Automatic Source Code Transformation for GPUs Based on Program Comprehension

Pasquale Cantiello and Beniamino Di Martino

Second University of Naples, Italy
{pasquale.cantiello,beniamino.dimartino}@unina.it

Abstract. In this work is presented a technique to transform sequential source code to execute it on parallel architectures as heterogeneous many-core systems or GPUs. Source code is parsed and basic algorithmic concepts are discovered from it in order to feed a knowledge base. A reasoner, by consulting algorithmic rules, can compose this basic concepts to pinpoint code regions representing a known algorithm. This code can be annotated and / or transformed with a source-to-source process. A prototype tool has been built and tested on a case study to analyse the source code of a matrix multiplication. After recognition of the algorithm, the code is modified with calls to nVIDIA GPU cuBLAS library.

Keywords: comprehension, GPU, manycore, source-to-source, reengineering.

1 Introduction

The development of software for scientific applications through the years has seen different seasons. Continuous growth in performance requests to fulfil specific calculus needs, drove the birth of parallel machines and related concurrent programming models. Lot of effort has been spent on developing parallelization techniques to port applications on parallel, vectorial or super-scalar architectures in the ninety.

Subsequently, continuous improvements on the hardware systems and mainly on clock processors' clock speeds, caused lacking of interest on research activities on parallelization, since the performances of applications got a *natural growth*.

But in the last few years the processors' clock speed growth has stopped due to physical limits on junctions dimensions and to the dissipated power. Processor improvements have to follow a different path by multiplying the number of processing units on a chip (multi-core systems). Chips producers nowadays announce systems, no longer with higher frequencies, but with increased number of cores. Special purpose devices as the GPUs, designed for graphics applications, can be used to do parallel computations. Not only in systems for scientific applications, but also in common personal computers, there are now multi-core CPUs and GPUs.

It is hard to write parallel code and it requires skilled developers. Great effort is needed to port existing software and manual parallelization of applications with high orders of magnitude of lines of code is a critical and error-prone process.

Nowadays, research activities on automatic parallelization systems to migrate existing code, in a more or less assisted way, to heterogeneous architectures, are again *hot-topics.* Next years of compiler research will be mostly devoted to the generation and verification of parallel programs[12].

In this paper we will see how to pinpoint potentially parallelizable code regions from source code, starting from extracting basic algorithmic concepts, composing them and reasoning upon them to find common algorithms implementations. The outlined regions of code are processed with a source-to-source transformation technique in order to take gain of the target architecture. If an optimized library exists for the specific algorithm, the code is replaced with a function call.

The paper continues, after this introduction, with the section 2 with an analysis of related works on code comprehension and translation. In the section 3 will be introduced the technique to analyse code to find concepts, to reason on them and to translate related code for heterogeneous architectures.

The first implementation of a tool to drive the process and translate code to CUDA[17] code or CUBLAS calls is presented in section 4. A case study to validate the technique is shown in section 5, and the paper ends with section 6 with conclusions and future work directions.

2 Related Works

A description of the Algorithmic Recognition used here can be found in [7] or in [5] where is introduced the definition of *algorithmic concept* and the technique to describe the algorithms by using an attributed grammar. Two tools to do program comprehensions were presented in [6]. That work was tailored to be used on the *Vienna Fortran Compiler* [3]. Present work, even if is still using the same recognition engine, it is different since adds the source-to-source capabilities for GPUs, adds support for object oriented languages as C++, and new source code parsing technique.

Several papers have been presented on *clone detection* or searching for similar code fragments in programs. These clones are potentially fault causes due to the necessity to maintain multiple copies of the same code. The works in this field have been developed with several approaches. Text-based approaches [8], syntax-based [2], or graph-based [14]. All of these techniques can only detect duplicates that are nearly identical each other and so cannot identify implementation variants or *code perturbations.* They focus mainly on code that originates from the copy actions (e.g. cut and paste), instead of investigating on the functionality of the code. A semantic approach can be found in [10] where the authors investigate on extracting sub-graphs from Program Dependence Graph (PDG) [9], convert them to a simpler tree form and do tree similarities studies with a scalable method [13].

Source to source transformation for parallelizzation can be found in several works. In [16] the semantics of standard abstractions as (C++ Standard Template Library), or Array-Based Computation Loop drives the process of source code transformation with the support of OpenMP directives and clauses. In [15] the transformation has been made on OpenMP code by analizing parallel constructs and work-sharing constructs to extract candidate kernels and transform them to CUDA code. A framework for optimization of affine loop nests, with polyhedral compiler model, has been described in [1]. All these works start with code that is already parallel or user annotated code to drive the transformation and not from sequential code as ours.

3 Source Code Transformation

The proposed technique, shown in figure 1, begins with a static analysis of code. The source file is processed by a front-end that translates it into an intermediate representation (IR) which is an enriched Abstract Syntax Tree (AST). The Extractor traverse this structure searching for patterns that can be recognized as *basic concepts* and emits Prolog Facts corresponding to them. Now the Transformer can submit queries to the reasoner to search for known algorithmic concepts. The reasoner, by using these facts and by consulting a set of rules, gives replies about any found algorithm. Answers include references to the code region related to the algorithm and to the data involved. The transformer can now pick from the repository an alternative implementation of the algorithm that is suitable for the target architecture, and modifies the AST accordingly. The user can interact in this phase by setting preferences on the selection of the alternatives. A final unparsing of the IR can generate new source code for the target architecture.

3.1 Algorithm Recognition

The recognition strategy is based on a hierarchical parsing. Starting from the intermediate representation of code, basic concepts are recognized. They can be seen as building blocks of *composed concepts* in a recursive way as described in [7] and in [5].

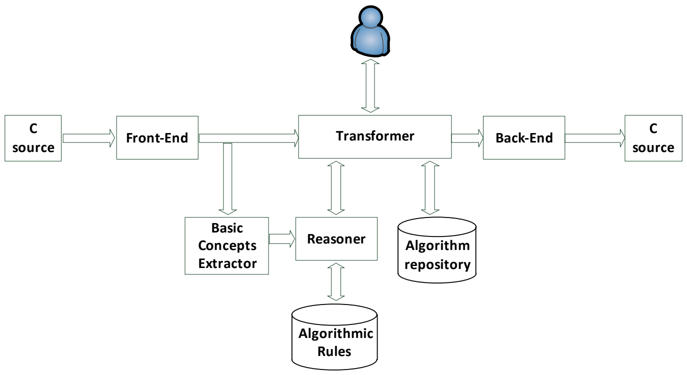

Fig. 1. The model of the process

For example the analysis of the statement: `int i = 0;` produces the facts in the listing 1.1.

```
scalar_var_def(i, def_list_1, elem_update_r, main).
scalar_var_inst(stp_1, i, elem_update_r, main).
val_inst(stp_2, 0, elem_update_r, main).
assign_r((def_list_1, stp_1), stp_1, stp_2, elem_update_r,
    main).
```

Listing 1.1. Facts produced by variable declaration

We can see that four facts are generated: a) the definition of a scalar variable; b) the usage of a scalar value; c) the usage of a constant; d) the assignment statement. In detail, the second fact above indicates a basic concept named `scalar_var_inst`. Its instance number is 1 (`stp_1`), its parameter is `i` (the variable name), the rule is recognized by is the `elem_update_r` and the function in which it is present is named `main`.

Similarly, in last fact, we see the composition of the previous concepts in a tree.

Another example is the loop statement: `for (i = 0; i < 10; i++)` which produces the facts in listing 1.2

```
for_r(15,for(15,exit_115),init_6,exit_115,incr_7,elem_update_r
    ,main).
scalar_var_inst(stp_11,i,elem_update_r,main).
val_inst(stp_12,0,elem_update_r,main).
assign_r(init_6,assign(init_6,stp_11),stp_11,stp_12,
    elem_update_r,main).
scalar_var_inst(stp_13,i,elem_update_r,main).
val_inst(stp_14,10,elem_update_r,main).
less(exit_115,stp_13,stp_14,elem_update_r,main).
scalar_var_inst(stp_15,i,elem_update_r,main).
post_incr(incr_7,stp_15,elem_update_r,main).
```

Listing 1.2. Facts produced by for loop

In this case the numbers are the pointers to the nodes of the AST.

Control dependence facts generated have a syntax like:
`control_dep(dependant_id, depend_from_id, type, class, method).`

Data dependence facts have a syntax like:
`data_dep(type,dependant_id,depend_from_id,variable,class,method).`

The *concept recognition rules* are the production rules of the parsing process; they describe the feature set that permits the identification of an instance of an algorithmic concept in the code. This feature set can be named *algorithmic pattern*. The rules can be defined as the way in which abstract concepts, as groups of statements in the code, are organized under an *abstract control structure*. With this definition we include structural relationships as *Control* and *Data Flow*, *Control* and *Data Dependence* and function calling.

Each recognition rule identifies the concept by using a composition hierarchy, specified with the set of composing sub-concepts, and a set of conditions and constraints that sub-concepts must satisfy.

The main aspects of the method are:

– Basic concepts, the starting points of the hierarchical abstraction process, are chosen among the elements of the intermediate representation of code. Properties and relationships that characterize them are still part of the representation [4]. Dependence informations are found in the PDG that is built during the analysis phase.
– Properties and relationships are chosen in order to privilege the structural features instead of syntax. So, dependences relationships assume a main role: they become the features that drive the *abstract control structure* among the concepts.

The chosen parsing strategy is *top-down*; so the concept parsing is descendant and the recognition is *demand-driven.* This choice has been motivated by our main objective that is the transformation of code to support parallelization of certain algorithms. We do not want to comprehend entire code, but only find if instances of particular algorithmic patterns are present. So the demand-driven approach can be used, since it has a less complex search space than the code-driven approach.

A knowledge base with the definition of recursive composition rules permits to describe an algorithm. It relies on Prolog as a system shell and takes advantage of Prolog's deductive inference-rule engine to perform the hierarchical concept recognition.

The Prolog engine is queried for specific goals. When one of them is satisfied, the result contains the recognized algorithm, the references to AST nodes involved, and the input and output data related.

After reasoning, composed concepts are recognized. Some examples of recognized concepts are:

– `elem_update`: This represents the update of the value of an element by an expression that depends on previous value of the same element.
– `count_loop`: This represents a `for` loop where the *init*, *test* and *output* statements are based on expressions involving only constants, except the loop variable.
– `scan`: This is the access (read or write) of a sequence of elements in an array.
– `dot_product`: This is concept of the product of two dimensions of two arrays.

3.2 Source to Source Transformation

The information obtained after the recognition of the algorithm drives the transformer module. The source code region that implements the algorithm can be replaced by optimized parallel code or by call to optimized libraries. The algorithm repository contains, for each target architecture, one or more possible implementations, stored in a parametric source code format. The parameters should be mapped to input and output data involved in the algorithm. The user can drive the selection of the code among the repository by setting preferences on alternative implementations. The transformer directly manipulates the intermediate representation of the analysed source program. By using the references given by the reasoner, the abstract syntax tree is modified with the following steps:

– The sub-tree corresponding to the code region is pruned from the AST and, if desired, a comment block with the original code is inserted.
– A new sub-tree is generated with transformed code. If needed (as in GPUs), it contains also: memory allocation on device, memory transfer from CPU to device, library invocation, memory transfer from the device back to the CPU and memory deallocation.
– This tree is appended in the AST at the removal point, just after the comment block.

After all the transformations done on the AST, an *unparsing* operation permits to generate the code ready to be compiled on the target platform.

4 Prototype Tool

To test the technique a prototype tool has been built. The reasoner has been implemented with SWI-Prolog [19] as a stand-alone module with a shell interface.

The rest of the work has been done by using Rose Compiler [18]. This is a complete compiler infrastructure, tailored for source-to-source transformations. It uses two front-end modules, one that can parse C/C++ and the other for Fortran 2003 and earlier. The intermediate representation used by Rose is very reach and preserve all the information from the source code (including source file references, code comments, macros and templates for C++). This can be valuable in the unparsing process to produce source code that can still be readable by humans. The programming interface of Rose Compiler is C++, so our work was done in this language.

Starting from the intermediate representation obtained by the front-end, the AST should be traversed in order to find basic concepts. We built a class that implements the Visitor Design Pattern [11] by extending the `ROSE_VisitorPattern` class and overriding the `visit()` methods for each node type we need to process. The AST is so traversed and the series of facts corresponding to the basic concepts are produced in a text file. Similarly control-dependence and data-dependence facts are produced by using the related Rose Library functions. Now the reasoner is invoked with a series of goals each corresponding to a known algorithm that is present in the repository. If a goal is satisfied the reasoner replies with the name of the algorithm, the references to the code region that implements it and the data involved. Since multiple queries can be done to search for different algorithms and the reasoning is a time consumption process it can be done separately from the transforming and results saved in intermediate files.

The transformer, starting with that information, cuts the original code (or simply enclosed in comments, depending on the preferences of the user), builds new code from the templates for the platform the user has chosen, and modify the AST accordingly. But before doing the code removal, a test to prove legality of the transformation is done. All the code that is enclosed in the AST sub-tree of the algorithm, but is not mapped to basic concepts of the algorithm (eg. extra added lines), is checked for data dependencies with the data involved in the algorithm.

To add new code and comments to the AST, Rose Compiler furnishes the so called *Rewrite mechanism*. It uses three simple functions: `insert()`, `replace()` and `remove()` that can be used at different levels of abstractions. Two low levels which interact directly with the nodes of the tree and permit a fine grained control on the generated nodes but they are extremely verbose, an intermediate level which lets the user express the transformation with strings and an higher level which can be used during the traversal operations. We have used the mid level since it gave use the best compromise between complexity and power of use.

After all the transformations, a final call to `backend()` function can be used to generate the source code from the AST in a new file.

5 Case Study

As a case study we used the source code for a sequential C implementation of a matrix-matrix multiplication. This contains one of the algorithms we can recognize at present.

In listing 1.3 we can see a fragment of the code that is given in input to the tool.

```
double x[10][10];
double y[10][10];
double z[10][10];
double temp = 0;
int i = 0;
int j = 0;
int k = 0;

for(i=0;i<10;i++) {
    for(j=0;j<10;j++) {
        temp = 0;
        for(k=0;k<10;k++) {
            temp = temp + x[i][k] * y[k][j];
        }
        z[i][j] = temp;
    }
}
```

Listing 1.3. Sequential Matrix multiplication

In listing 1.4 is shown a small excerpt of the Prolog facts with basic concepts and dependence information produced for the code.

```
array_var_definition(def_list_1 ,double,2,x,[10,10],simple_mmp,
    do_mmp).
array_var_definition(def_list_2 ,double,2,y,[10,10],simple_mmp,
    do_mmp).
array_var_definition(def_list_3 ,double,2,z,[10,10],simple_mmp,
    do_mmp).
scalar_var_def(i, def_list_4 ,simple_mmp,do_mmp).
scalar_var_inst(stp_1 ,i,simple_mmp,do_mmp).
val_inst(stp_2 ,0,simple_mmp,do_mmp).
% .... omitted
control_dep(17,19,true,simple_mmp,do_mmp).
control_dep(15,17,true,simple_mmp,do_mmp).
control_dep(100011,17,true,simple_mmp,do_mmp).
control_dep(100014,15,true,simple_mmp,do_mmp).
%
data_dep(true,100014,100011,z,0, simple_mmp,do_mmp).
```

Listing 1.4. Prolog Facts produced

In listing 1.5 we can see the response of the reasoner for a query of the goal `matrix_matrix_r`.

```
% hierarchy of concepts: references omitted
matrix_matrix_product(
    simple_scan(...),
    matrix_vector_product(
        simple_scan(...),
        dot_product(...),
        simple_scan(...),
        ),
    simple_scan(...)
).
```

Listing 1.5. Prolog hierarchy response for matrix_matrix_r goal

After that recognition, in listing 1.6 is shown the added source code with the calls to CUBLAS library, assuming the user has chosen that implementation. We have omitted the commented code block.

```
// .... omitted commented code ...
// ---> Added by Transformer ---
void* _dptr_x;
void* _dptr_y;
void* _dptr_z;
// Memory allocation
cudaMalloc ((void **)&_dptr_x, 10*10*sizeof(double));
cudaMalloc ((void **)&_dptr_y, 10*10*sizeof(double));
cudaMalloc ((void **)&_dptr_z, 10*10*sizeof(double));
cublasCreate(&handle );
// Data transfer CPU->GPU
```

```
cublasSetMatrix (10, 10, sizeof(double), x, 10, _dptr_x, 10);
cublasSetMatrix (10, 10, sizeof(double), y, 10, _dptr_y, 10);
// Matrix x Matrix Multiplication
cublasDgemm (handle, CUBLAS_OP_N, CUBLAS_OP_N, 10, 10, 10,
    0.0, _dptr_x, 10, _dptr_y, 10, 0.0, _dptr_z, 10);
// Data transfer GPU->CPU
cublasGetMatrix (10, 10, sizeof(double), _dptr_z, 10, z, 10);
// Memory deallocation
cublasDestroy (handle);
cublasFree (_dptr_x);
cublasFree (_dptr_x);
cublasFree (_dptr_x);
```

Listing 1.6. Code region added for Matrix multiplication with CUBLAS

6 Conclusion

In this work we have seen how to do source code analysis in order to recognize basic algorithmic concepts, to reason on them and drive a source to source transformation of code so that it can execute on new parallel architectures as GPUs. A prototype tool has been presented to validate the technique and a test on a case study has been shown.

The work must be intended as a starting point for future investigation. At present the rules can recognize basic linear algebra algorithms as matrix and vector multiplication, dot product, maximum and minimum search, reduction. One direction on which we are now working is the extension of the set of recognized algorithms and their implementation variants (i.e. variants with use of pointers and dynamic memory allocation). At the same time, since the reasoning is a time-consuming process, the recognition process does not scale well with the increasing in the number of recognized algorithms. We are studying techniques to finding code clones that maybe can be adapted to extract basic concepts.

Another research path is to add performance investigation on the transformed code; at present the transformation is done with no regards on the size of the problem. We know that for small problems, the overhead added by memory transfers can vanish the improvements obtained by the use of the parallel device. Conversely, large problems may not fit the device memory. We are working on adding test points on code so that they can be used to select, at runtime, different implementation variants depending on the size of the data involved. In this direction, an extension of the transformation to produce OpenCL code can be used to tailor heterogeneous architectures as many-core sytems.

References

1. Baskaran, M.M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: A compiler framework for optimization of affine loop nests for gpgpus. In: Proceedings of the 22nd Annual International Conference on Supercomputing, ICS 2008, pp. 225–234. ACM, New York (2008)

2. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: IEEE International Conference on Maintenance (ICSM 1998), p. 368 (1998)
3. Benkner, S.: Vfc: The vienna fortran compiler. Scientific Programming 7, 67–81 (1999)
4. Di Martino, B.: Algorithmic concept recognition to support high performance code reengineering. Special Issue on Hardware/Software Support for High Performance Scientific and Engineering Computing of IEICE Transaction on Information and Systems E87-D, 1743–1750 (2004)
5. Di Martino, B., Iannello, G.: Pap recognizer: A tool for automatic recognition of parallelizable patterns. In: International Workshop on Program Comprehension, p. 164 (1996)
6. Di Martino, B., Kessler, C.W.: Two program comprehension tools for automatic parallelization. IEEE Concurrency 8, 37–47 (2000)
7. Di Martino, B., Zima, H.P.: Support of automatic parallelization with concept comprehension. Journal of Systems Architecture 45(6-7), 427–439 (1999)
8. Ducasse, S., Rieger, M., Demeyer, S.: A language independent approach for detecting duplicated code. In: IEEE International Conference on Software Maintenance (ICSM 1999), p. 109 (1999)
9. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems (TOPLAS) 9(3) (1987)
10. Gabel, M., Jiang, L., Su, Z.: Scalable detection of semantic clones. In: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, pp. 321–330. ACM, New York (2008)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
12. Hall, M., Padua, D., Pingali, K.: Compiler research: the next 50 years. Communications of the ACM 52, 60–67 (2009)
13. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: Deckard: Scalable and accurate tree-based detection of code clones. In: Proceedings of the 29th International Conference on Software Engineering, ICSE 2007, pp. 96–105. IEEE Computer Society, Washington, DC (2007)
14. Komondoor, R., Horwitz, S.: Using Slicing to Identify Duplication in Source Code. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 40–56. Springer, Heidelberg (2001)
15. Lee, S., Min, S.-J., Eigenmann, R.: Openmp to gpgpu: a compiler framework for automatic translation and optimization. SIGPLAN Not. 44, 101–110 (2009)
16. Liao, C., Quinlan, D., Willcock, J., Panas, T.: Extending Automatic Parallelization to Optimize High-Level Abstractions for Multicore. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 28–41. Springer, Heidelberg (2009)
17. NVIDIA. Cuda: Compute unified device architecture, `http://www.nvidia.com/cuda/`
18. Quinlan, D.: Rose compiler, `http://www.rosecompiler.org/`
19. Wielemaker, J.: Swi-prolog, `http://www.swi-prolog.org/`