, after which it is grafted onto the Abstract Syntax Tree. The design of the backend using code templates will ease the implementation of new future backends.

Whenever we need to use a device, we can identify several common tasks: initialization, local data allocation, device invocation, data retrieval, and memory deallocation, among others. Each of these tasks identifies a pattern and each pattern is implemented through a code template. To manipulate these code templates and insert them in the IR, `llCoMP` defines a set of operations that are collected in a library and exhibit a common facade.

4 Code optimizations

In our first effort at automatic code generation for CUDA, we focused on simplicity rather than on code optimization. However, we detected some situations where improvements in the target code will enhance performance. We are currently working on the implementation of these improvements and other complex optimizations that will be included in future releases of `llCoMP`.

One optimization already included in the current version of the translator is the use of a specialized kernel to perform reduction operations. With a small effort, we improved the performance of codes that make use of reductions, providing the user

```
1    ...
2    #pragma omp parallel shared(uold, u, ...)  private(i, j, resid)
3    {
4      #pragma omp for
5      for (i = 0; i < m; i++)
6        for (j = 0; j < n; j++)
7          uold[i][j] = u[i][j];
8      #pragma omp for reduction(+:error )
9      for (i = 0; i < (m - 2); i++)
10       for (j = 0; j < (n - 2); j++)
11         resid = ...
12           ...
13         error += resid * resid;
14   }
15   ...
```

Listing 2 Main loop of a molecular dynamic code simulation in `llc`

with a specialized kernel that implements a parallel prefix reduction, similar to that presented in the CUDA toolkit.

Another key issue to enhance the performance in a CUDA architecture is the reduction in data transfer rates between host and device. Our experiments show that this transfer rate barely exceeds 1.7 GB/s in our environment, constituting a critical bottleneck.

In our strategy, at the end of each parallel loop we synchronize the memory of host and device (according to the computational model [2] underlying the `llc` implementation). Let us consider the code in Listing 2, which is part of the implementation of the Jacobi iterative method both in `llc` and OpenMP, as taken from the OpenMP official website. Without applying any optimization, the translation of the parallel loops in lines 4 and 8 to CUDA leads to communications between the CPU and GPU at the beginning and end of each loop. However, this behavior introduces unnecessary communications since memory positions have not been modified between the end of the first loop and the beginning of the second.

To avoid this overhead, our compiler injects the communications at the beginning and end of parallel regions. Inside the parallel region, we assume that memory locations allocated in the host remain unchanged.

5 Computational results

This paper represents a preliminary evaluation of the results obtained with the new backend of the `llc` compiler. In order to evaluate the performance of the `llCoMP` translation, we used three algorithms: the Mandelbrot set computation, a Molecular Dynamic simulation, and the solution of a finite difference equation using the Jacobi iterative method. All three source codes share the same implementation in OpenMP and `llc` (no specific `llc` annotations were used). In all cases the figures compare the speedup of the pure OpenMP implementation with 8 cores against the CUDA code

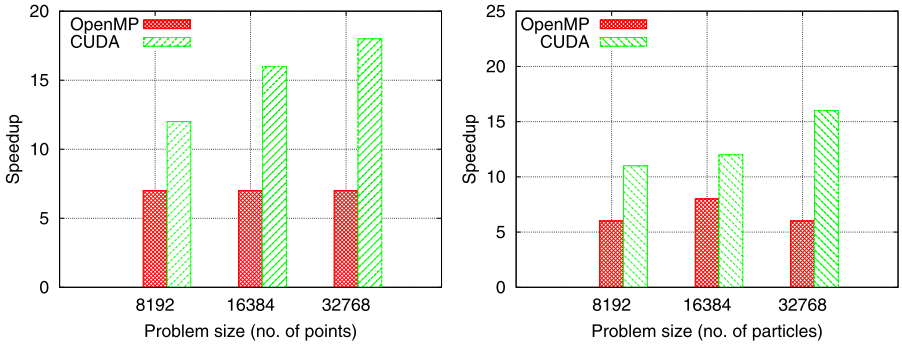

Fig. 1 Speedup obtained for three increasing problem sizes in the Mandelbrot set (*left*) and MD simulation (*right*)

generated by `llCoMP` using 64 threads per multiprocessor. The number of threads was selected by varying this number over the course of several experiments in order to keep that number which maximizes performance. The speedups are computed using exactly the same source code for the `llCoMP` and OpenMP versions. The sequential code was obtained by deactivating the OpenMP flags. Neither OpenMP, CUDA, nor serial versions of the code required additional libraries or compilation flags, apart from the usual ones: `-O3` for serial, `-O3 -fopenmp` for OpenMP and `-O3 -arch=1.3` for CUDA.

The computational experience was carried out on a system built from two AMD Opteron QuadCore processors (8 cores) with 4 GB of RAM. This system was attached through a PCI-express 16x bus, a Tesla C1060 card with 4 GB and 1 GPU with 240 cores.

For the Mandelbrot set computation (Fig. 1, left), by combining the reduction explained in Sect. 4 with the optimization of data transfers between CPUs and the device, we managed to improve the performance of the code. 2% of this improvement corresponds to the specialized reduction kernel.

In the case of MD (Fig. 1, right), the size of the problem represents the number of particles involved in the simulation. For this test, the speedup of the CUDA code also grows with the problem size, while the OpenMP version does not show a regular behavior. This is probably due to memory constraints. The MD code has a predefined number of iterations (ten in our experiments), and each iteration involves calls to two different functions that have been parallelized independently. One of these two functions has a parallel loop similar to that seen in Listing 2, while the other function is a simple matrix update. This scenario is not the most beneficial for our approach since it produces an unnecessary amount of communications.

The iterative loop of the Jacobi method code was shown in Listing 2 and the corresponding results are presented in Fig. 2. The size of the problem corresponds to the dimension of the square matrices used in the computation. This code benefits from the optimization explained in Sect. 4, which minimizes the communications between host and device in the case of several parallel for-loops inside the same parallel region. Again, while the OpenMP speedup remains almost constant when the problem

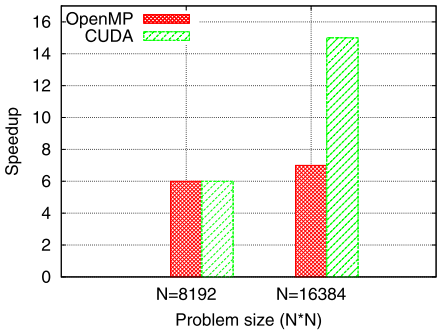

Fig. 2 Speedup of the Jacobi iterative method. Due to memory constrains, the size of the problem could not be increased. The relationship between computation and memory transfer is too low with $N = 8192$, but increasing granularity to $N = 16384$ improves performance

size is doubled, the CUDA implementation takes advantage of the larger amount of data involved.

6 Conclusions and future work

We have presented the results obtained with the new implementation of the CUDA backend for the `llc` compiler. Taking into account the smaller effort to develop codes using `llc` as compared with direct CUDA implementations, we conclude that `llc` is appropriate for implementing some classes of parallel applications. Our current implementation aims to ease the development of CUDA codes for non-expert end users. We aim to target scientific codes where most of the computation takes part in one or more loops compatible with the OpenMP 'or' construct.

We have the first version of a source-to-source compiler, written in a modern, flexible, and portable language that represents a starting point for future efforts. We believe that the development of the new compiler backend is a first milestone in our path to the future version of the `llc` language.

With the experience gained in the production of the CUDA backend, we believe that the incorporation of new target languages (OpenCL, for example) should not require an unaffordable effort. From this point forward, our goal is to continue developing the language in order to increase its capabilities.

We are continuing to work in this area, in particular:

- To increase the number of algorithms parallelized using our compiler, with particular attention to commercial applications
- To study and implement additional compiler optimizations that will enhance the performance of the target code
- To study the generation of hybrid CUDA + OpenMP code

Acknowledgements The authors wish to thank the anonymous reviewers for their suggestions on how to improve the paper.

This work has been partially supported by the EU (FEDER), the Spanish MEC (Plan Nacional de I+D+I, contract TIN2008-06570-C04-03), and the Canary Islands Government (ACIISI, contracts SolSubC200801000285 and SolSubC200801000307).

References

1. Brodtkorb AR, Dyken C, Hagen TR, Hjelmervik JM, Storaasli OO (2010) State-of-the-art in heterogeneous computing. Sci Program 18:1–33
2. Dorta AJ, González JA, Rodríguez C, de Sande F (2003) `llc`: a parallel skeletal language. Parallel Process Lett 13(3):437–448
3. Dorta AJ, López P, de Sande F (2006) Basic skeletons in `llc`. Parallel Comput 32(7–8):491–506
4. Fatahalian K, Houston M (2008) A closer look at GPUs. Commun ACM 51(10):50–57
5. Khronos Group (2009) The OpenCL specification, version 1.0, online. http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf
6. Nickolls J, Buck I, Garland M, Skadron K (2008) Scalable parallel programming with CUDA. Queue 6(2):40–53
7. Owens JD, Luebke D et al (2007) A survey of general-purpose computation on graphics hardware. Comput Graph Forum 26(1):80–113
8. Reyes R, Dorta AJ, Almeida F, de Sande F (2009) Automatic hybrid MPI+OpenMP code generation with `llc`. In: Ropo M, Westerholm J, Dongarra J (eds) 16th Euro PVM/MPI UGM, Espoo, Finland. LNCS, vol 5759. Springer, Berlin, pp 185–195