Automatic Adaptive Approximation for Stencil Computations

Maxime Schmitt
Université de Strasbourg, Inria
Strasbourg, France
max.schmitt@unistra.fr

Philippe Helluy
Université de Strasbourg, Inria
Strasbourg, France
helluy@unistra.fr

Cédric Bastoul
Université de Strasbourg, Inria
Strasbourg, France
cedric.bastoul@unistra.fr

ABSTRACT

Approximate computing is necessary to meet deadlines in some compute-intensive applications like simulation. Building them requires a high level of expertise from the application designers as well as a significant development effort. Some application programming interfaces greatly facilitate their conception but they still heavily rely on the developer's domain-specific knowledge and require many modifications to successfully generate an approximate version of the program. In this paper we present new techniques to semi-automatically discover relevant approximate computing parameters. We believe that superior compiler-user interaction is the key to improved productivity. After pinpointing the region of interest to optimize, the developer is guided by the compiler in making the best implementation choices. Static analysis and runtime monitoring are used to infer approximation parameter values for the application. We evaluated these techniques on multiple application kernels that support approximation and show that with the help of our method, we achieve similar performance as non-assisted, hand-tuned version while requiring minimal intervention from the user.

CCS CONCEPTS

• **Theory of computation → Approximation algorithms analysis**; *Pattern matching*; • **Software and its engineering** → *Compilers*; *API languages*;

KEYWORDS

Approximate Computing, Compilation, Code Optimization, Application Programming Interface, Stencil Code

1 INTRODUCTION

Adaptive approximation aims at providing targeted, relaxed computations while preserving good precision on demanding computation kernels. Automatic compiler adaptive optimization unlocks the possibility to use this kind of optimization not available to a majority of non-expert developers. Adaptive techniques are particularly useful in the context of applications such as scientific computing and data analysis where time constraints are important and the demand for computing capacity grows faster than the available resources. Because they respect language standards, compilers strictly adheres to the program semantics and are not allowed to use approximate computing transformations. Only a limited set of compiler options relax compliance to standards to enable optimization, e.g. considering that some floating point operators are associative may dramatically improve automatic parallelization. In this paper we present a new annotation-based technique that relaxes the program semantics and lets the compiler automatically optimize the annotated code section using adaptive approximation techniques.

Compute intensive applications may require a considerable execution time to produce a result. Approximation techniques provide a solution to trade result accuracy for execution time. Adaptive methods is a powerful class of numerical analysis techniques which targets the approximation in the dynamic regions where their effect on the output solution is minimal. These methods provide strong approximation guarantees while lowering the size of the data and computation to an acceptable level [8]. Adaptive techniques are well suited for applications with localized disturbances such as numerical simulations, signal or data processing. In this work, we exploit the fact that substantial solution variation, which implies that precise computations should be done in the region where it happens, may be detected at runtime.

Using adaptive techniques requires a high level of expertise and a fair amount of application engineering to be used on purpose and efficiently. To open such optimization to a wider audience, we propose a semi-automatic, compiler based, optimization which generates adaptive version of an existing code base. We target Adaptive Code Refinement (Section 3), an application programming interface that provides annotations to express approximate computing transformations (code alternatives) and monitoring capabilities for adaptive methods.

The automatic generation of adaptive code relies on the extraction of the following features:

- The monitoring of the application's data at runtime to identify the region of adaptive interest (Section 4.1). This allows the compiler's runtime to pinpoint the application regions where approximation can advantageously be applied.
- The information about the type of computation performed by the kernel (Section 4.2). This allows the compiler to generate

alternative code versions, by using approximation methods to lower the computational complexity with possible deviation of the result of the application.
- The granularity of the adaptive grid at which the code alternatives are applied (Section 4.3). This allows the detection of coarse or fine grained disturbance regions and select the size that presents the best trade-off between computation gain and the deviation of the application's output.

We use the information gathered by our method to automatically select the best parameters and code alternatives to provide to ACR. In section 5 we demonstrate that our technique can detect useful metrics and use them to generate an adaptive version on a range of representative applications.

In this paper we make the following contributions:

- We propose a novel method to automatically generate adaptive code from target compute intensive kernels.
- We identify the information required by a compiler to generate adaptive codes and provide algorithms to extract them.
- We present a code transformation to automatically generate approximate version for stencil computation kernels, which is one of the main class, if not the main class, of computation kernels in application which supports approximation.
- We provide experimental evidence that our technique is successful at generating adaptive versions which compete with hand-tuned versions.

2 MOTIVATING EXAMPLE

Existing automatic approximation tools [2, 3, 10, 33] are able to search for applicable approximation, but none of them, to the best of our knowledge, supports adaptive techniques. Our work enables automatic adaptive optimization through language annotations and compiler techniques. To illustrate our approach, we consider the main computational kernel of a steady-state heat equation solver [17] shown in Figure 1. An example annotation has been added at line five. This annotation states that the compiler is allowed to generate approximate version for the code block that follows.

The compiler runs both static and profiling analyses on the kernel, here the loop nest, and extracts information required to generate an adaptive version. Three main features are extracted to allow the compiler to generate the adaptive versions automatically:

(1) **Code alternatives** — Generate kernel alternatives with different levels of approximation: for this example, the compiler detects that the kernel uses stencil computation for which it is able to generate different versions with different approximation levels.
(2) **Adaptive decision** — Find a metric to decide at runtime about the convenient level of approximation to use in a given subset of the computation space: in this example, there is only one written array in this code, hence the deviation of its values drives the selection of the alternative kernels (low deviation corresponds to low precision and high deviation corresponds to high precision).
(3) **Decision granularity** — Select the cell size when decomposing the computation space into a regular grid: profiling information allows to select the convenient cell size to get the best precision/performance trade-off.

We use the ACR programming interface (Section 3) to generate the adaptive code from the extracted high level adaptive features. Section 5.1 shows our technique applied on this example, along with the generated ACR annotations.

During the execution of the optimized program, a runtime system selects, for each cell, the best suited alternative with respect to the dynamic data values, i.e., achieving precise computation only where it matters. In this example, we reduced the number of floating point computation by 74.7% and achieve a speedup of 1.81 with a deviation below 10^{-4}, see Section 5 for the complete experimental study.

3 ADAPTIVE CODE REFINEMENT AND EXTENSIONS

Adaptive Code Refinement (ACR) is a recent language and compiler approach proposed by Schmitt et al. which provides a simple interface for developers that want to exploit approximate computing techniques [39, 40]. Our goal is to demonstrate that our technique, when applied to ACR, leads to an automation of this tool. In this section we present ACR's use case, its user interface, the underlying compiler infrastructure and the code transformation and generation mechanism that produce alternative versions of the source code automatically. We introduce a new annotation and special construct parameters that triggers the automatic discovery of pertinent approximations and ACR parameters.

ACR targets applications with a compute intensive kernel that updates data multiple times, e.g., numerical simulation or video frame encoding. In such applications, a relation exists between the data used by two consecutive kernel calls. ACR exploits this property to reduce the computation intensity locally, where the expensive update is known or expected to to have little effect on the result.

3.1 ACR's Application Programming Interface

ACR framework relies on supplementary semantics provided by the developer through code annotations. These annotations can be inserted at hot spots of the application and allows the compiler to apply additional code transformations, related to approximate computing. The set of annotations is implemented for the C language using compiler directives (#pragma acr). The following compiler directives are provided by ACR to annotate the input source code:

`#pragma acr grid(Size)`
: The grid construct provides the ability to break down the original problem data space into hyper-squares shaped chunks of the size provided as input. Each of these chunk is called "a cell". A cell represents the atomic element over which transformation and alternative computation selection are performed.

`#pragma acr monitor(Array, Folding_function)`
: The monitor construct specifies the application's relevant data to the compiler for ACR. This data will be used by the decision mechanism, to drive the dynamic selection of the approximate alternative computation. The folding function is a way for the compiler to extract the information from a cell. This function summarizes the data cell into one unique observation. An observation is defined as a natural number whose value represents the level of approximation that a

```c
double solve_jacobi(size_t sizeX, size_t sizeY,
                    double t[sizeX][sizeY],
                    double tNext[sizeX][sizeY]) {
  double max_delta = 0.;
#pragma asa auto (1e-4)
  for (size_t i = 1; i < sizeX - 1; ++i)
    for (size_t j = 1; j < sizeY - 1; ++j) {
      tNext[i][j] = 0.25 *
        (t[i - 1][j] + t[i + 1][j] +
         t[i][j - 1] + t[i][j + 1]);

      double delta = t[i][j] - tNext[i][j];
      delta *= delta;
      max_delta = delta > max_delta ?
                  delta : max_delta;
    }
  return sqrt(max_delta);
}
```

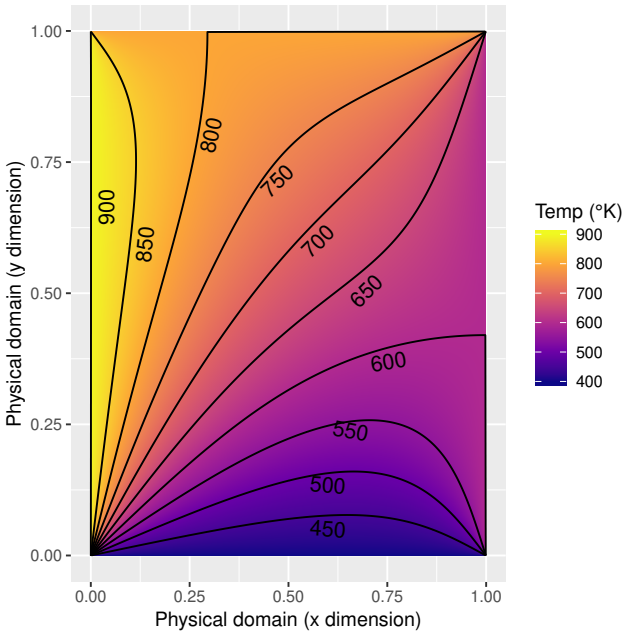


Figure 1: Heat conduction solver using the Jacobi method. The left part of the figure shows the kernel which updates the temperature and computes the maximum temperature change during an update step. The solver is executed repeatedly until the simulation settles, which happens whenever the maximum temperature delta is less than a given threshold, e.g., in this example the maximum difference is less than 10^{-4} degrees Kelvin. An example simulation output is displayed on the right part of this figure. In this simulation the border conditions are set to a constant temperature.

cell should withstand. An observation with a value of zero means that no approximation is allowed.

`#pragma acr alternative AltId(Type, TParam)`

The alternative construct defines possible alternative computation to be generated by the compiler. The alternative computation is generated from the original kernel by application of specific code transformations. The type of supported transformations are:

Parameter modifies the value of a constant parameter with a constant value or another parameter

Code allows the developer to provide an alternative code block

Zero-compute treats the target code block as if it is empty

Interface-compute disables computation that do not contribute to update the data located on an interface between two grid cells

Some alternatives take a parameter here represented as `TParam`, e.g., `parameter type` accepts a variable identifier or a constant value. The user provides a unique alternative identifier using `AltId`. This identifier is used within the strategy construct to select the conditions required to use this alternative.

`#pragma acr strategy Type (SParams, AltId)`

This construct defines the compiler's strategy, i.e., the necessary conditions for which an alternative should be active. The strategy may be one of the following types:

Static: the alternative will unconditionally be active. The user can choose whenever to apply it on localized portions or the whole data space.

Dynamic: the alternative will be active depending on runtime observation of the monitored data defined with the `monitor` construct. This strategy applies at the cell granularity. A cell will use the alternative if the folding function returns the same observation as the one defined by this strategy.

In addition to existing ACR pragmas, we propose the following new Automatic Stencil Approximation (ASA) constructs that allows the compiler to automatically generate ACR annotations with pertinent alternatives and parameters:

`#pragma asa auto (AllowedPrecisionLoss)`

This construct replaces all the previously mentioned constructs. It asks the compiler to find the best possible alternatives and parameters for the target kernel.
`AllowedPrecisionLoss` specifies the maximum deviation for which approximate computation may be used.

`#pragma asa interactive (AllowedPrecisionLoss)`

This construct is similar to auto but lets the user select between a range of compiler-proposed alternative and parameters.

These new constructs rely on compiler facilities, here ACR, and additional information to generate the approximate versions with little intervention of the user.

3.2 ACR's Compiler Infrastructure

With the additional semantic information provided by the user annotations, ACR can generate a statically or dynamically optimized program. The static version is generated at compile time and the dynamic version relies on just-in-time compilation to generate a specialized version of the kernel. ACR allows for a fixed Cartesian grid overlay to delimit the cells (cf. the grid annotation in Section 3.1). The dynamic version allows the compiler to optimize a kernel where dependencies forbid the static transformation or when the user instructs the compiler to select the grid size dynamically.

ACR relies on the polyhedral representation of programs to apply transformations and generate an optimized version of the code. Figure 2 discloses code examples generated from the static and dynamic ACR code generators. During the execution of the program, the monitoring extracts the observation of each cell to decide which alternative should be used. The dynamic runtime uses this information to build a kernel with low control overhead following the same execution order as the source program (Figure 2b). The static version uses a guard statement and data collected during execution to switch between the different alternatives for each cell. Hence, the execution order is modified and a cell is visited entirely before continuing to the next one (Figure 2a).

ACR's dynamic runtime uses a set of threads, each with a specific role. It consists of: a **monitor thread** that is responsible to extract for each cell, from raw data, the alternative to use. A **polyhedral code generation thread** that utilizes the information from the monitor thread to generate an optimized kernel with low control overhead.[1] A **compilation thread** that takes the output of the code generation stage and compiles it to a binary format. Finally, a **coordinator thread** which is responsible to organize the other threads, schedule their execution and inject the optimized version to be used by the application once it is ready.

4 APPROXIMATION EXTRACTION

Using approximate computing either requires deep knowledge of the user in this field or, if automated, ask the user to delegate the transformation to the compiler. Each of these two techniques have their pros and cons. An expert developer will select the best approximation technique suitable for his problem. On the other hand, an automatic method may discover potential approximation targets using binary error injection, using genetic/machine learning algorithms or using pattern matching to find a suitable approximation strategy to be applied to the original algorithm [2, 10, 35]. The goal is to lower the program's computation footprint while keeping the program's output deviation below a user-defined threshold. The deviation is defined as the l^∞ norm of the difference between two states corresponding to the same data, i.e., $max(|\mathbf{state}_i - \mathbf{state}'_i|)$.

In this section we provide a new method that automatically extracts adaptive approximation information from a targeted portion of a program. Our method also allows interaction between the user and the compiler to select the best alternatives. Automatic stencil approximation (ASA) relies on the polyhedral model to analyse the code. The polyhedral model abstracts a set of statement having static control (SCoP) [4]. This includes the class of program with loops, conditional statements and data accesses which depend on affine expression of constant parameters and outer-loop variables.

[1] Note that this does not involve, but is complementary to, polyhedral optimizations techniques such as parallelization or data locality optimization.

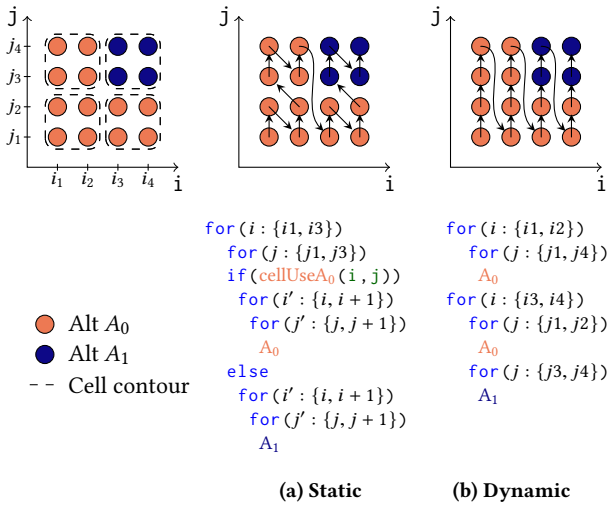


Figure 2: Example of static and dynamic code generated on a four-cell example. Three of these cells require the alternative A_0 and one requires the alternative A_1. The static version, which is generated at compile time, visits a cell entirely before going to another. A guard is needed to select the alternative before entering a cell. The dynamic version generates a specific code at runtime, which follows the application original schedule shown on the top right.

4.1 Precision Level Discovery

The first semantic information required by the compiler to use approximate techniques is related to the alternative selection mechanism, i.e., in which conditions do we allow a less precise versions of the algorithm to be used. We suppose that the approximate alternative code versions can be ordered according to their level of approximation, e.g., $V_1 \prec V_2 \prec V_3$ means that V_1 is more precise than V_2 that is more precise than V_3, representing a total order. There will be two or more versions, the original code represented as V_1 and its alternatives (Section 4.2). In this section we present an algorithm to find a function to select the alternative versions at runtime.

The input data of the decision function is the application's runtime data, representing the state of the problem that the user has to solve. The compiler has to detect which data has the convenient characteristics to be able to generate a function to select an alternative globally or on a localized portion of the application's data domain. We propose a set of characteristics extracted from our study of typical applications from the field of approximate computing.

The first characteristic which usually provides relevant precision information from the application data is the deviation between the updates. Mathematically, this characteristic represents the first derivative of a function [12]. Whenever this rate is measured to be low, it is a good sign that we could use a less precise update function in this area of the application's data domain. Furthermore, this effect is usually local, and in numerical simulations, singularities spread

and do not appear in the middle of uniform regions, hence, missing a significant event is unlikely. If the effect is localized, we consider clusters of data as a single entity and apply the alternative to each updates of this cell (more in Section 4.3).

To discover this first characteristic the compiler has to look for the following patterns in the source code:

Swapping arrays They are frequently used in physic simulation and image processing where the transformation uses the data from the previous transformation as input. This characteristic is statically analyzed if the program is simple enough or dynamically asserted by running the application on datasets and monitoring the array addresses accessed and recognize a "n cycle" pattern.

Write-only arrays They are usually the sign of a multi-stage update algorithm, e.g., any advection-diffusion equation which updates the physical variables in multiple steps/functions or store statistic information of the application's state. This is addressed with a static analysis of the kernel on written-only data arrays.

A second characteristic, lie in the application's own statistic gathering, e.g., multimedia codec gathers information about previous frames to achieve a better compression ratio, and numerical simulations may use a physical variable or error rate as a simulation stopping condition. Statistics are usually gathered globally and have smaller size than the application's dataset. The application may gather the statistics in a dedicated function which could be marked by the user in the same fashion as the kernel to optimize. Hence, the compiler can extract correlations between the data updated by the kernel and the statistics gathered by the application. In this paper we only consider statistic update located inside the kernel to optimize. Our statistics gathering follows the following pattern:

Accumulators They can be used to store statistic information and are most likely used by the application to take algorithmic decisions. Such accumulators are of particular interest for our method as they may store the information we are interested in. Accumulator are many-to-one information gathering. The compiler may allocate memory to save this information at a finer granularity if needed.

With these information, the compiler builds the most suitable precision discovery function by monitoring the output of the different functions at runtime. It may ask the user for advice depending on the annotation the user selected, i.e., either interactive or automatic.

In addition to explicitly selecting the target computation kernel, the user inputs the level of deviation under which it becomes acceptable to use approximation, whereas the compiler generates N alternatives. In that case, the compiler needs to affect a deviation level, related to the alternative approximation level, for which each alternative will be selected. Assuming that lower deviation allows for even more aggressive approximation, we can affect decreasing deviation values to more approximate alternatives. The alternative deviation values may be generated following a negative linear or exponential slope. We rely on an exponential slope to increase the density of alternative in the neighborhood of the user's defined deviation threshold.

Algorithm 1: Example 2D stencil algorithm

```
1 def Five points stencil:
      Data: Input array I[M][N]
      Result: Output array O[M − 2][N − 2]
      for i ← 1 to M − 1 do
          for j ← 1 to N − 1 do
              O[i][j] ←
               I[i][j]⊕I[i−1][j]⊕I[i+1][j]⊕I[i][j−1]⊕I[i][j+1];
2 def Five points stencil with two iterations merged:
      /* operation · distributive over ⊕          */
      Data: Input array I[M][N]
      Result: Output array O[M − 2][N − 2]
      for i ← 1 to M − 1 by 2 do
          for j ← 1 to N − 1 do
              O[i][j] ← ( 5 · I[i][j] ) ⊕
               2 · ( I[i + 1][j] ⊕ I[i][j − 1] ⊕
               I[i][j + 1] ⊕ I[i − 1][j − 1] ⊕
               I[i + 1][j + 1] ⊕ I[i − 1][j + 1] ⊕
               I[i − 1][j] ⊕ I[i + 1][j − 1] ) ⊕
               I[i − 2][j] ⊕ I[i + 2][j] ⊕ I[i][j − 2] ⊕ I[i][j + 2];
```

4.2 Automatic Alternative Stencil Generation

Alternatives are different versions of the application's kernel which uses approximation techniques to reduce the overall kernel calculation and data access intensity. Such versions suffer a precision loss compared to the original algorithm. In this section we present a technique to identify pertinent approximate computing transformations, generate the alternative versions and classify them by level of approximation.

Stencil code are a class of iterative kernel that uses nearest-neighbor computations on a grid or graph data structure, i.e., the update of a point on the grid depends on the adjacent neighbor values according to a fixed pattern. Stencils are extremely commonly found in scientific computing, machine learning and image processing applications. E.g., Algorithm 1 shows a five point stencil kernel on a 2D Cartesian grid. The compiler can rely on a static data access analysis in order to automatically detect the most common form of stencils. In our case, we are interested in data read and write accesses for each statement of the kernel to detect stencil-like computations. We provide Algorithm 2 to identify the stencils. This algorithm builds for each written data at a given iteration, the set of read data that contributes to the computation of the written data (both inside a statement or transitively through several statements). It considers each write statement in the lexicographical order and selects the one with the most reads to the same array. It outputs the set of written + contributing data, with maximum cardinality.

We propose the following adaptive stencil alternatives which reduces the stencil computation and data accesses:

Skipping Skipping some stencil computation entirely (e.g. skip it every N iteration of the computing loop) when the precision function has monitored a low deviation.

Algorithm 2: Search the biggest stencil in a statement set

Data: Ordered statement set S
Data: For each $S_i \in S$, read access set S_i^{read}
Data: For each $S_i \in S$, write access set S_i^{write}
Result: Tuple of written and read location of a stencil

$\mathtt{Biggest}_{\mathtt{read}} \leftarrow \emptyset$;
$\mathtt{Biggest}_{\mathtt{write}} \leftarrow \emptyset$;
$n \leftarrow \|S\|$;
while $n > 0$ **do**
  /* Statement accesses + transitive accesses */
  $N^{\mathsf{read}} \leftarrow S_n^{\mathsf{read}}$;
  repeat
    $N_{\mathsf{prev}}^{\mathsf{read}} \leftarrow N^{\mathsf{read}}$;
    foreach $N_i \in N_{\mathsf{prev}}^{\mathsf{read}}$ **do**
      foreach $W_i \in \mathtt{statementWrite}(S, N_i)$ **do**
        $N^{\mathsf{read}} \leftarrow N^{\mathsf{read}} \cup W_i^{\mathsf{read}}$;
  until $N^{\mathsf{read}} = N_{\mathsf{prev}}^{\mathsf{read}}$;
  /* Check for stencil pattern for each arrays */
  $G \leftarrow \mathtt{groupBy}(\mathtt{array}, N_{\mathsf{read}})$;
  $\mathtt{BestLocalMatch} \leftarrow \mathtt{maxCardinal}(G)$;
  if $\|Biggest_{\mathtt{read}}\| < \|\mathtt{BestLocalMatch}\|$ **then**
    $Biggest_{\mathtt{read}} \leftarrow \mathtt{BestLocalMatch}$;
    $Biggest_{\mathtt{write}} \leftarrow S_n^{\mathsf{write}}$;
  $n \leftarrow n - 1$;
return $(Biggest_{\mathtt{write}}, Biggest_{\mathtt{read}})$
`statementWrite(S, N)`: returns the set of statement in S which writes at the location N.
`groupeBy(P, S)`: returns a set of set grouped by P
`maxCardinal(S)`: returns the largest cardinal set in S

Narrowing Reduce the size of a stencil to reduce both its calculation and memory load.

Border Activation Only activate the stencil computation at the cell borders, skipping the computation in the center of cells while allowing information to pass through the interfaces with their neighbors. Cells having a neighbor with an alternative more precise than theirs will use border activation to capture a possible incoming singularity.

Stencil narrowing may rely on the fusion of multiple stencil steps to grow the size of the halo. For example, the stencil in Algorithm 1 top has a halo of distance one. It is possible to increase the size of this stencil halo by merging two stencil loops, resulting in the bottom stencil in Algorithm 1. A larger halo offers more possibilities to build an approximate version. The halo could be reduced by redistributing the stencil weights of the outside most point of the halo into the remaining weights (narrowing). Another method consists on keeping only one stencil update every N steps (skipping).

The analysis of deviation between updates allows adaptive methods to target the expensive computations in part of the domain where significant updates are expected [5, 20]. Hence, our algorithm relies on this feature to apply the previously mentioned alternatives.

Stencil skipping allows the compiler to generate a version that reduces the computation intensity, hence the precision, by a factor of N. Stencil narrowing allows to generate a wider range of approximate versions, not available with skipping only. For example, it is possible to remove P points of a stencil with the lowest absolute factors to lower the computation/precision. If the number of points removed is below what stencil skipping deletes from a level N to $N-1$, it corresponds to fully generated intermediate approximate versions. Stencil merging combined with these techniques allows to generate enough alternatives for an adaptive purpose. In the current state of our implementation choosing a high number of alternative leads to a high overhead, consequently generating only a limited number of versions gives the best results.

4.3 Granularity Selection

At this point, the compiler has decided upon one precision feature and multiple associated alternatives. It has yet to decide at which granularity to take the decision. A fine grid will have a better mapping to the application's problem but will have a higher managing overhead whereas a grid that is too coarse may miss approximation opportunities. In this section we present an algorithm to select a pertinent granularity based on an empirical study of a set of applications representative of the approximate computing field.

State of the art adaptive techniques use modular grids, i.e., the grid shape is updated during execution to focus the computation where it is needed. A modular grid creates special cases at the interface of two cell with different precisions. These cases need to be handled by the compiler and complicates the automatic alternative and grid generation. In this work we only consider Cartesian, static grid. We discovered that the application's behaviour to the grid size can be transposed between distinct input problems and data sizes.

Figure 4 shows the three main recognizable patterns seen while monitoring different applications. All of them show a local minimum but with a different distribution of performance. Applications that show one of these three behaviours will present the same behaviour with different data sizes or problem statements (see Section 5). Hence, we propose to match these patterns to categorize the application among one of these three forms and compute the ratio between the maximum grid size and the value read at the local minimum. Our empirical study suggests that this ratio can be reused with different data sizes to scale the grid size accordingly. Figure 3 shows that such algorithm is precise enough to approach the best value for bigger dataset based on data collection on the small dataset. Hence, the compiler has an estimate of the best grid size for any dataset size and the curve indicating the slope direction.

With the current static-grid back end, we cannot guarantee the precision level of the output with respect to any input data. The guarantee we provide is that no approximated version is used when the monitored deviation is greater than the one selected by the user. The user must provide input data to the optimized application within the same granularity scaling than the one used during profiling. The user should run a new automatic profiling step for different input value scales. Ongoing work aims at removing this requirement using dynamic grids and mathematical transformations.

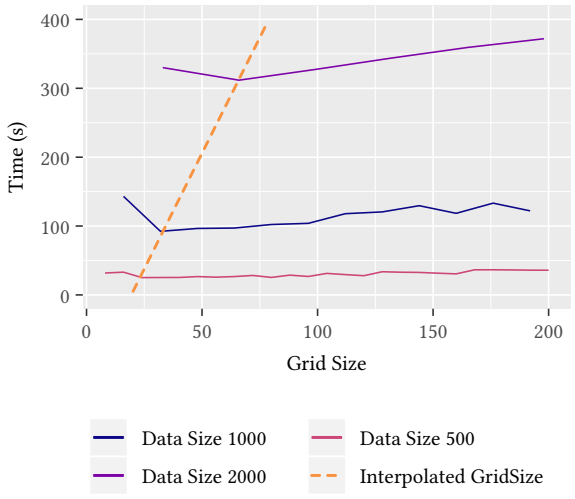
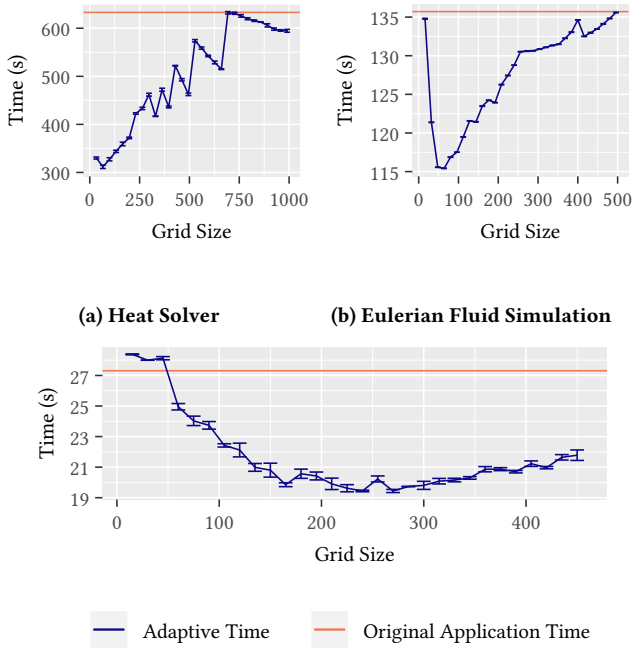


Figure 3: Heat solver time in function of the grid size, execution with multiple dataset sizes. The grid size that performs the best can be interpolated using a linear regression.



(c) Finite Difference Time Domain

Figure 4: Running time of three optimized applications as a function of the grid size. Each application is representative of a class of programs sharing a similar behaviour when increasing the grid size. In a given class, we could observe that the behaviour is the same independently of the dataset size or initial conditions of the simulation.

4.4 User-Compiler Interaction

Our ASA compiler takes as input a file containing the source code with a kernel annotated with an ASA annotation presented in Section 3.1. Its output is a new file where the kernel has been annotated with the relevant ACR pragmas with relevant parameters, and where new functions corresponding to new code alternatives have been added if necessary. A user wishing to interact with the compiler, for any of the three features presented in the sections 4.1 4.2 and 4.3, can investigate the generated file to see which approximations have been built. The compiler accepts user ACR annotations in addition to the ASA annotation and they always prevails over the compiler decisions. This allows an advanced user to provide specific knowledge to the compiler. The user can also access the data generated by the compiler, as the monitoring information from the applications (see Figure 4) and output deviation. The interaction with the optimizing compiler allows the user to have a high-level feedback about the monitoring data, the approximation strategy and the adaptive grid size.

5 EXPERIMENTAL STUDY

In this section we present the experimental results achieved by our method on application compute intensive kernel on the set of benchmarks used to evaluate ACR [39] with the addition of the heat equation simulation. This benchmark set includes typical codes that are resilient to approximation. We used a source-to-source approach and targeted the annotations presented in section 3. Because of the runtime overhead of managing many alternatives, we restricted our study to the generation of three alternatives (precise, approximated and border activation). This comes without loss of generality, but we may expect even better results using future ACR compiler versions with smaller runtime overhead. The experimental setup consists of an Intel Xeon E5-2620 with 16 gigabytes of ram. We use the ACR development version[2] with GCC 8.2.

5.1 Heat Equation

This application, introduced in Section 2, is a solver for the steady state heat equation. It is used for material applications and physics simulations [17]. The main computational kernel of this application is shown in Figure 1, left.

This application is a good candidate for optimization using approximate computing techniques. By profiling the application, a developer may know that the kernel is the biggest contention part of the program. Hence, this developer may add the annotation present at line five: #pragma asa auto (1e−4). This annotation states that the following kernel can be optimized by the compiler to take advantage of adaptive techniques. In the following of this section, we explain the compiler process to extract the information necessary to generate the approximate computing version.

Heat Equation Precision Level Discovery The compiler generates then executes an instrumented version of the program on a small user-provided dataset to gather information about the array accesses at runtime. In this program, it happens that the kernel's

[2] http://gauvain.u-strasbg.fr/%7Eschmitt/acr

written array switches between two addresses.[3] The compiler can automatically generate the comparison function as there are two arrays to compare. The developer already expressed it's intention to allow approximation when deviation is below 10^{-4}. Therefore, the compiler generates the following ACR annotation to select the precision level:

#pragma acr monitor(tNext[i][j], min, diffT)

Where the `min` function selects the inside cell alternative of the most precise version and diffT is the pre-processing function that embodies the computation of the deviation between the value of arrays `tNext` and `t`. The monitor compares the deviation to 10^{-4} and returns zero if the original code has to be executed or one if approximation is acceptable, allowing for one alternative along the original code.

Heat Equation Alternative Generation Algorithm 2 detects the stencil pattern. This stencil being too narrow for further optimization, the compiler modifies the kernel to merge two consecutive stencil computations to increase the width of the stencil, creating a loop where the iteration count is defined by a new variable `kernel_iter` (the value 2 corresponds to the original kernel). A stencil skipping alternative can be created by modifying the number of iterations of the kernel:

#pragma acr alternative alt0(parameter, kernel_iter=2)
#pragma acr alternative alt1(parameter, kernel_iter=1)

ACR requires additional annotations to link the alternatives to the value obtained by the monitoring. Therefore, the following annotations link the monitoring folded value zero to the original computation and the monitoring folded value one to the early-terminated number of kernel iterations:

#pragma acr strategy dynamic(0, alt0)
#pragma acr strategy dynamic(1, alt1)

Heat Equation Grid Selection The compiler tests the generated alternative on a small user-provided dataset to assess the viability of this alternative. The compiler requires a metric to evaluate the level of deviation of the result of the alternative. The preferred metric for ACR is the relative difference or error with respect to the original application's output [43]. The compiler uses this metric to compare the result of the alternative version against the original application and requires that the difference be less than five percent in total. The application is evaluated w.r.t. multiple grid sizes to obtain the data plotted in Figure 4a. Using this curve, it is possible to assess the ratio corresponding to the local minimum that will be used for other datasets, here $\frac{24}{500}$ which will be multiplied to the maximum between $sizeX$ and $sizeY$ to compute the new grid size.

Heat Equation Automatic Method Evaluation Table 1 shows the speedup obtained with our algorithm and the best value obtained by a search over all possible grid values. Results show both the benefits to exploit adaptive technique for this benchmark and that the automated process achieves a performance close to the best hand tuned version (see Section 5.6).

[3]The corresponding statement is flagged using a specific annotation in our compiler prototype, but it may be discovered automatically as this is a well known pattern in such codes.

5.2 Eulerian Fluid Simulation

Figure 5 shows a kernel presented by Stam [42] which solves the Navier-Stokes equation using an iterative solver. This solver must be fast and the precision accurate enough to be human imperceptible as it is targeted for real time video applications or games.

```
1  #pragma asa auto (1e-2)
2  for (unsigned k = 0 ; k < P ; ++k) {
3    for (unsigned i = 1; i < M-1; ++i)
4      for (unsigned j = 1; j < N-1; ++j)
5        x[i][j] =
6          ( x0[i][j] +
7            a * (x[i-1][ j ] +
8                 x[i+1][ j ] +
9                 x[ i ][j-1] +
10                x[ i ][j+1])
11         ) / c;
12   set_bnd(M, N, x);
13 }
```

Figure 5: A Gauss-Seidel iterative solver of linear equations.

Fluid Simulation Precision Level Discovery In this kernel, the compiler has assessed that the arrays x and $x0$ are swapped between each call to the kernel. Hence, it generates a function that will compare the two arrays to compute the update function derivative:

#pragma acr monitor(x[i][j], min, diffX)

Fluid Simulation Alternative Generation The compiler recognizes a stencil in the two inner dimensions of the kernel. This stencil repeats P times during the execution. Therefore, the compiler does not have to widen the stencil size but only has to reduce the number of stencil computation to generate an alternative:

#pragma acr alternative alt0(parameter, k=P)
#pragma acr alternative alt1(parameter, k=1)
#pragma acr strategy dynamic(0, alt0)
#pragma acr strategy dynamic(1, alt1)

Fluid Simulation Grid Selection Figure 4b shows the application's behaviour to the grid size. The best ratio extracted from the curve is $\frac{60}{500} \times \max(M, N)$.

5.3 Cellular Automaton

Game of life is cellular automaton which is ruled by three easy rules:

- A cell *becomes* alive if there is at least three alive cells in its direct surrounding at the previous time step.
- A cell *stays* alive if it had two or three alive neighbours at the previous time step.
- Otherwise a cell is considered dead at the next step.

This automaton has recurring patterns and empty areas that creates potential for optimization [18]. It is possible to create an adaptive version that does little computations in the empty zones. The main kernel of our benchmark implementation is shown in Figure 6.

Cellular Automaton Precision Level Discovery The compiler detects that the written array is switching between two addresses and will use it to compute the derivative.

#pragma acr monitor(current_grid[i][j], min, diff)

```
1  #pragma asa auto(0)
2  #pragma acr alternative \\
3              alt_interface(interface_compute)
4  for (int i = 0; i < nb_row; ++i) {
5    for (int j = 0; j < nb_col; ++j) {
6      int num_alive = 0;
7      for (int k = i - 1; k <= i + 1; ++k) {
8        for (int l = j - 1; l <= j + 1; ++l) {
9          num_alive += previous_step_grid[k][l];
10       }
11     }
12     if (previous_step_grid[i][j])
13       num_alive -= 1;
14     }
15     switch (num_alive) {
16     case 3:
17       current_grid[i][j] = 1; break;
18     case 2:
19       current_grid[i][j] =
20         previous_step_grid[i][j]; break;
21     default:
22       current_grid[i][j] = 0; break;
23     }
24   }
25 }
```

Figure 6: Game of Life algorithm to update the state of the cell grid from one generation to the next one.

Cellular Automaton Alternative Generation To generate a non-approximate stencil version which allows the simulation cell states to spread while disabling the empty regions, the user decides to add the `interface_compute` alternative which activates the neighboring ACR cells in the locations where the simulation cells are active. The user can help the compiler to select an alternative by adding a back-end annotation, here an ACR alternative shown in Figure 6 lines two and three. The compiler can then generate the link between the monitoring and the added alternative:

```
#pragma acr alternative alt1 (zero_compute)
#pragma acr strategy dynamic(2, alt1 )
#pragma acr strategy dynamic(1, alt_interface )
```

Cellular Automaton Grid Selection This application follows the same pattern as the heat solver Figure 4a. The computed best grid ratio over data size is $\frac{275}{8000} \times \max(nb\_row, nb\_col)$.

5.4 Finite Difference Time Domain (FDTD)

Finite-difference time-domain implements a solver for the Maxwell-Faraday equations of electrodynamics differential equations [29]. The kernel Figure 7 shows the update of the value of the electric plane E as a function of the magnetic field H. This application's singularity resides in its two-phases algorithm, the electric field update followed by the magnetic field update.

FDTD Precision Level Discovery This kernel updates two arrays at the same time, Ex and Ey. The array pointers are not modified during the execution of the program. The compiler arbitrarily chose to monitor Ex (in such case, it is likely that the two updates are correlated, minimizing the impact of that choice: there is actually a correlation which can be statically analysed in this benchmark). The ACR annotation generated is:

```
#pragma acr monitor(Ex[i][j], min, diff )
```

```
1  #pragma asa auto (1e-3)
2  for (unsigned i = 1; i < I-1; ++i) {
3    for (unsigned j = 1; j < J-1; ++j) {
4      Ex[i][j] += alpha_Ex *
5                  (Hz[i][j] - Hz[i][j-1]);
6      Ey[i][j] += -alpha_Ey *
7                  (Hz[i][j] - Hz[i-1][j]);
8    }
9  }
```

Figure 7: Kernel of a FDTD update loop of the electric and magnetic field in a 2D space.

FDTD Alternative Generation For this application, the stencil merging technique could not help because the arrays are not swapped and the array Hz is not written inside the kernel. Hence, the only remaining possible alternative optimizations are stencil skipping and narrowing:

```
#pragma acr alternative alt1 (zero_compute)
#pragma acr strategy dynamic(1, alt1 )
```

FDTD Grid Selection Figure 4c shows the application's runtime with respect to the grid size. For this application, the maximum grid size is 3000 and the best grid size is located in the plateau between 200 and 300 with $\frac{240}{3000} \times \max(I, J)$

5.5 K-means clustering

K-means is used for data characterization, e.g., image processing, and machine learning [24]. Its purpose is to place N observations into B buckets where the observations being the closest will appear in the same bucket or centroid. The main kernel of the application is shown in Figure 8. Firstly, the algorithm places the observations into the closest bucket using a comparison function. Secondly, the barycenter of the bucket is updated with the observations belonging to it. The algorithm caries on until all the observations settles.

K-means Precision Level Discovery The only available array to monitor is `point_centr_map`:

```
#pragma acr monitor(point_centr_map[pos], min, diff )
```

K-means Alternative Generation Having access to the whole loop, the compiler uses the stencil merging technique to reduce the iteration count by half in the approximated regions by applying stencil skipping:

```
#pragma acr alternative alt0 (parameter, kernel_iter =2)
#pragma acr alternative alt1 (parameter, kernel_iter =1)
#pragma acr strategy dynamic(0, alt0 )
#pragma acr strategy dynamic(1, alt1 )
```

K-means Grid Selection We used the K-mean algorithm to characterize images of different sizes, ranging from 640×425 to 4288×2848. The best ratio extracted for this application is $\frac{90}{272000} \times points$.

5.6 Performance Evaluation

In this section we compare the application's performance against the original code version and a hand-tuned version using the ACR annotations. Table 1 shows the performance and deviation of the output results of our method against user search of the parameters. We can see that, while the performance of AMA is lower than what a specialist can extract with the ACR annotations, our technique

```
do {
  has_converged = true; // Assume convergence
#pragma asa auto(0)
  for (size_t pos = 0; pos < points; ++pos) {
    unsigned new_centr = 0;
    float new_centr_dist = dist(point[pos],
                                centr[0]);
    for (unsigned i = 1 i < num_centr; ++i) {
      float centr_dist = dist(point[pos],
                              centr[i]);
      if (centr_dist < new_centr_dist) {
        new_centr = i;
        new_centr_dist = centr_dist;
      }
    }
    if (point_centr_map[pos] != new_centr) {
      has_converged = false;
    }
    point_centr_map[pos] = new_centr;
  }
  update_centroid_barycenter(centroid,
    num_centroid, data, point_centr_map);
} while(!has_converged);
```

Figure 8: K-Means core algorithm where the observations points are placed into the clusters and the center of the cluster is updated.

Application	ASA Speedup / Output Deviation	Hand ACR Speedup Output Deviation
Heat	1.25	Training set
Heat 1000	1.68 / < 0.001%	1.76 / < 0.001%
Heat 2000	1.81 / < 0.001%	2.03 / < 0.001%
Fluid	1.1 / 0%	Training set
Fluid flame	1.17 / 0.8%	1.27 / 1.51%
Fluid vortex	1.16 / 0.01%	1.19 / 0.01%
FDTD	1.12 / 0%	Training set
FDTD 500	1.16 / 0%	1.22 / 0%
FDTD 3000	1.24 / 0.1%	1.4 / 0.5%
GOL	1.44 / 0%	Training set
GOL big	2.09 / 0%	2.2 / 0%
K-means	2.14 / 3.58%	Training set
K-means med	1.51 / 0.7%	2.50 / 2.62%
K-means big	2.18 / 9.03%	1.57 / 4.48%

Table 1: AMA and hand tuned ACR version performance and deviation of the output results.

achieves sensible performance gains often close to hand-tuned versions. The deviation of the output is similar and sometimes better with with the hand optimized versions and well below five percent, besides one over-optimization with K-mean where the automatic method reached nine percent deviation while the hand tuned version was selected with a bigger grid size which allowed for less approximation.

ACR uses a multi-threaded infrastructure to generate and compile the adaptive versions in parallel to the program execution. This requires resources which may not be exploited due to Amdahl's law [19], or be better exploited with AMA. Furthermore, adaptive methods are complementary to parallelization techniques, and information from the adaptive grid may be used for a better resource scattering.

6 STATE OF THE ART

Relaxing Semantics Relaxing the program's semantics provides new optimization potential for the compiler. Precimonious is a tuning assistant which uses this strategy to selectively lower the precision of floating point arithmetics (e.g., double to float) to optimize memory traffic and computation time [34]. HELIX-UP chooses to ignore dependencies and synchronizations to increase parallelism in applications [9]. SAGE and Meng's et al. work allows for better parallelism on GPU architectures by targeting GPU specific approximate optimizations and using a best effort computing model respectively [24, 36]. Loop perforation technique allows the compiler to skip loop iterations to gain in performance [41]. Task-based workflows can be approximated by skipping certain task probabilistically [32]. Rhahimi et al. explored memoization techniques for error-tolerant applications [25, 31]. Chippa et al. propose an automatic method to characterize the resilience of application to approximation [10]. Approximate computing application can also take advantage of specialized hardware. Esmaeilzadeh et al. used a neural network based optimizer that runs on a neural accelerator [15]. Specialized approximate hardware instructions and memory location can complement software methods to achieve efficient approximate computing [11, 14, 16, 26, 27, 38].

Compared to existing techniques, our solution monitors the data at runtime and takes local choices. Hence, while previous methods used the same alternative on the whole application's data domain, ours makes sure the approximate versions are active on portions of the computation space where they may be beneficial.

Automatic-Tuning Automatic tuning enables the compiler at automatically searching for the best parameters that allow for approximate computing to be efficient. PetaBricks is a programming language and compiler which uses empirical approach to select the best kernel among a pre-set of algorithms [1]. PetaBricks's compiler has been augmented to automatically generate approximate versions using a genetic algorithm which targets a minimum deviation of the output [2, 13]. Sculptor has extended loop perforation to target specific instruction and apply perforation at a finer grain with feedback [21]. Paraprox uses pattern recognition to generate approximate versions of kernels [35]. ASAC constructs a model to minimize a n-dimensional problem of selecting the values for approximate variables while keeping an acceptable quality of the result [33]. ACCEPT proposes a programming model that includes the approximate annotation in the language's grammar [37]. Green provides a set of code annotations that allow developers to use a range of approximation strategies with bounded error settings [3].

Our technique relies on the same information extraction methods as some of the previously mentioned works. We improved the previous works by presenting a novel high-level approximation optimization applied to stencil-based computation which generates a set of approximated versions with various performance/precision tradeoffs. Our solution is also the first one, to the best of our knowledge, to automatically apply adaptive approximation techniques to a compute intensive kernel.

Adaptive Methods Adaptive mesh refinement is a mathematical method which can be applied to solve partial differential equations, mainly for physics applications [5, 7]. Historically, the adaptive mesh method was tightly coupled with the application that used the method, but more recently packages such as PARAMESH [22], HAMR [28], DAGH [30] and AMRClaw [6, 23] allows developers to build new application using efficient adaptive mesh refinement implementations. Porting existing applications to these packages is time consuming and error prone. ACR provides a set of easy to use, optional annotations that can be positioned in front of compute expensive kernel [39, 40]. The annotations instruct the compiler to generate the adaptive version of the code, however they still require a significant domain-specific expertise from the programmer to set them in a pertinent way.

In our work, we propose a set of techniques to automatically set ACR annotations with convenient parameters, which builds the first automatic compiler optimization technique based on adaptive approximation methods.

7 CONCLUSION

With one simple annotation, our method allows for the first time developers to delegate to the compiler the time consuming task of optimizing an application's kernel using adaptive approximation techniques. We presented three important features that the compiler has to extract from the source code to generate a pertinent adaptive version. Precision level discovery, allows the compiler to choose a precision level from the application's dynamic data. This information is critical to find where the application needs the most precision and where more or less aggressive approximations can be used. The alternative generation requires the compiler to determine what kind of computation is performed by the kernel to apply appropriate code transformations. We used pattern matching to target stencil computation and proposed to merge multiple stencil steps to unleash more optimization opportunities. The third information, the granularity, looks for the best performance/precision tradeoff. With a grid too thin, the adaptive overhead will be too high and with a grid too coarse the perturbation will not be captured by the adaptive grid.

We evaluated our method on a set of representative applications resilient to inexact computations. We showed that we can extract enough information from these kernels and their profiling to generate an adaptive optimized version automatically. We also showed that the compiler can be instructed to use a specific alternative when the user sees it fit. Finally, we provided experimental evidence that the generated adaptive versions perform better than the original version while maintaining a good quality of the result.

Ongoing work includes investigation of adaptive techniques on more computational patterns. The automatic alternative generation could, in the future, use several alternatives simultaneously and evaluate the effect that an alternative has on the application's quality of the output, to e.g., statistically provide an upper bound on the deviation of the result.

REFERENCES

[1] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. Dublin, Ireland, 38–49.

[2] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. 2011. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 85–96.

[3] Woongki Baek and Trishul M. Chilimbi. 2010. Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. Toronto, Ontario, Canada, 198–209.

[4] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. 2003. Putting Polyhedral Loop Transformations to Work. In *International Workshop on Languages and Compilers for Parallel Computing*. College Station, Texas, USA, 209–225.

[5] Marsha J Berger and Phillip Colella. 1989. Local adaptive mesh refinement for shock hydrodynamics. *Journal of computational Physics* 82, 1 (1989), 64–84.

[6] M. J. Berger and R. J. LeVeque. 1998. Adaptive Mesh Refinement using Wave-Propagation Algorithms for Hyperbolic Systems. *SIAM J. Numer. Anal.* 35 (1998), 2298–2316.

[7] Marsha J Berger and Joseph Oliger. 1984. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.* 53, 3 (1984), 484 – 512.

[8] Hans-Joachim Bungartz and Michael Griebel. 2004. Sparse grids. *Acta numerica* 13 (2004), 147–269.

[9] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David M. Brooks. 2015. HELIX-UP: Relaxing Program Semantics to Unleash Parallelization. In *IEEE/ACM CGO*. San Francisco, USA, 235–245.

[10] Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*. ACM, 113.

[11] Vinay K. Chippa, Debabrata Mohapatra, Anand Raghunathan, Kaushik Roy, and Srimat T. Chakradhar. 2010. Scalable Effort Hardware Design: Exploiting Algorithmic Resilience for Energy Efficiency. In *Proceedings of the 47th Design Automation Conference (DAC '10)*. ACM, Anaheim, California, 555–560.

[12] Richard Courant and Fritz John. 2012. *Introduction to calculus and analysis I*. Springer Science & Business Media.

[13] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. 2015. Autotuning algorithmic choice for input sensitivity. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 379–390.

[14] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Architecture Support for Disciplined Approximate Programming. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, London, England, UK, 301–312.

[15] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Vancouver, B.C., CANADA, 449–460.

[16] Y. Fang, H. Li, and X. Li. 2012. SoftPCM: Enhancing Energy Efficiency and Lifetime of Phase Change Memory in Video Applications via Approximate Write. In *IEEE 21st Asian Test Symposium*. 131–136.

[17] C.J. Geankoplis. 1978. *Transport processes and unit operations*. Allyn and Bacon.

[18] R Wm Gosper. 1984. Exploiting regularities in large cellular spaces. *Physica D: Nonlinear Phenomena* 10, 1-2 (1984), 75–80.

[19] John L. Gustafson. 1988. Reevaluating Amdahl's Law. *Commun. ACM* 31, 5 (May 1988), 532–533.

[20] Weizhang Huang and Robert D Russell. 2010. *Adaptive moving mesh methods*. Vol. 174. Springer Science & Business Media.

[21] Shikai Li, Sunghyun Park, and Scott Mahlke. 2018. Sculptor: Flexible Approximation with Selective Dynamic Loop Perforation. In *Proceedings of the 2018 International Conference on Supercomputing (ICS '18)*. Beijing, China, 341–351.

[22] Peter MacNeice, Kevin M. Olson, Clark Mobarry, Rosalinda de Fainchtein, and Charles Packer. 2000. PARAMESH: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications* 126, 3 (2000), 330 – 354.

[23] Kyle T Mandli, Aron J Ahmadia, Marsha Berger, Donna Calhoun, David L George, Yiannis Hadjimichael, David I Ketcheson, Grady I Lemoine, and Randall J LeVeque. 2016. Clawpack: building an open source ecosystem for solving hyperbolic PDEs. *PeerJ Computer Science* 2 (2016), e68.

[24] Jiayuan Meng, Srimat Chakradhar, and Anand Raghunathan. 2009. Best-effort parallel execution framework for recognition and mining applications. In *IPDPS'09*. 1–12.

[25] Donald Michie. 1968. "Memo" functions and machine learning. *Nature* 218, 5136 (1968), 19.

[26] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load Value Approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. Cambridge, United Kingdom, 127–139.

[27] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, Portland, Oregon, USA, 309–328.

[28] Henry Neeman. 2000. HAMR: The Hierarchical Adaptive Mesh Refinement System. In *Structured Adaptive Mesh Refinement (SAMR) Grid Methods*, Scott B. Baden, Nikos P. Chrisochoides, Dennis B. Gannon, and Michael L. Norman (Eds.). Springer New York, New York, NY, 19–51.

[29] Ardavan F Oskooi, David Roundy, Mihai Ibanescu, Peter Bermel, John D Joannopoulos, and Steven G Johnson. 2010. MEEP: A flexible free-software package for electromagnetic simulations by the FDTD method. *Computer Physics Communications* 181, 3 (2010), 687–702.

[30] Manish Parashar and James C. Browne. 2000. Systems Engineering for High Performance Computing Software: The HDDA/DAGH Infrastructure for Implementation of Parallel Structured Adaptive Mesh. In *Structured Adaptive Mesh Refinement (SAMR) Grid Methods*, Scott B. Baden, Nikos P. Chrisochoides, Dennis B. Gannon, and Michael L. Norman (Eds.). Springer New York, New York, NY, 1–18.

[31] Abbas Rahimi, Luca Benini, and Rajesh K Gupta. 2013. Spatial memoization: Concurrent instruction reuse to correct timing errors in simd architectures. *IEEE Transactions on Circuits and Systems II: Express Briefs* 60, 12 (2013), 847–851.

[32] Martin Rinard. 2006. Probabilistic Accuracy Bounds for Fault-tolerant Computations That Discard Tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS '06)*. ACM, Cairns, Queensland, Australia, 324–334.

[33] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. 2014. ASAC: Automatic Sensitivity Analysis for Approximate Computing. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '14)*. ACM, Edinburgh, United Kingdom, 95–104.

[34] C. Rubio-González, Cuong Nguyen, Hong Diep Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12.

[35] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-based Approximation for Data Parallel Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, Salt Lake City, Utah, USA, 35–50.

[36] Mehrzad Samadi, Janghaeng Lee, Amir Jamshidi, D. Anoushe Hormati, and Scott Mahlke. 2013. SAGE: Self-tuning Approximation for Graphics Engines. In *MICRO'13 IEEE/ACM Intl. Symp. on Microarchitecture*. Davis, California, 13–24.

[37] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. 2015. Accept: A programmer-guided compiler framework for practical approximate computing. *University of Washington Technical Report UW-CSE-15-01* 1 (2015).

[38] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. 2014. Approximate Storage in Solid-State Memories. *ACM Trans. Comput. Syst.* 32, 3, Article 9 (Sept. 2014), 23 pages.

[39] M. Schmitt, P. Helluy, and C. Bastoul. 2017. Adaptive Code Refinement: A Compiler Technique and Extensions to Generate Self-Tuning Applications. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. Jaipur, India, 172–181.

[40] Maxime Schmitt, César Sabater, and Cédric Bastoul. 2017. Semi-Automatic Generation of Adaptive Codes. In *IMPACT 2017 - 7th International Workshop on Polyhedral Compilation Techniques*. Stockholm, Sweden, 1–7.

[41] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 124–134.

[42] Jos Stam. 2003. Real-Time Fluid Dynamics for Games. In *Proceedings of the Game Developer Conference*. 25.

[43] Leo Törnvist, Pentti Vartia, and Yrjö Vartia. 1985. How Should Relative Changes Be Measured? *The American Statistician* 39 (02 1985), 43–46.