RECKA and RPromF: Two Frama-C Plug-ins for Optimizing Registers Usage in CUDA, OpenACC and OpenMP Programs

Rokiatou Diarra, Alain Merigot, Bastien Vincke

RECKA and RPromF: two Frama-C Plug-ins for Optimizing Registers usage in CUDA, OpenACC and OpenMP Programs

Rokiatou DIARRA
SATIE, Univ. Paris-Sud
Univ. Paris-Saclay
94235 Cachan, France
rokiatou.diarra@u-psud.fr

Alain MERIGOT
SATIE, Univ. Paris-Sud
Univ. Paris-Saclay
94235 Cachan, France
alain.merigot@u-psud.fr

Bastien VINCKE
SATIE, Univ. Paris-Sud
Univ. Paris-Saclay
94235 Cachan, France
bastien.vincke@u-psud.fr

***Abstract*—Pointer aliasing still hinders compiler optimizations. The ISO C standard 99 has added the *restrict* keyword that allows programmer to specify non-aliasing as an aid to the compiler's optimizer. The task of annotating pointers with the *restrict* keyword is still left to the programmer and this task is, in general, tedious and prone to errors. Scalar replacement is an optimization widely used by compilers. In this paper, we present two new Frama-C plug-ins, RECKA for automatic annotation of CUDA kernels arguments with the *restrict* keyword, and RPromF for scalar replacement in OpenACC and OpenMP 4.0/4.5 codes for GPU. More specifically, RECKA works as follows: (i) an alias analysis is performed on CUDA kernels and their callers; (ii) if not found any alias then CUDA kernels are cloned, the clones are renamed and their arguments are annotated with the *__restrict__* qualifier; and (iii) instructions are added to kernels call sites to perform at runtime a less-than check analysis on kernel actuals parameters and determine if the clone must be called or the original one. RPromF includes five main steps: (i) OpenACC/OpenMP offloading regions are identified; (ii) functions containing these offloading codes and their callers are analyzed to check that there is no alias; (iii) if there is no alias then the offloading codes are cloned; (iv) clone's instructions are analyzed to retrieve data reuse information and perform scalar replacement; and instructions are added to be able to use the optimized clone whenever possible. We have evaluated the two plug-ins on PolyBench benchmark suite. The results show that both scalar replacement and the usage of *restrict* keyword are effective for improving the overall performance of OpenACC, OpenMP 4.0/4.5 and CUDA codes.**

***Index Terms*—CUDA, OpenACC, OpenMP, scalar replacement, static analysis, alias analysis, Frama-C**

I. Introduction

Nowadays all computing systems are becoming heterogeneous for higher power efficiency and computation throughput. Such systems may include general purpose CPUs and accelerators, such as the Graphics Processing Unit (GPU). There are various parallel programming frameworks for GPU. Kernel-based languages, CUDA or OpenCL, offer a number of features for performance optimization as the architecture is directly accessible to the user but this adds complexities for application developers. Directive-based programming models (e.g.: OpenMP 4.0/4.5, OpenACC) may therefore become an interesting solution.

Variants of the C language (e.g., CUDA, OpenCL) are still used to program on GPU machines. One of the most important features of languages such as C is the existence of pointers. Pointer can hinder compiler optimization. Indeed, it is hard to know where pointers are pointing and compilers must be conservative in their presence. Consider the C version in the example provided in listing 1, without further knowledge or special hardware support, the compiler must assume that A, B, x, y and tmp may refer to the same memory region or overlapping regions so that the loop cannot be parallelized or software-pipelined because it has to be ensured that an update of $tmp[i]$ is performed before the next value of $x[j+1]$ is loaded for example. Therefore, because the compiler must conservatively assume the pointers alias, it will compile this code inefficiently. Thus on CPU, compilers typically generate various assembly codes for the C version of *gesummv* routine (shown in listing 1), depending on the arguments passed to *gesummv* on the call site.

```
1 void gesummv(float *A, float *B, float *tmp,
2               float *x, float *y) {
3   int i, j;
4   for (i = 0; i < NI; i++) {
5     for (j = 0; j < NJ; j++) {
6       tmp[i] += A[i * NJ + j] * x[j];
7       y[i]   += B[i * NJ + j] * x[j]; }
8     y[i] = ALPHA * tmp[i] + BETA * y[i]; } }
```

Listing 1. A simple example of gesumv kernel

Unlike the CPU case, GPU codes compilers (e.g.: nvcc, pgcc, clang) generate a single PTX code for the kernel assuming that kernel arguments pointers might be aliased at runtime. Thus, it may be useful in the absence of aliasing to inform the compiler that kernel pointers arguments are not aliased, so that it can generate a more optimized code.

To ensure that the correct code is generated in the presence of aliases, compilers have to perform an alias analysis. Although the literature of alias analysis is abundant and much work (e.g, [1]–[3]) has been done in the last few decades, the research community has not yet solved pointer alias analysis satisfactorily. Many alias analyzer are implemented in mainstream compilers but the results of these analyzers

are often inaccurate. Pointer analysis imprecision prevents the compiler from optimizing some code where there is no aliasing. To mitigate the problem posed by pointers, the C programming language, since the C99 Standard, features the *restrict* keyword, that can be used by the programmer to give the compiler information about aliasing. For instance, if the arguments of the routine in listing 1 have been annotated with the *restrict* qualifier, thereby allowing the compiler to perform more aggressive optimization, such as instructions scheduling, register promotion, redundant load/store elimination, etc. Although the *restrict* keyword has been available for several years already, it remains less used by programmers and its insertion is left to the programmer. The task of inserting the *restrict* qualifier is, in general, tedious and prone to errors.

The goal of scalar replacement is to identify repeated accesses made to the same memory address and to remove the redundant accesses by keeping the data in registers. This is done by identifying sections of the code in which it is safe to place the redundantly accessed data in a register.

Since the release of the OpenACC and OpenMP 4.0 standards, many works (e.g., [4], [5]) have been done to evaluate their performance against those of CUDA and OpenCL. Other works (e.g., [6], [7]) suggested some optimizations based on the implementation of new directives. Moreover, many efforts have been done to improve OpenACC and OpenMP compilers in order to generate more optimized code. However, like other mainstream compilers, they still fail to recognize even the simplest opportunities for reuse of subscripted variables.

In this paper, we present RECKA and RPromF, two new Frama-C plug-ins we developed for automatic insertion of *restrict* keyword and scalar replacement in GPU codes. The remainder of this paper is organized as follows. Section II provides a brief introduction about CUDA, OpenACC and OpenMP 4.0/4.5 programing models. Section III reviews related work. Section IV describe briefly the Frama-C framework. Section V develops the design of different components of the RECKA and RPromF plug-ins. Section VI deals with the evaluation of the two plug-ins and analysis of the obtained results. Finally, Section VII presents concluding remarks and future works.

II. Background

In this section, we provide a brief introduction about CUDA, OpenACC and OpenMP 4.0/4.5 programing models.

A. CUDA programming model

CUDA is a parallel computing programming model that fully utilizes hardware architecture and software algorithms to accelerate various types of computation. In CUDA, the programmer writes device code in functions called **kernel**. To obtain optimized code, the programmer must understand well GPU architecture and CUDA optimization strategies like memory-coalescing access, efficient usage of shared memory or tiling technology. Additionally, grid and block configurations, computing behaviors of each thread, and synchronization problems also need to be carefully tuned. Listing 2 shows a simple restricted CUDA implementation of the C version of *gesumv* kernel provided in listing 1.

```
1  __global__ void gesummvCUDA(float *__restrict__ A,
2  float *__restrict__ B, float *__restrict__ tmp,
3  float *__restrict__ x, float *__restrict__ y)
4  {
5    int i = blockIdx.x * blockDim.x + threadIdx.x;
6    if (i < NI) {
7      int j;
8      for(j = 0; j < NJ; j++) {
9        tmp[i] += A[i * NJ + j] * x[j];
10       y[i]   += B[i * NJ + j] * x[j]; }
11     y[i] = ALPHA * tmp[i] + BETA * y[i];
12   } }
```

Listing 2. A simple implementation of gesumv in CUDA

B. OpenMP 4.0/4.5 programming model

OpenMP 4.0 introduced new directives for programming accelerators such as GPUs and Intel Xeon Phi. In order to offload a region of code into device, OpenMP 4.0/4.5 uses the **target** construct. Various directives are provided to express the levels of parallelism. The **teams** construct, creates a league of thread teams where the master thread of each team executes the region. The **distribute** construct specifies loops which are executed by the thread teams. The **target data** construct handles data transfers between host and accelerators. An exhaustive description of OpenMP directives is provided in [8]. Listing 3 shows OpenMP version of code provided in listing 1, when scalar replacement is performed.

```
1  void gesummvMP(float *A, float *B, float *tmp,
2                 float *x, float *y) {
3    int i, j;
4    #pragma omp target teams distribute \
5    is_device_ptr(A,B,tmp,x,y)
6    for (i = 0; i < NI; i++) {
7      float ti = tmp[i];
8      float yi = y[i];
9      for (j = 0; j < NJ; j++) {
10       ti += A[i * NJ + j] * x[j];
11       yi += B[i * NJ + j] * x[j]; }
12     y[i] = ALPHA * ti + BETA * yi;
13   } }
```

Listing 3. OpenMP version of gsummv kernel using scalar replacement

C. OpenACC programming model

OpenACC is another specification focused on directive-based ways to program accelerators. OpenACC has fewer constructs but most of them are analogous to those of OpenMP 4.0/4.5. The OpenACC **parallel** construct starts parallel execution on the current accelerator device by creating one or more gangs of workers. The **kernels** construct defines a region of the program that is to be compiled into a sequence of kernels for execution on the current accelerator device. The **loop** construct specifies the distribution of iterations. Detailed description of OpenACC directives is provided in [9]. Listing 4 shows OpenACC version of code showed in listing 1.

```
1  void gesummvACC(float *A, float *B, float *tmp,
2                  float *x, float *y) {
3    int i, j;
4    #pragma acc kernels loop independent \
5    deviceptr(A,B,tmp,x,y)
```

```
6   for (i = 0; i < NI; i++) {
7     for (j = 0; j < NJ; j++) {
8       tmp[i] += A[i * NJ + j] * x[j];
9       y[i]   += B[i * NJ + j] * x[j]; }
10    y[i] = ALPHA * tmp[i] + BETA * y[i];
11  } }
```

Listing 4. OpenACC version of gesummv kernel

III. Previous Work

Alias analysis is one of the most used techniques that aims to optimize languages with pointers. The literature of alias analysis is abundant. The most popular algorithm for context-insensitive, flow-insensitive, iterative and constrains-based points-to analysis is known as inclusion-based or Andersen-style analysis [1]. Alves et al. provided in [10] two ways to determine at runtime when two memory locations can overlap. They concluded that the combination of cloning plus dynamic disambiguation of pointers is an effective way to make compiler optimizations more practical. Sperle et al. presented in [11] three different techniques to disambiguate pointers used as arguments of functions. Their first technique relies on the static alias analysis already available in mainstream compilers to perform pointer disambiguation and the others combine static bound inference with code cloning, hence, extending the reach of pointer disambiguation. Maalej et al. introduced in [3] a new technique to disambiguate pointers, which relies on a less-than analysis. Their alias analysis uses the observation that if $p1$ and $p2$ are two pointers, such that $p1 < p2$, then they cannot alias. The original scalar replacement algorithm was proposed by Carr-Kennedy in [12] more than 20 years ago. Since then, several works have been done to improve this algorithm in many aspects. Surendran et al. presented in [13] new algorithms for scalar replacement and dead store elimination based on Array SSA form. Byoungro et al. described in [14] an algorithm for scalar replacement that can exploit reuse opportunities across multiple loops. Tian et al. presented in [15] an extension to the classical scalar replacement algorithm for optimizing registers usage in OpenACC codes. Their approach is based on feedback information regarding register utilization and a memory latency-based cost model to select which array references should be replaced by scalar references.

IV. Preliminaries: Frama-C

Frama-C[1] is a static analyzer for C code. It provides its users with a collection of plug-ins that perform static analysis, deductive verification, and testing, for safety- and security-critical software [16]. The platform is based on a common kernel, which hosts analyzers as collaborating plug-ins. Frama-C kernel is based on a modified version of CIL. CIL is a front-end for C that parses ISO C99 programs into a normalized representation. For instance, *for* loops are replaced by equivalent *while* loops , normalized expressions have no side-effects, ect. Frama-C extends CIL to support other features such as ACSL (ANSI/ISO-C Specification Language) annotations. This modified CIL front-end produces the C + ACSL abstract syntax tree (AST). The AST assigns unique identifiers for statements and blocks that can be used for the program counter, it also keeps line numbers. In addition to the AST, the kernel provides several general services. For example, it provides a visitor mechanism to facilitate crawling through the AST. In general, writing a Frama-C plug-in requires to visit the AST to compute information for some C constructs.

V. Design Principles

This section presents alias analysis we implemented to identify on which pointers of CUDA kernel arguments must be added the *restrict* keyword, our scalar replacement algorithm implementation, and describes the design of the different modules composing the RECKA and RPromF tools.

A. The alias analysis module

Since the optimized version of a function by performing the scalar replacement is equivalent to its original version only if there is no alias and marking aliased pointers as restricted may result in undefined behavior, we run an alias analysis before making any changes in the source code. For that purpose, we have implemented in Frama-C a simple and fast alias analyzer inspired by Andersen-style analysis [1] and the concepts of the *basicaa* pass of LLVM[2]. In our implementation, we only considered two pointers operations: taking the address of a variable (e.g.: $p = \&val$) and assignments (e.g.: $p = q$, $p = Tab$ where p and q are pointers and Tab a constant array).

B. Pre-processing module

Frama-C can't analyze CUDA codes or programs comprising OpenACC and OpenMP directives. On the other hand, Frama-C transform all *for* loops in *while* loops thus denaturing the source code. Thus, it is necessary to do some text processing tasks on the source code before and after the analysis with Frama-C. To this end, we developed two programs in Perl for the pre-processing operations. The first takes as input a CUDA program, copies it into a new file, hides all CUDA specific keywords, types or API functions such as __global__, cudaMalloc, etc. It also adds an identifier to the kernels names so that we can find them in next steps. The second takes as input a source file with OpenACC/OpenMP pragmas, copies it in a new file, replaces all OpenMP/OpenACC pragmas by flags that will allow us to find, in next steps, offloading regions and to know what directives has been used. In both cases, these two programs generate an intermediate C file that will be passed to Frama-C.

C. Post-processing module

According to the results of the analysis made with Frama-C on the file generated by the pre-processing step , changes must be made to the source file. We developed another program in Perl to make these modifications. This program takes as input the initial source file and the output file of our Frama-C plug-ins and modifies the initial source file. For example, if the

[1]https://frama-c.com/

[2]https://llvm.org/docs/AliasAnalysis.html#the-basicaa-pass

source file is a CUDA program then the output file of the RECKA plug-in is used and the changes made to the source file are: (i) clone a CUDA kernel; (ii) change the clone's name and add the restrict keyword on its arguments; and (iii) change this kernel call sites by adding an if statement allowing to call either the clone or the original version depending on the runtime less-than check result.

D. RECKA: REstrictification of CUDA Kernel pointers Arguments

The RECKA plug-in performs some alias analysis on a CUDA program and adds the *restrict* qualifier to kernels pointers arguments, in the absence of aliases. It takes as input the intermediate file generated in the pre-processing step and return a text file. This file contain messages that will make it possible to change the initial source file during the post-processing step. The RECKA plug-in first starts by identifying all CUDA kernels definition and then performs an alias analysis in these kernels. Then for each kernel where there is no alias, he performs another alias analysis in the launcher function of this kernel. If no alias is found then it print messages to clone this kernel, rename it and add *restrict* keyword on its pointers arguments. It also saves in an hash table all device pointers and allocated arrays sizes. This table is then used to print messages that will make it possible to insert, in the source file, instructions to check on runtime if kernel arguments pointers do not actually point to overlapping regions. For generating messages for the runtime less-than check test, we implemented a module in Frama-C inspired from the methods described in [3]. Figure 1 shows an overview of this plug-in.

E. RPromF: Register Promotion with a Frama-C plug-in

The goal of this plug-in is to perform the scalar replacement in OpenACC/OpenMP programs. It takes as input an intermediate file generated by the pre-processing module. First, it identifies OpenACC/OpenMP offloading regions by using flags added by the preprocessing step. After that, an alias analysis is performed on functions containing these offloading codes as well as their callers if any. If alias analysis concludes that there is no alias involving any variables used in the offloading regions then generates messages that will make it possible to clone the offloading codes. Then try to identify repeated accesses made to the same memory address by using our simplified implementation of Carr-Kennedy algorithm [12]. If found repeated accesses then prints messages that will make possible to perform the scalar replacement in offloading regions clones. Finally, it generates messages that will make it possible to insert in the source code an *if* statement allowing at runtime to use either the clone or the original version depending on the result of the runtime less-than check analysis. Figure 2 shows an overview of this plug-in.

VI. Evaluation

We tested our tool on kernels taken from the PolyBench[3] benchmark suite, to assess the ability of RECKA and RPromF plug-ins. We choose this benchmark because there are many potentials pointers aliasing situations in its kernels. For our experiments, performance data were collected on a NVIDIA GPU Quadro M2000M hosted in an Intel I7 CPU. LLVM/-Clang[4] (4.0) was used to compile our OpenMP versions. For OpenACC, we used the PGI 17.10.0 compiler (pgcc). All performance data were collected with NVIDIA profiler, *nvprof*. For all arrays, we used *float* as data type and the extra large dataset value provided in PolyBench. All versions were compiled with *-O3* flag.

Results analysis

Our preliminary results showed that RECKA was successfully able to annotate all pointers arguments of kernels with the *__restrict__* keyword without any misplacement. RPromF has also been able to place correctly all the data accessed redundantly in a register. Since our goal is to evaluate the impact of the *restrict* keyword and scalar replacement on application performances, we did not compare OpenACC or OpenMP against CUDA.

Figure 3 shows factors relative to the original codes for the CUDA kernels performance including compute time, executed instructions count, global load and L2 read transactions. We observed that the *restrict* qualifier reduces kernel compute time by 24% on average. However, this factor is more interesting for the *correlation* and *covariance* application where there are many potential pointer aliasing and redundant load/store. We found that annotating pointers arguments with the *restrict* reduced the executed instruction count by $6\%s$ on average and the global load transaction only when there are many redundant load as it is the case for the *adi*, *gesum*, *gramschmidt* and *mvt* kernels. Regarding the L2 cache read transactions, we found that the *restrict* keyword reduced it by 18% on average. We can conclude that *restrict* keyword improve CUDA kernels overall performance, and that it must be used whenever possible to help *nvcc* to generate more optimized code.

Regarding the impact of scalar replacement on OpenACC and OpenMP codes for GPU, we compared the performance of the generated code, after running RPromF on PolyBench kernels, to that of the original versions to show that the scalar replacement, performed in source level, really contributes to improve performance. Figure 4 shows factors relative to the original code for the offloading regions compute time. We observed that with OpenACC, the scalar replacement reduced kernels compute time by 43% on average. While with OpenMP we found that scalar replacement reduce kernels compute time by 31% on average. We can show on both figures 4 and 3, the performance gain obtained on *gesum* kernel which was taken as an example in section II. In summary, we observed

[3]https://sourceforge.net/projects/polybench/

[4]https://github.com/clang-ykt

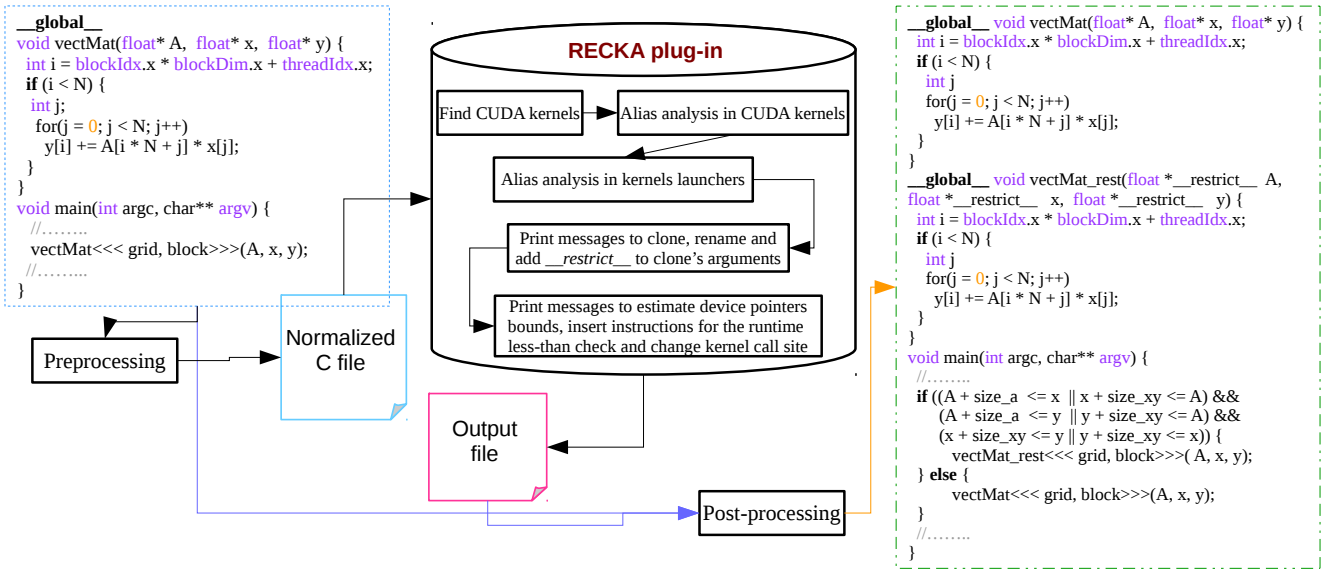

Fig. 1. An overview of RECKA

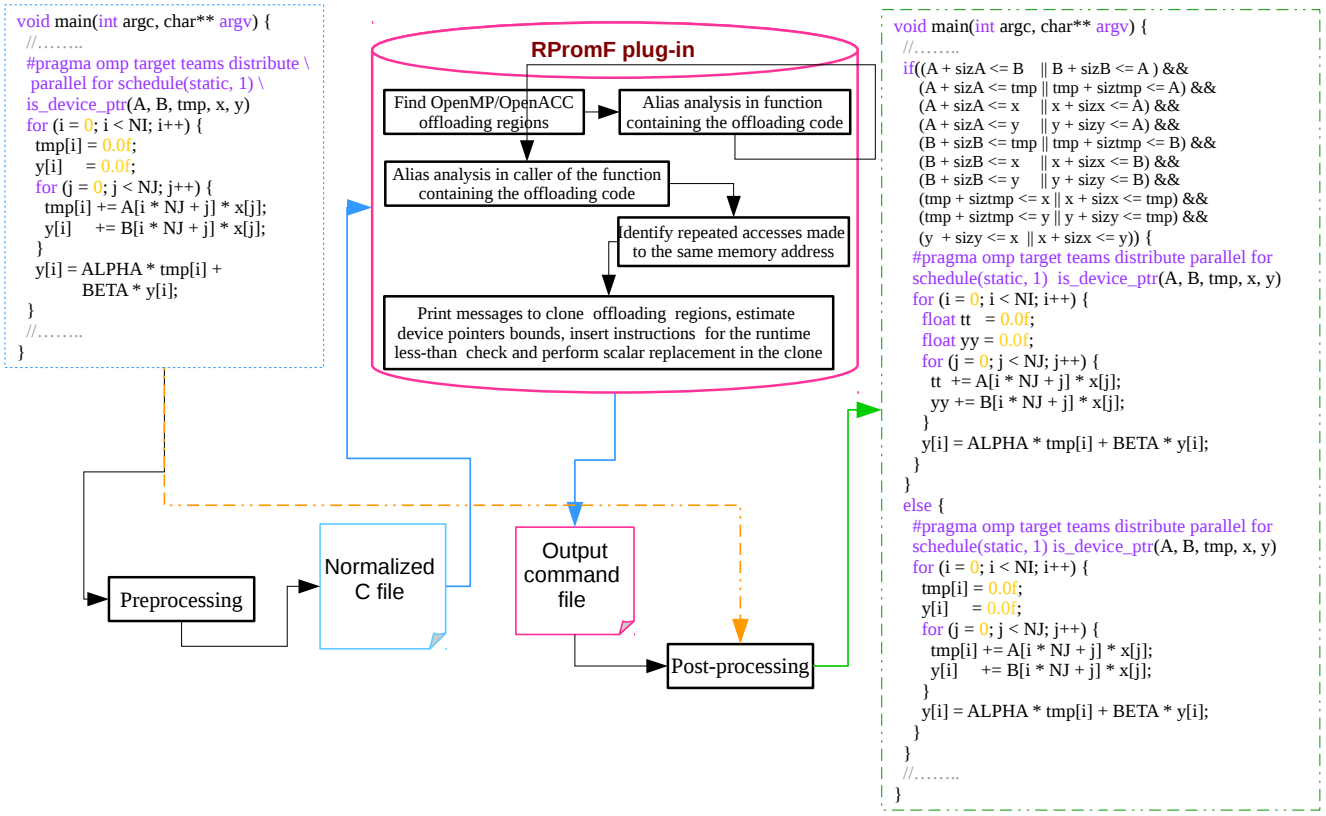

Fig. 2. An overview of RPromF

that in general scalar replacement improve both OpenACC and OpenMP code performances.

VII. Conclusion and Future Work

In this paper, we have implemented two new Frama-C plug-ins, RECKA for automatic annotation of CUDA kernels arguments with the *restrict* keyword, and RPromF for scalar replacement in OpenACC and OpenMP 4.0/4.5 codes for GPU. RECKA works in three main steps. First, an alias analysis is performed on CUDA kernels and their callers to be sure that there is no alias and that kernels arguments can be marked as restricted pointers. Second, if not found any alias then CUDA kernels are cloned, the clones are renamed and

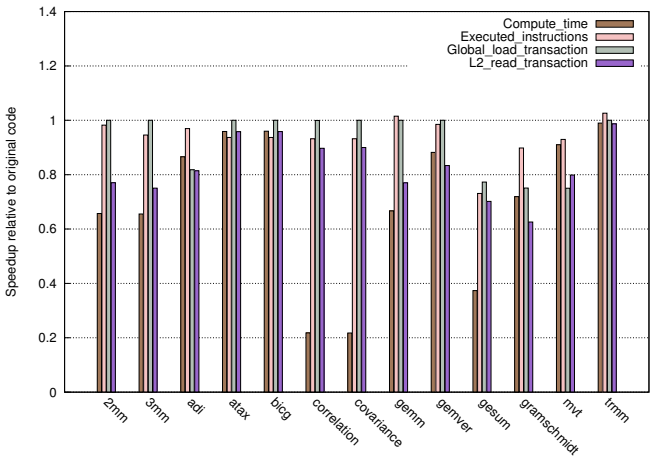

Fig. 3. Restrict keyword impact on CUDA kernels performance

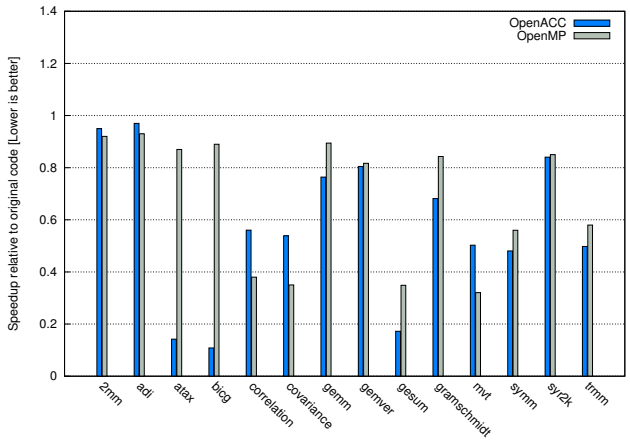

Fig. 4. Speedup results of PolyBench benchmark suite with scalar replacement

their arguments are annotated with the *__restrict__* qualifier. Third, instructions are added to kernels call sites to perform at runtime a less-than check analysis on kernel actuals parameters and determine if the clone must be called or the original one. On the other hand, RPromF includes five main steps. First, OpenACC/OpenMP offloading regions are identified. Second, functions containing these offloading codes and their callers are analyzed to verify if there is no alias implying any variables used in offloading regions. Third, if there is no alias then the offloading code are cloned. Fourth, clone's instructions are analyzed to retrieve data reuse information and perform scalar replacement. Fifth, instructions are added to be able to use the optimized clone whenever possible. We have assessed the two plug-ins on the PolyBench benchmark suite. The results show that both scalar replacement and the usage of *__restrict__* keyword are effective for improving the overall performance of OpenACC, OpenMP $4.0/4.5$ and CUDA codes. In future work, we plan to evaluate both plug-ins on more complex benchmarks such as SPEC and Rodinia for example and trying to add a cost model to avoid registers spilling with scalar replacement. While Frama-C has two deductive verification plug-ins, we intend to use them in our tool in order to be able to prove the reliability of the generated code.

REFERENCES

[1] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, DIKU, University of Copenhagen, may 1994.

[2] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 289–298. [Online]. Available: http://dl.acm.org/citation.cfm?id=2190025.2190075

[3] M. Maalej, V. Paisante, P. Ramos, L. Gonnord, and F. M. Q. a. Pereira, "Pointer disambiguation via strict inequalities," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 134–147. [Online]. Available: http://dl.acm.org/citation.cfm?id=3049832.3049848

[4] M. Martineau, S. McIntosh-Smith, and W. Gaudin, "Evaluating openmp 4.0s effectiveness as a heterogeneous parallel programming model," *IEEE International Parallel and Distributed Processing Symposium Workshops*, may 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7529889/?arnumber=7529889

[5] S. Memeti, L. Li, S. Pllana, J. Kolodziej, and C. Kessler, "Benchmarking opencl, openacc, openmp, and cuda: Programming productivity, performance, and energy consumption," in *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, ser. ARMS-CC '17. New York, NY, USA: ACM, 2017, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/3110355.3110356

[6] A. Lashgar and A. Baniasadi, "Openacc cache directive: Opportunities and optimizations," *Third Workshop on Accelerator Programming Using Directives*, nov 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7836580/

[7] A. Hayashi, J. Shirako, E. Tiotto, R. Ho, and V. Sarkar, "Exploring compiler optimization opportunities for the openmp 4.x accelerator model on a power8+gpu platform," *Third Workshop on Accelerator Programming Using Directives*, nov 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7836582/

[8] O. A. R. Board, "Openmp application programming interface," http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, 2015.

[9] OpenACC-Standard, "The openacc application programming interface," http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf, 2015.

[10] P. Alves, F. Gruber, J. Doerfert, A. Lamprineas, T. Grosser, F. Rastello, and F. M. Q. a. Pereira, "Runtime pointer disambiguation," *SIGPLAN Not.*, vol. 50, no. 10, pp. 589–606, Oct. 2015. [Online]. Available: http://doi.acm.org/10.1145/2858965.2814285

[11] V. H. Sperle Campos, P. R. Alves, H. Nazaré Santos, and F. M. Quintão Pereira, "Restrictification of function arguments," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: ACM, 2016, pp. 163–173. [Online]. Available: http://doi.acm.org/10.1145/2892208.2892225

[12] S. Carr and K. Kennedy, "Scalar replacement in the presence of conditional control flow," *Softw. Pract. Exper.*, vol. 24, no. 1, pp. 51–77, Jan. 1994.

[13] R. Surendran, R. Barik, J. Zhao, and V. Sarkar, "Inter-iteration scalar replacement using array ssa form," *Cohen A. (eds) Compiler Construction. CC 2014. Lecture Notes in Computer Science*, vol. 8409, 2014.

[14] B. So and M. Hall, "Increasing the applicability of scalar replacement," in *Compiler Construction*, E. Duesterwald, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 185–201.

[15] X. Tian, D. Khaldi, and D. Eachempati, "Optimizing gpu register usage: Extensions to openacc and compiler optimizations," *45th International Conference on Parallel Processing (ICPP)*, aug 2016.

[16] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Form. Asp. Comput.*, vol. 27, no. 3, pp. 573–609, May 2015. [Online]. Available: http://dx.doi.org/10.1007/s00165-014-0326-7