Technical Report CLIP1/2015.0

Towards a Rule-Based Approach to Generate High-Performance Scientific Code*

Guillermo Vigueras[1], Salvador Tamarit[2], Manuel Carro[1,2], and Julio Mariño[2]
{guillermo.vigueras,manuel.carro}@imdea.org
{salvador.tamarit,julio.marino}@upm.es

[1] IMDEA Software Institute, Campus de Montegancedo
28223 Pozuelo de Alarcón, Madrid, Spain
[2] Universidad Politécnica de Madrid, Campus de Montegancedo
28660 Boadilla del Monte, Madrid, Spain

Abstract. Obtaining good performance when programming heterogeneous computing platforms (including multi-core computers) poses significant challenges for the programmer. We present an approach where architecture-agnostic scientific code with semantic annotations is transformed into a functionally equivalent one better suited for a given platform. The transformation steps are formalized (and implemented) as rules which can be fired when certain syntactic and semantic conditions are met. Rule firing is guided by heuristics which aim at capturing how behavioral run-time characteristics (e.g., resource consumption) are affected by the transformation steps. Program properties, to be matched with rule conditions, can sometimes be automatically inferred or, alternatively, stated by hand with pragmas. A tool based on these ideas is under implementation.

Keywords: High-Performance Computing, Scientific Computing, Heterogeneous Platforms, Rule-Based Program Transformation.

1 Introduction

There is currently a strong trend in high-performance computing towards the integration of various types of computing elements: GPUs being used for non-graphical purposes, vector processors, FPGA modules, etc. As each of these components is specially suited for some class of computations, this is proving to be a cost-effective alternative to more traditional supercomputing architectures. However, this specialization comes at the price of additional complexity in hardware and, notably, software. Developers must take care of very different features to make the most of the underlying computing infrastructure. Programming these systems is restricted to a few experts, which hinders its widespread adoption and increases the likelihood of bugs. Also, portability is greatly limited. For these reasons it is crucial to develop programming models that ease the task of efficiently programming heterogeneous systems. In this context, we propose a framework for the sound, mechanical transformation of programs written in an architecture-agnostic way to versions suitable for heterogeneous platforms.

* Work partially funded by EU FP7-ICT-2013.3.4 project ref. 610686 *POLCA*, by Comunidad de Madrid project S2013/ICE-2731 *N-Greens Software* and by Spanish MINECO Projects TIN2012-39391-C04-03 and TIN2012-39391-C04-04 (*StrongSoft).*

We have focused on scientific code, on one hand, because heterogeneous computing is highly applicable in this domain. This is due to the suitability of certain components to some classes of computations – e.g. GPUs and linear algebra – but also, to the high energy and time consumption of key scientific applications, an aspect where heterogeneous computing has been found to be advantageous [9]. On the other hand, most scientific applications rely on a large base of existing algorithms that must be ported to the new architectures in a way that gets the most out of their computational strengths, while avoiding pitfalls and bottlenecks, and preserving the meaning of the original code.

Our framework assumes that scientific code implements the computation of mathematical formulas and follows patterns rooted in that mathematical origin. This makes it possible to state mathematical properties associated to the code and use them to define a sound transformation process which parallels the manipulation of mathematical formulas. The underlying mathematical description is provided by the programmer as annotations in the input code. Since the programmer should be aware of the underlying mathematical foundation, this should be an easy task for her.

Rule-based program transformation is a large and fruitful area [17,18]. Our proposal has as distinguishing features the focus on scientific code and the aim to achieve efficiency on heterogeneous platforms using quantitative measures of non-functional, run-time properties to guide which transformations to apply. Also, there is a crucial distinction between systems that generate new code from the mathematical model of an implementation into a new model and then generating new code, and those which use mathematical properties to transform an existing code. The former (automatic code synthesis) has long been subject of research and usually generates underperforming code because of its generality. The latter usually requires that the initial code is in some "canonical" form. Our approach, based on chaining mathematically sound small-step code transformations, tries to avoid those problems. We briefly comment on related works in the following paragraphs, pointing out similarities and differences.

CodeBoost[2], built on top of Stratego-XT[16], performs domain-specific optimizations to C++ code following an approach similar in spirit to our proposal. User-defined rules specify domain-specific optimizations; code annotations are used as preconditions and inserted as postconditions during the rewriting process. Concept-based frameworks such as Simplicissimus[15] transform C++ based on user-provided algebraic properties. The rule application strategy can be guided by the cost of the resulting operation, although this is done at the expression level (and not at the statement level).

The transformation of C-like programs so as to optimize parallelism in its compilation into a FPGA is treated in Handel-C[8]. It is however focused on a synchronous language, and therefore some of its assumptions are not valid in more general settings. A completely different approach is to use linear algebra to transform the mathematical specification of concrete scientific algorithms [12,11,10]. Here, the starting point is a mathematical formula and, once the formula is transformed, code is generated for the resulting expression. However, the good acceleration factors over hand-tuned code shown happens only for those algorithms, and applying the ideas to other contexts – like the aforementioned reuse of legacy code – does not seem straightforward.

$s \not\mapsto l$	statements s do not write into location l: $l \notin \text{writes}(s)$
$s \not\leftarrow l$	statements s do not read the value in location l
$s_1 \not\mapsto s_2$	statements s_1 do not write into any location read by s_2
$s_1 \not\leftarrow s_2$	statements s_1 do not read from any location written by s_2
e pure	expression e is *pure*, i.e. does not have side effects nor writes any memory locations.
g distributes_over f	$\forall x, y, z.\ g(f(x, y), z) \approx f(g(x, z), g(y, z))$
l fresh	l is the location of a *fresh* identifier, i.e. does not clash with existing identifiers if introduced in a given program state.

Table 1. Some basic predicates used for the side conditions of transformation rules.

2 Sketch of the Approach

Our approach transforms code by means of small-step rules which come from decomposing higher-level mathematical transformations into simpler, self-contained, individually meaningful, provably correct code transformation steps (Table 4). These rules feature syntactical pattern matching and semantic conditions which determine when they can be safely fired. Then, given a program $\mathcal{P}$ and a rewriting rule r whose conditions are fulfilled, if $\mathcal{P} \xrightarrow{r} \mathcal{P}'$ then $[\![\mathcal{P}]\!] \equiv [\![\mathcal{P}']\!]$: the original and the transformed program are semantically equivalent, although they may differ in their non-functional behavior. For example, rule JOINASSIGNMENTS in Table 4 removes the statement `l = e_1` by propagating its effect to a later statement `l = e_2` after substituting `l` for `e_1` in `e_2`. In order to do this safely, it is required that no code `s_2` in between reads from or writes to `l` and `e_1`. Since the application of individual rules is sound, chaining them gives also a sound result: if $\mathcal{P} \xrightarrow{r_i}{}^{*} \mathcal{P}'$, for arbitrary rules r_i whose conditions are held when applied, then $[\![\mathcal{P}]\!] \equiv [\![\mathcal{P}']\!]$.

Therefore, a big-step transformation R mimicking a high-level transformation of the mathematical structure such that $\mathcal{P} \xrightarrow{R} \mathcal{P}'$ can be decomposed into small-step transformations $\mathcal{P} \xrightarrow{r_0} \mathcal{Q} \xrightarrow{r_1} \mathcal{W} \cdots \xrightarrow{r_n} \mathcal{P}'$ individually sound and applicable in isolation. This makes it possible to treat code (say $\mathcal{Q}$) which syntactically differs from the "starting point" $\mathcal{P}$ of a high-level transformation, either by applying r_1 to continue the transformation chain or applying r_0^{-1} (if it exists) to go back to $\mathcal{P}$ [7,6]. Since correct and functionally equivalent code can be written in many ways, the situation where code implementing a mathematical computation does not conform to a "standard pattern" is very common and should be treated.

Table 1 presents some of the semantic conditions necessary. Some of them are runtime properties which express read / write dependencies between statements., necessary to ensure the correctness of many transformations (see Table 4). Rules can also include a measure of its impact on *adequacy properties*. This measure helps in determining how applying a rule changes the non-functional, runtime behavior of the code. Whether this change is an improvement depends on the pursued goal (save real estate / reduce delay in an FPGA, increase data-parallel computation for a GPU, increase data locality in regular CPUs, and combinations thereof for hybrid architectures). For space reasons we cannot show detailed versions of these adequacy properties and how they are measured, but intuitive examples are presented at the end of Section 3.

property	*addition*	*external product*
commutativity	$\mathbf{u}+\mathbf{v}=\mathbf{v}+\mathbf{u}$	
associativity	$\mathbf{u}+(\mathbf{v}+\mathbf{w})=(\mathbf{u}+\mathbf{v})+\mathbf{w}$	$a(b\mathbf{w})=(ab)\mathbf{w}$
identity element	0	1
distributivity laws:		
external over vector addition: $a(\mathbf{u}+\mathbf{v})=a\mathbf{u}+a\mathbf{v}$		
external over scalar addition: $(a+b)\mathbf{v}=a\mathbf{v}+b\mathbf{v}$		

Table 2. Properties for vector space V over field K with $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w} \in V$ and a, $b \in K$.

The variation of some metric quantitatively approximates the advantages of applying a given rule, and it is used to decide which rules are candidate to be applied in a given configuration. While other systems use either built-in or user-defined [7,16,4,5] static strategies, they fall short to account for run-time characteristics of the resulting code. Depending on the relative importance of these characteristics for a target architecture and a goal (e.g., is it more important to avoid data or instruction cache misses?), different fitness functions can be used.

We expect the number of rules to be large and dependent on the application domain. For this reason, rules can be written by final users with a C-like language (inspired by CTT [3] and CML [8]) and dynamically loaded into our prototype.[3] As an example, Fig. 3 is the user-level rule corresponding to the JOINASSIGNMENTS transformation mentioned before.

3 Example: Optimization of Linear Algebra Operations

```
join_assignments {
 pattern: {
  cstmts(body0_1);
  cexpr(v1) = cexpr(val_v1);
  cstmts(body0_2);
  cexpr(v1) = cexpr(val_v2);
  cstmts(body0_3);
 }
 condition: {
  no_mod_use(cexpr(v1),cstmts(body0_2));
  no_mod(cstmts(body0_2),cexpr(val_v1));
  pure(cexpr(val_v1));
 }
 generate: {
  cstmts(body0_1);
  cstmts(body0_2);
  cexpr(v1) =
       subs(cexpr(val_v2),cexpr(v1),
       cexpr(val_v1));
  cstmts(body0_3);
 }
 metrics: {
  metric_1: delta_of_metric_1;
  metric_2: delta_of_metric_2;
  ...
 }
}
```

Fig. 1. Joining two assignments

We will show how some rules derived from a linear algebra identity can be used in the optimization of a simple code fragment. The use of a linear algebra example is motivated by its widespread usage in many scientific applications. We will first describe the underlying mathematical structure, operations, and associated properties and then we will apply then to transform our sample code.

A *vector space* [13] is a pair (V, K) made up of a field K of *scalars* and an Abelian group V of *vectors*: vectors can be added, scalars have both addition and multiplication, and vectors can be stretched by scalars using the so called *external product*, which must be doubly distributive. Table 2 summarizes some of the algebraic properties of vector spaces.

Table 3 shows, on the left, the computation of $\mathbf{c} = a\mathbf{v}+b\mathbf{v}$ and, on the right, the

[3] http://goo.gl/yuOFiE

$\mathbf{c} = a\mathbf{v} + b\mathbf{v}$	$\mathbf{c} = (a+b)\mathbf{v}$

`#pragma def is_vc_space(V,[c,v])`
`#pragma def vc_space(V,F,+,*)`
`#pragma def is_sc_field(F,[a,b])`
`#pragma def`
`  sc_field(F,float,+,-,*,/)`
 	
`float c[N], v[N], a, b;`
 	
`for(int i=0;i<N;i++)`
`  c[i] = a*v[i];`
 	
`for(int i=0;i<N;i++)`
`  c[i] += b*v[i];`
	`#pragma def is_vc_space(V,[c,v])`
	`#pragma def vc_space(V,F,+,*)`
	`#pragma def is_sc_field(F,[a,b])`
	`#pragma def`
	`  sc_field(F,float,+,-,*,/)`
	
	`float c[N], v[N], a, b;`
	
	`float k = a+b;`
	`for(int i=0;i<N;i++)`
	`  c[i] = k*v[i];`

Table 3. Loop fusion transformation enabled by algebraic properties of a vector space

$$\texttt{for}\ (l{=}e_{ini};rel(l,e_{end});mod(l))\ \{s_1\}\quad \texttt{for}\ (l{=}e_{ini};rel(l,e_{end});mod(l))\ \{s_2\} \Rightarrow \texttt{for}\ (l{=}e_{ini};rel(l,e_{end});mod(l))\ \{s_1;s_2\}$$
when $s_1 \not\mapsto s_2, e_{ini}, e_{end}, rel$ pure, $(s_1;s_2) \not\mapsto \{l, e_{ini}, e_{end}\}$, writes$(mod(l)) = \{l\}$
(FOR-LOOPFUSION)

$$l\ \texttt{+=}\ e; \Rightarrow l = l + e;$$
(SPLITADDITIONASSIGN)

$$s_1;\ l = e_1;\ s_2;\ l = e_2;\ s_3; \Rightarrow s_1;\ s_2;\ l = e_2[e_1/l];\ s_3;$$
when $s_2 \not\mapsto l, s_2 \not\dashv l, s_2 \not\mapsto e_1, e_1$ pure
(JOINASSIGNMENTS)

$$f(g(e_1,e_3),g(e_2,e_3)) \Rightarrow g(f(e_1,e_2),e_3)$$
when e_1, e_2, e_3 pure, g distributes_over f
(UNDODISTRIBUTE)

$$\texttt{for}\ (e_1;e_2;e_3)\ \{s_b\} \Rightarrow l = e_{inv};\ \texttt{for}\ (e_1;e_2;e_3)\ \{s_b[l/e_{inv}]\}$$
when l fresh, e_{inv} occurs in s_b, e_{inv} pure, $s_b \not\mapsto e_{inv}$
(LOOPINVARIANTCODEMOTION)

Table 4. Some of the source code transformations used in the example.

computation of the equivalent expression $\mathbf{c} = (a+b)\mathbf{v}$. The code is annotated by the user stating that variables `v` and `c` belong to a vector space and variables `a` and `b` belong to the field of real numbers. These annotations are internally translated into properties of the program which are matched against rule conditions. Basic mathematical structures such as vector spaces and scalar fields are internally known to the tool. We plan to allow final users to define additional structures based on previously existing ones by giving the properties of their components. Note that if vectors and scalars were not arrays and floats, but other objects implemented through an abstract data type, annotations stating which varibles are elements of a vector space and which are its operations would enable the same transformations.

The rules we present have been studied in the literature [1,14] or are common knowledge. The rule FOR-LOOPFUSION performs a loop fusion, resulting in the code in Listing 1.1. As mentioned before, we estimate the impact of a rule by means of

Listing 1.1. Loop fusion.

```
for(i=0;i<N;i++) {
  c[i] = a*v[i];
  c[i] += b*v[i];
}
```

Listing 1.2. Expansion of compound assignment.

```
for(i=0;i<N;i++) {
  c[i] = a*v[i];
  c[i] = c[i] + b*v[i];
}
```

Listing 1.3. Variable substitution.

```
for(i=0;i<N;i++)
  c[i] =
      a*v[i]+b*v[i];
```

Listing 1.4. Distributive property.

```
for(i=0;i<N;i++)
  c[i] = (a + b) * v[i];
```

Listing 1.5. Loop invariant motion.

```
float k = a + b;
for(i=0;i<N;i++)
  c[i] = k * v[i];
```

Fig. 2. Transformation steps.

metrics that estimate how changes of the code influence its adequacy properties. This particular transformation allows the compiler to schedule instructions more efficiently since the body of the fussed loop becomes larger; reduces the overhead caused by having duplicated loop control statements; improves cache locality by making previously accessed data (`c[i]` and `v[i]`) in the two separated loops available in a single body, thus increasing task granularity and giving a compiler for regular CPUs more opportunities to exploit instruction-level parallelism [14].[4] On the negative side, it would restrict the possibility of parallel execution and thus it may not be selected when the target architecture is a multiprocessor.

Rule SPLITADDITIONASSIGN (producing code 1.2) does not affect non-functional properties but enables the application of other rules. Rule JOINASSIGNMENTS (code 1.3) eliminates dependencies among statements and reduces register pressure. Rule UNDODISTRIBUTE (code 1.4) reduces the number of arithmetic operations. Finally, rule LOOPINVARIANTCODEMOTION (code 1.5) moves a common computation for each loop iteration out of the loop, reducing the number of arithmetic operations performed. However, if registers are scarce and the expression moved is inexpensive to compute, code motion may actually deoptimize the code, since register spills will be introduced in the loop.

As mentioned in Section 2, if the initial code to be transformed does not match the initial step of an algebraic transformation but can be matched by some intermediate step, our tool can apply the rule, but starting in the intermediate state. It can, therefore, start with the code in Listing 1.1 and apply the sequence of rules starting in Listing 1.2 and reach the code in Table 3, right.

4 Conclusions and Future Work

We have presented a rule-based approach to transform scientific code targeting heterogeneous platforms. The transformation uses mathematical properties enabling transformations that could not be performed otherwise. Mathematical properties are provided by annotations in the code and used to apply algebraic transformations rules on the code. Metrics associated to each rule reflect non-functional properties and guide the transformation process in order to generate optimized code for heterogeneous platforms.

[4] Note that coarse-grained bodies are not necessarily good for FPGAs or GPUs.

We have developed a prototype tool implementing the approach described. Currently the tool can transform code implementing some algebraic operations. As a future work we plan to develop more rules and metrics to target more complex scientific programs.

References

1. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Computing Surveys (CSUR) 26(4), 345–420 (1994)
2. Bagge, O.S., Kalleberg, K.T., Visser, E., Haveraaen, M.: Design of the CodeBoost Transformation System for Domain-Specific Optimisation of C++ Programs. In: Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003. pp. 65–75. IEEE (2003)
3. Boekhold, M., Karkowski, I., Corporaal, H.: Transforming and parallelizing ANSI C programs using pattern recognition. In: High-Performance Computing and Networking. pp. 673–682. Springer (1999)
4. Borovanskỳ, P., Kirchner, C., Kirchner, H., Moreau, P.E.: ELAN from a Rewriting Logic Point of View. Theoretical Computer Science 285(2), 155–185 (2002)
5. Borovanskỳ, P., Kirchner, C., Kirchner, H., Moreau, P.E., Ringeissen, C.: An Overview of ELAN. Electronic Notes in Theoretical Computer Science 15, 55–70 (1998)
6. Boyle, J.M.: Abstract Programming and Program Transformation - An Approach to Reusing Programs. In: Biggerstaff, T.J., Perlis, A.J. (eds.) Software Reusability, vol. 1, pp. 361–413. ACM Press (1989)
7. Boyle, J.M., Harmer, T.J., Winter, V.L.: The TAMPR Program Transformation System: Simplifying the Development of Numerical Software. In: Arge, E., Bruaset, A.M., Langtangen, H.P. (eds.) Modern Software Tools for Scientific Computing, pp. 353–372. Birkhauser Boston Inc., Cambridge, MA, USA (1997)
8. Brown, A., Luk, W., Kelly, P.: Optimising Transformations for Hardware Compilation. Tech. rep., Imperial College London (2005)
9. Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S.: The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. pp. 63–74. ACM (2010)
10. Di Napoli, E., Fabregat-Traver, D., Quintana-Orti, G., Bientinesi, P.: Towards an efficient use of the BLAS library for multilinear tensor contractions. Applied Mathematics and Computation 235, 454–468 (May 2014)
11. Fabregat-Traver, D., Bientinesi, P.: Application-tailored linear algebra algorithms: A search-based approach. International Journal of High Performance Computing Applications (IJHPCA) 27(4), 425 – 438 (Nov 2013)
12. Franchetti, F., Voronenko, Y., Püschel, M.: FFT Program Generation for Shared Memory: SMP and Multicore. In: Supercomputing (SC) (2006)
13. Hogben, L.: Handbook of Linear Algebra. (Discrete Mathematics and Its Applications), Chapman & Hall/CRC, 1 edn. (Nov 2006)
14. Kennedy, K., McKinley, K.S.: Maximizing loop parallelism and improving data locality via loop fusion and distribution. In: Languages And Compilers For Parallel Computing. pp. 301–320. Springer-Verlag (1994)
15. Schupp, S., Gregor, D., Musser, D., Liu, S.M.: Semantic and behavioral library transformations. Information and Software Technology 44(13), 797–810 (2002)

16. Visser, E.: Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. In: Lengauer, C., Batory, D., Consel, C., Odersk, M. (eds.) Domain-Specific Program Generation, Lecture Notes in Computer Science, vol. 3016, pp. 216–238. Springer-Verlag (June 2004)
17. Visser, E.: A Survey of Strategies in Rule-Based Program Transformation Systems. Journal of Symbolic Computation 40(1), 831–873 (2005), special issue on Reduction Strategies in Rewriting and Programming
18. van Wijngaarden, J., Visser, E.: Program Transformation Mechanics. Tech. rep., Technical Report UU-CS-2003-048, Universiteit Utrecht (2003)