Semi-Automatic Generation of Adaptive Codes

Maxime Schmitt, César Sabater, Cédric Bastoul

Semi-Automatic Generation of Adaptive Codes

Maxime Schmitt
Univ. of Strasbourg and Inria
Strasbourg, France
max.schmitt@math.unistra.fr

César Sabater
Univ. Nacional de Rosario
Rosario, Argentina
csabater89@gmail.com

Cédric Bastoul
Univ. of Strasbourg and Inria
Strasbourg, France
cedric.bastoul@unistra.fr

Figure 1: Fluid simulation snapshot (left), corresponding monitored state grid (center), generated computation volume (right)

ABSTRACT

Compiler automatic optimization and parallelization techniques are well suited for some classes of simulation or signal processing applications, however they usually don't take into account domain-specific knowledge nor the possibility to change or to remove some computations to achieve "good enough" results. Quite differently, production simulation and signal processing codes have adaptive capabilities: they are designed to compute precise results only where it matters if the complete problem is not tractable or if computation time must be short. In this paper, we present a new way to provide adaptive capabilities to compute-intensive codes automatically. It relies on domain-specific knowledge provided through special pragmas by the programmer in the input code and on polyhedral compilation techniques to continuously regenerate at runtime a code that performs heavy computations only where it matters at every moment. We present a case study on a fluid simulation application where our strategy enables significant computation savings and speedup in the optimized portion of the application while maintaining a good precision, with a minimal effort from the programmer.

IMPACT 2017
Seventh International Workshop on Polyhedral Compilation Techniques
Jan, 2017, Stockholm, Sweden
In conjunction with HiPEAC 2017.

http://impact.gforge.inria.fr/impact2017

1. INTRODUCTION

A large range of compute-intensive applications are calculating approximate results. This is especially true for simulation codes which are based on inherently imperfect models that try to emulate as precisely as possible a real world object or phenomenon. This is also true for signal processing applications which are limited by the precision of, e.g., input sensors or processing algorithms. Some other applications may also compute approximate results for functional reasons, e.g., to meet a deadline such as in real-time video decoding, or because the precise result is not tractable or valuable such as late earthquake prediction, or simply because the user needs a rough result to drive further precise investigations such as in geophysics. For such applications, "ideal" naive computation kernels, that would be convenient with an infinite computation power, are often designed at first for algorithmic tuning and debugging purposes. Then they are optimized to a "production" version exploiting possible approximations to scale to the actual problem size or to meet the deadline. Translating an ideal code to a production code is complex, time consuming, leads to less maintainable codes and must be redone when a major change in the initial strategy arises. In this paper, we present a new compiler technique that automates the conversion from an ideal code to a version that exploits approximations dynamically, by adjusting computations with respect to the current state of the program (Fig. 1). It aims at improving both developer's productivity and approximation's quality.

State-of-the-art automatic optimization and parallelization compiler techniques heavily rely on the *polyhedral model* to manipulate computation-intensive kernels to aggressively restructure them at the iteration level [7, 4, 11]. They are usually based on exact, or over-approximated, data depen-

dence analysis to ensure that the optimization preserves the original program semantics. Relaxed semantics models are possible, e.g., to support commutativity to enable vectorization [9], however all input code iterations are to be executed in the optimized code. On the other hand, more aggressive techniques to automatically compute approximations have been designed, e.g., by ignoring some dependences to enable parallelization [6], by providing alternative implementations of some code parts [1], or by skipping computations [14, 12]. In our work, we investigate a new way, called *Adaptive Code Refinement* (ACR), inspired by Adaptive Mesh Refinement [3], a classical numerical analysis technique providing the ability to dynamically tune a computational grid to achieve precise computation only where it matters. We achieve this goal by exploiting high-level information provided by the user, a dynamic optimization strategy and state-of-the-art polyhedral code generation techniques.

Our strategy is supported by high-level information provided by the user and a static-dynamic code generation approach. First, we designed a set of pragmas to provide the user with the means to express both static and dynamic approximation-related information. It allows the user to focus only on an "ideal" version of the computational kernels. Those kernels can actually compile and run while ignoring the pragmas, but they may only produce results in a reasonable amount of time for small problems. The pragma set is quickly described in Section 2. Next, ACR uses polyhedral code generation techniques to generate a code to compute approximate results according to static information, with the ability to regenerate itself according to the evolution of the computed values and to pre-defined approximation strategies. In a nutshell, ACR decomposes the computation space into a grid and monitors specific values at the grid level. Depending on their evolution, it will tune the computation volume onto each grid cell to ensure complex computation is done only where and when it matters. According to this monitoring, it will (1) generate an optimized version of the code specific to the current situation, (2) compile it dynamically and (3) switch the execution to the new optimized code when it is ready. Specific threads are devoted to monitoring and code generation for a minimum overhead. The continuous monitoring and code generation process enables strong computation savings while limiting accuracy loss. The complete process is depicted in Section 3.

We present a case study and an evaluation of ACR on an Eulerian fluid simulation implementation in Section 4. The study shows empirical evidence that ACR allows significant time and computation savings while maintaining accuracy with minimal efforts from the programmer, even compared to a hand-tuned version.

2. ACR PRAGMA SET

ACR offers a set of high-level pragmas that allow the user to provide high-level, domain-specific approximation information. The ACR optimizing algorithm will exploit them to compute quality approximate results automatically. ACR pragmas are language extensions to be inserted before a computational loop to compute approximations using either alternative implementations to those provided inside the loop or a reduced number of computations in some areas of the computation space. The pragma set is divided into four constructs:

1. The `monitor` construct specifies which data has to be monitored for the dynamic optimization strategy and how the monitoring is summarized for a complete cell. Its format is:
   `#pragma acr monitor(`*data*$[f(\vec{i})]$`,`*summary*`[, `*filter*`])`
   where *data* is the data array to monitor (using the access function $f(\vec{i})$ of the iteration vector $\vec{i}$), *summary* specifies how monitored values should be summarized into one value at the cell level (using predefined policies, e.g., `mean`, `max`, `min`, etc.) and *filter* is an optional function that may be used to preprocess values being monitored (e.g., to classify them into categories): the actual value used to drive the dynamic optimization strategy will be *filter(data)*.

2. The `grid` construct defines the granularity of the dynamic approximation strategy: the `data space` is decomposed into cells of a size equal to the specified grid value. The monitoring as well as the approximation strategy will be done at the cell-level granularity. Its format is:
   `#pragma acr grid(`*size*`)`
   where *size* is a constant number: if the computation space is 2-dimensional, the cell size will be $size \times size$.

3. The `strategy` construct specifies in which conditions which alternatives should be used for each cell. They may be either static (specifying areas of the computation space where to apply a given alternative) or dynamic (specifying which alternative to use in a cell depending on the monitored value or its evolution in that cell). The case study presented in the remainder of the paper uses a set of direct links between a dynamic monitor value and alternative computations. Its format is:
   `#pragma acr strategy direct(`*value*`, `*alternative*`)`
   where *value* is a value that can be reported by a monitor for a grid cell and *alternative* is the alternative to apply to a cell if the monitor reports that value for that cell.
   An example of static strategy has the following format:
   `#pragma acr strategy zone(`*area*`, `*alternative*`)`
   where *area* is a part of the computation space expressed using set notations (e.g., `{i | 0<=i<=3}` to express the part of the space where `i` is between 0 and 3) and *alternative* is the name of an alternative defined with the `alternative` construct. Many other strategies are possible, however describing the complete set is out of the scope of this paper.

4. The `alternative` construct defines an alternative computation to the default one provided in the code. Its format is:
   `#pragma acr alternative` *name*`(`*type*`, `*effect*`)`
   where *name* is the strategy name, *type* specifies the strategy type, i.e. either `parameter` to keep the original code but with new parameter values to be defined in the *effect* field, or `code` to provide an alternative code block in the *effect* field.

Figure 2 depicts how this pragma set may be used to specify approximate computation on a simplified version of a part of our case study, detailed in Section 4.

```
// Loop iterating over frames of the fluid simulation
while(true) {
  ...
  // lin_solve kernel
  #pragma acr grid(10)
  #pragma acr monitor(density[i][j], max, filter)
  #pragma acr alternative low(parameter,    MAX = 1)
  #pragma acr alternative medium(parameter, MAX = 4)
  #pragma acr alternative high(parameter,   MAX = 8)
  #pragma acr strategy direct(1, low)
  #pragma acr strategy direct(2, medium)
  #pragma acr strategy direct(3, high)
  for (k = 0; k < MAX; k++) {
    for (i = 1; i <= N; i++) {
      for (j = 1; j <= N; j++) {
        lin_solve_computation(k, i, j);
      }
    }
  }
  ...
}
```

Figure 2: **Pseudo-code depicting the application of ACR to the `lin_solve` kernel of the fluid simulation case study**. Here, the `filter` function returns in which range a given density value is: near zero (1), medium (2) or high (3). The monitor value for a cell corresponds to the maximum of all ranges in the cell. This range is used to apply a different alternative computation for each cell. In this example, the loop `k` will iterate more or less depending on that range.

3. ADAPTIVE CODE REFINEMENT

The application domain of ACR corresponds to computational kernels such that the possible values of each loop iterator can be modeled by a set of affine constraints on that loop iterator, outer loop iterators and fixed parameters. A simple and useful case corresponds to loops such that the loop stride is a known constant and the loop bounds are linear functions of the outer loop counters and fixed parameters. Hence the application domain is code that can be raised to a polyhedral representation [7] with more general loop bodies where any kind of data access is possible and data dependent conditions are also allowed. However, ACR makes a strong use of polyhedral compilation techniques. In particular we represent the computation space and its cells as polyhedra in the same way polyhedral compilers represent iteration domains [7, 4, 11]. For instance, the computation space of the `lin_solve` loop in Figure 2 is:

$$\mathcal{S}_{lin\_solve} = \left\{ \begin{pmatrix} k \\ i \\ j \end{pmatrix} \middle| \begin{array}{lll} 0 \leq & k & < MAX \\ 1 \leq & i & \leq N \\ 1 \leq & j & \leq N \end{array} \right\}$$

To manipulate polyhedra, we rely on `isl` [16] to achieve, e.g., polyhedral unions and differences and on CLooG [2] to generate a code to scan them.

Computational loops annotated with user-provided ACR pragmas are candidates for approximate computation using our approach. In such kernels, the rigid data-dependence model used by compilers is explicitly relaxed and alternative computation can be used to make the computation faster, according to the user-defined strategy. This strategy may be static, i.e., independent of the computed values in specific areas of the computation space, or dynamic, i.e. dependent on the monitored values in parts of the computation space where no static strategy applies. The ACR optimizing algorithm is based on three main components. First it continuously evaluates a "state grid" of the data space according to the user-defined strategy as detailed in Section 3.1. Second, a dedicated thread generates an optimized code according to the state grid, this process is explained in Section 3.2. Finally, a runtime ensures the best version of the code for both performance and accuracy is used. It is discussed in Section 3.3.

3.1 State Grid

ACR needs a way to gather information about the computation at runtime to identify different approximation regions, i.e., the parts of the computation space where the approximation strategy should be different. This task should be simple enough to avoid adding significant computation overhead with regard to the time saved by the optimized code. To achieve that, a uniform grid is embedded into the data space to represent different zones of the simulation, according to the `grid` pragma. Every cell in the grid represents an (hyper-)square shaped portion of the data space. The dimension of the grid may affect the accuracy and the efficiency of the generated code in contradictory ways, hence a good tradeoff is preferable.

The information stored in the grid, or its evolution during execution, should allow to decide about the desired accuracy in the corresponding portion of the computation space. The grid is refreshed by a specific thread according to the user-provided monitoring specification. Moreover, to avoid too frequent changes in the grid (which would translate to code generations that may be obsolete when they become available), we did study different post-processing policies (evaluated on our case study in Section 4.3):

Raw (no post-processing): the grid reflects exactly the approximation strategy which should be applied according to ACR pragmas, with the risk that some regions oscillate rapidly between various approximation strategies, not leaving enough time to generate optimized codes between two changes.

Versioning : grid cells are updated when a higher precision is needed whereas precision downgrade is ignored. When the difference between the current grid and the `raw` grid is more than a given threshold, we restart with the `raw` grid. Hence some grid cells are set to an over-approximation with respect to ACR pragmas, with the risk of a less efficient computation. However this policy reduces the need for a new code compilation because we allow some cells to use higher precision temporarily.

Stencil : each grid cell is evaluated not only according to the monitored values in that cell, but also according to neighboring cells. For instance if a grid cell is set to low precision but is surrounded by high-precision cells, it is switched to high-precision as well to anticipate a probable change. Here again, some grid cells over-approximate the computation with respect to ACR pragmas, with the risk of a less efficient computation.

Once the grid is filled with the information gathered, and post-processed if requested, the regions are constructed by joining grid cells with similar approximation strategy. Technically, each grid cell is represented as a polyhedron and the region is constructed by aggregating cells with polyhedral

unions thanks to `isl`. Figure 1 shows a snapshot of a fluid simulation and the corresponding grid state: each shade of grey corresponds to a region where a similar approximation strategy should be applied.

3.2 Dynamic Code Generation

Once a state grid has been computed to reflect the current mapping of approximation strategies onto different regions of the computation space, the next step is to generate on the fly an optimized code to replace the current one (if the current state grid actually differs from the previous one) to implement the approximation.

Our solution is to translate this problem to a code generation in the polyhedral model task. Tools like CLooG [2] are able to generate an efficient code from a polyhedral representation made of two set of objects: *iteration domains* which describe the set of statement instances to execute, and *scheduling relations* which describe the relative order of the statement instances. After aggregating grid cells with similar approximation strategies together using polyhedral union operations, each region is modeled as a union of convex polyhedra. Those regions are then mapped back to the computation space by considering the iteration subspace that updates those regions. We associate each subspace with the corresponding computation that reflects the approximation strategy, i.e., a block of code or some new constraints such as parameter values (as in our example in Figure 2). Those subspaces form the input iteration domains of the code generation problem. Then, to ensure the approximated computations are processed in a similar order than the original one to preserve accuracy, we enforce the lexicographic ordering of the original computation space dimensions as the input scheduling relations of the code generation problem. Then CLooG is able to generate a code with extremely optimized control overhead to, e.g., avoid costly tests at the innermost level of the computational loop to choose the right approximation strategy, which could not be possible with a static approach.

Once CLooG has generated an approximation code, it is compiled and loaded dynamically to the computation process. Figure 1(right) provides a view of the computation space executed by the generated code according to the state grid in the center and the high-level user information from Figure 2. The automatically generated code has the following properties: (1) it has no costly internal tests to decide about the optimization strategy, (2) it makes more computation only where it is necessary and (3) the remaining computations are done with respect to the initial ordering to preserve accuracy.

3.3 Runtime

The ACR runtime is decoupled into five threads to exploit multicore architectures and to reduce the technique's overhead: (1) the *computation thread* is responsible for the main computation itself, (2) the *monitoring thread* computes the state grid, (3) the *CLooG thread* provides a code generation service: it waits for polyhedral code generation requests and generates the corresponding C codes with low control overhead[1], (4) the *compilation thread* provides a compilation service: it waits for C code compilation requests and generates the corresponding object codes[1], finally (5) the *coordinator thread* creates and manages all the other threads.

The runtime operates as summarized in Fig. 3. At the beginning of the computation, no optimized code is available. Hence, the computation thread executes the original code for the first iteration and updates the internal data structures. The monitoring thread constantly watches the monitored data as specified by ACR pragmas. When necessary, it updates the state grid and signals the coordinator thread. When the coordinator thread is signaled about the availability of a new state grid, it builds a code generation request to get an optimized code corresponding to the current situation. Then it sends it to the CLooG thread. When the CLooG thread answers, the coordinator thread sends a compilation request to the compilation thread, who answers with an object code. In the meantime, the computation thread continues the iterations with its current code. When the coordinator thread receives a new compiled optimized code, it checks whether the code generated still fits the current state of the grid or not. If yes, it updates the code of the computation thread for the next kernel call. If not, it ignores it, updates its request for an optimized code and lets the computation thread continue with the original code. The same happens if the state grid evolves while an old, not convenient anymore, optimized code is being used by the computation thread: the computation code is switched back to the original.

The runtime is optimized in several ways to ensure a convenient optimized code is available for the computation thread as soon as possible. First, the coordinator thread requests two different compiled codes for the same C input: a non-optimized one which may be generated and used quickly (we use TCC, the Tiny C Compiler for this) and a very optimized code that may be available later and that will replace the non-optimized one (we used GCC with aggressive optimization options for this). Second, the coordinator thread is using a cache of generated codes to immediately use an already generated code for a known state grid. Finally, the coordinator accepts over-approximations instead of switching back to the original code: what is needed is that the optimized code performs the same or more complex computations than the levels specified in the current grid, for every grid slot. In that way we can say that the computations done are "safe" and they do at least what was specified by the domain-specific information.

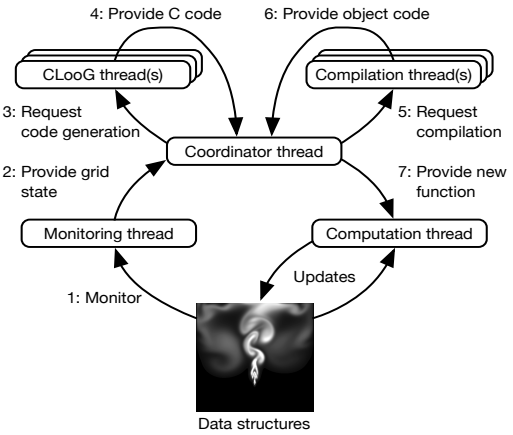


Figure 3: ACR runtime thread interaction diagram.

[1]Several server threads may coexist to process several requests concurently.

4. CASE STUDY: FLUID SIMULATION

The Adaptive Code Refinement approach has been implemented in a prototype and applied to a fluid simulation application to evaluate its effectiveness[2]. This application, called Eulerian fluid simulation, has the characteristic of being a grid based simulation. A snapshot of such simulation is shown in Fig. 1(left). Particle-based simulations and grid-based simulations are the most effective ways of simulating the behavior of fluids. Grid-based methods respond to the so-called Eulerian approach, where fluids are represented by fixed points in the space with information about the fluid in time and they are updated at every time step of the simulation. Grid based techniques often suffer from mass loss and are slower than particle based methods, but they usually have higher accuracy and better tracking of smooth fluid surfaces. They form a very suitable family of codes to apply ACR, because on one hand the simulation is an approximation of a physical phenomenon and on the other hand the processing is done on a highly regular computation space where each element requires complex computations.

4.1 Exploiting Domain-Specific Knowledge

In fluid simulation, the state of a fluid is typically represented by a velocity vector and a density value for every point in the space. The density in a given point represents the amount of fluid concentrated and the velocity vector represents the direction and intensity of the flow in that point. The evolution of the simulation is described by the Navier Stokes equations. The simulation steps can be decomposed in Advection, Diffusion and External Forces influence. Advection is the phenomena that describes how velocity moves the fluid and other objects in the space along with the flow. Diffusion describes the resistance of a fluid to flow because of its viscosity. The influence of External Forces describe local or body forces applied to a specific region or all the fluid like a fan blowing air, gravity, etc. Density is carrying the pertinent information for efficient monitoring: precise computations should be done in regions where this value is high and conversely. This domain-specific knowledge is encoded for ACR through the `monitor` pragma to maximize the accuracy of the approximation.

4.2 Applying ACR

To apply ACR to the simulation, the grid state is filled according to the density values. The value of a grid cell is the level of the maximum density point in that cell. The specific target of the optimization by approximation is the portion of the simulation code dedicated to the diffusion phase. Diffusion computations corresponds to a significant part of the total computations of the simulation.

The diffusion phase is computed with a numerical iterative method to obtain a solution. The numerical algorithm gets better solutions as more iterations it does. The original code is programmed to do a fixed amount of iterations in the whole space. To do diffusion, the iterations of the numerical method are done one by one in the complete simulation space. That is because the particles need to know about the solution of its neighbors to compute the next approximation of its own solution. We have used the ACR approach to make the numerical algorithm perform less iterations while maintaining dependencies as much as possible on areas with little amount of fluid or no fluid at all. We have chosen to have 3 levels of complexity for the regions: the optimized algorithm will do the first basic iterations over the entire iteration step, then the other iterations over a more restricted region where there is more than a negligible amount of fluid, and will end doing more iterations only where there is a considerable amount of fluid that needs extra iterations to reach a good enough solution. A simplified excerpt of the corresponding code with the ACR pragmas is shown in Figure 2.

[2]Note: ACR will be demonstrated at the workshop.

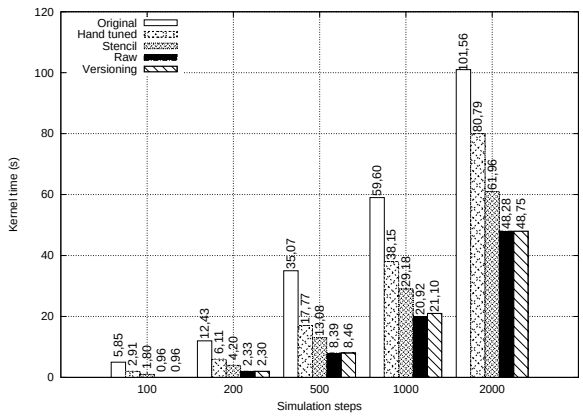
4.3 Experimental Results

ACR was evaluated against a single threaded implementation of a 2D eulerian fluid simulation described by Stam [15] of 400 by 400 particles. We compared the approximate computation code relying on ACR with various grid updating policies (detailed in Section 3.1) against the original code as well as a hand-tuned version that mimics the ACR strategy without dynamically generated code. We observe both performance, computation savings and accuracy over a range of simulation iterations. During simulation, fluid is injected regularly in the iteration space together with a directional force to make it acquire velocity. The test cases have different types of regions with similar appropriate approximation policy and they evolve over time.

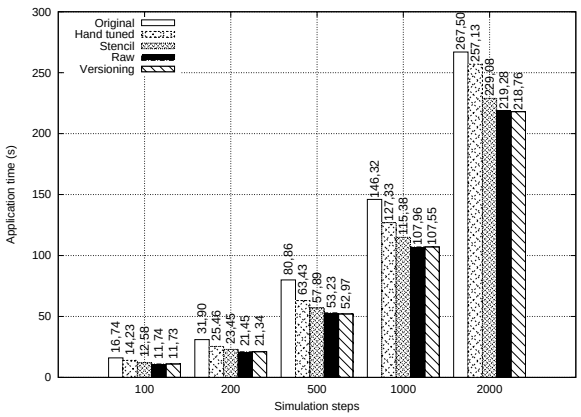
The experimental setup is a quad-core Intel Core 2 Quad Q6600 system with 4 GB of memory. All codes are compiled using GCC 6.2.1 with `-O3 -march=native` option (which builds the best performing original and hand-tuned versions). In addition to GCC, the compilation thread of the ACR runtime also uses TCC (Tiny C Compiler) 0.9.26 to minimize the dynamic compilation time. The grid parameter is set to 40 (10×10 cells), providing a good trade-off between polyhedral generation time and cell size.

Overall execution time is reported in Figure 4 at different steps of the simulation, for the original code, a manually optimized version and the original code optimized using ACR with three different state grid post-processing strategies. The manually optimized code corresponds to a version with the same approximation strategy as ACR, including the same computation of the grid state and the selection of the most appropriate alternative implementation for each grid cell at runtime. However its code is purely static and the alternative selection is done by a switch integrated in the original iteration space which is entirely visited, hence with a high control overhead. In contrast, ACR is generating an optimized code specifically for the current state of the grid state, hence with a minimal control overhead. Performance results after 1000 simulation steps show a speedup of 2.85 for the optimized kernel for ACR with `raw` state grid update strategy (resp. 2.82 for `versioning` and 2.04 for `stencil`) with respect to the original code and 1.82 (resp. 1.81 and 1.31) with respect to the manually optimized code. The `versioning` policy achieves equivalent performance and better precision compared to the `raw` policy: it requires 3% more computations but saves 43% code compilations on average.

It is worth noticing that the performance improvement is only due to the approximation strategy: the generated computation thread is sequential and no other polyhedral optimization has been applied. ACR is complementary to existing optimization techniques and will be composed with them in future work. The present paper only evaluates the benefits of "pure" ACR.



(a) Execution time for the ACR optimized kernel alone



(b) Execution time for the whole application

Figure 4: Execution time for the optimized kernel and the complete fluid simulation application. `Original` is the original application; `Hand tuned` is a manually written version that mimics ACR without dynamically generated code; `Stencil`, `Raw` and `Versioning` are ACR versions with corresponding state grid post-processing policies (detailed in Section 3.1).

Computation savings of the diffusion part for a complete simulation step (where it is used 3 times) with respect to the original code are shown in Figure 5. Our metric for computation savings is the difference of the number of iterations of the original code (corresponding to the number of calls to the `lin_solve_computation()` in the pseudo-code in Fig. 2, i.e., $3 \times MAX \times N \times N$, with MAX set to the same value as the high precision ACR alternative) and the number of iterations actually executed by the computation thread. Results show very significant computation savings compared to the number of computations of the original diffusion, ranging from 73 to 89%.

Iterations	100	200	500	1000	2000
% saved comput.	89.45	88.05	82.72	77.60	73.61

Figure 5: Saved computations with ACR - `raw` policy

Accuracy results are shown in Figure 6. We measured the difference in density for every particle in fixed snapshots of both simulations, with the original application as reference. We measured both the mean and maximum difference of every particle. Values of the density fluid vary between 0 and 20. We observe that even after 1000 iterations and removing 77% of the computations, the mean difference for all particles is only a fraction of the maximum density value. The `stencil` policy shows the best precision property at the price of efficiency as shown in Fig. 4. Moreover, the maximum measured deviation is below 0.05 while in the application, two particles may be displayed using different colors only if they do not belong to the same plateau, each separated by 0.08 and starting at 0. Hence, those results show ACR has a limited impact on the simulation despite its aggressive computation savings.

The results show considerable computation savings and performance improvement while preserving a good accuracy. The speedup is comparable to reported results using loop perforation technique [14] which also skips computations, but for complete iterations. However, building on domain-specific knowledge and on dynamic monitoring instead of a static training makes ACR more accurate and adaptable to different simulations. The dynamic polyhedral code generation approach also clearly outperforms a manual optimization, which is actually a quite complex code while ACR preserves the original program.

Grid update policy	`Raw`	`Versioning`	`Stencil`
Average deviation	0.0017	0.0011	0.0003
Maximum deviation	0.0484	0.0464	0.0247

Figure 6: Precision results for ACR after 1000 iterations (density values range from 0 to 20)

Compiler Techniques The idea of relaxing data dependences and skipping computation to trade accuracy for performance has been studied in the past in various ways. Loop perforation [8, 14] is a static technique which removes complete loop iterations selected with respect to a training phase. Contrary to loop perforation, ACR is a dynamic approach which may remove only selected parts of a given loop execution and is designed to produce more accurate results with end-user guidance. Our work shares the concept of high-level information including alternative implementations provided through pragmas with Green [1], however Green is based on an offline training to select the final code while ACR is continuously recomputing the best code. EnerJ [13] uses the type system to specify approximate variables to save energy. SAGE [12] is a GPU-oriented technique skipping or simplifying processing with respect to performance impact while we primarily focus on accuracy. HELIX-UP [6] ignores some dependences to enable code parallelization, which is complementary to our approach.

Numerical Analysis Techniques ACR has been inspired by numerical analysis techniques, in particular Adaptive Mesh Refinement [3] which can maintain the consistency of a solution for a bounded error in the minimum possible amount of time in simulation problems. [10] extends the use of a multiscale analysis for grid adaptation to incorporate locally varying time stepping. Space filling

curves are also used to transform multidimensional data for better parallelization of multiscale adaptation methods [5]. The differences between our approach and these works is the manipulation of the simulation space since they modify the shape of the simulation space at runtime. Its underlying iteration structure is a hierarchical grid which incorporates and looses points during the simulation. On the opposite, the ACR approach keeps the original iteration space intact while the generated code that achieves the computation may change dynamically.

5. CONCLUSION

In this paper, we introduced Adaptive Code Refinement (ACR), a new compiler technique to improve performance at the price of accuracy. Starting from a reference program and high-level approximation information provided by the user, it generates automatically a program that continuously adapts the optimization strategy to the most appropriate one in regions of the computation space. ACR provides a unique set of features to offer performance, accuracy and flexibility. Performance is provided by building on state-of-the-art polyhedral code generation techniques to generate an optimized code and by exploiting multicore architecture with several specialized threads to minimize the runtime overhead. Accuracy is preserved as much as possible by relying on a dynamic strategy that achieves precise computation only when and where it matters and by preserving the original computation ordering. Finally, flexibility is achieved through a simple yet powerful set of pragmas to drive the approximation strategy, allowing the user to focus on a simple ideal computation kernel while the approximation is managed by the ACR system.

To evaluate ACR, we built a prototype and we used it on an existing fluid simulation application. The experimental results demonstrate significant performance improvement with low accuracy deviation. We also showed that ACR outperforms a manual optimization that would mimic the same dynamic approximation but which, by construction, cannot have an optimized control flow that can only be done by our runtime code generation.

ACR is still in its early days and many studies, improvements and extensions are possible to improve it, including ways to increase the flexibility of the grid, to make the technique even more automatic or to reduce the mechanism overhead. However it is clearly a very promising new way to explore for any application where approximation is either desirable or possible.

6. REFERENCES

[1] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI'10 Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 198–209, Toronto, Canada, June 2010.

[2] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, Sept. 2004.

[3] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.*, 82(1):64–84, May 1989.

[4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI'08 ACM Conf. on Programming language design and implementation*, Tucson, USA, June 2008.

[5] K. Brix, S. S. Melian, S. Müller, and G. Schieffer. Parallelisation of multiscale-based grid adaptation using space-filling curves. *ESAIM*, 29:108–129, Dec. 2009.

[6] S. Campanoni, G. Holloway, G.-Y. Wei, and D. M. Brooks. HELIX-UP: Relaxing program semantics to unleash parallelization. San Francisco, USA, Feb. 2015.

[7] P. Feautrier. Some efficient solutions to the affine scheduling problem: one dimensional time. *Intl. Journal of Parallel Programming*, 21(5):313–348, october 1992.

[8] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical Report MIT-CSAIL-TR-2009-042, MIT, Sept. 2009.

[9] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When polyhedral transformations meet SIMD code generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 127–138, Seattle, USA, June 2013.

[10] S. Müller and Y. Stiriba. Fully adaptive multiscale schemes for conservation laws employing locally varying time stepping. *J. of Scientific Computing*, 30(3), 2007.

[11] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100, Tucson, Arizona, June 2008. ACM Press.

[12] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. Sage: Self-tuning approximation for graphics engines. In *MICRO'13 IEEE/ACM Intl. Symp. on Microarchitecture*, pages 13–24, Davis, California, Dec. 2013.

[13] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *PLDI'11 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, San Jose, California, USA, June 2011.

[14] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *FCE'11 ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 124–134, Szeged, Hungary, Sept. 2011.

[15] J. Stam. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*, 2003.

[16] S. Verdoolaege. *isl*: An integer set library for the polyhedral model. In *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software*, pages 299–302, Kobe, Japan, Sept. 2010.