Received: 19 September 2017 | Accepted: 15 December 2017
DOI: 10.1002/cpe.4423

SPECIAL ISSUE PAPER

WILEY

ATF: A generic directive-based auto-tuning framework

Ari Rasch | **Sergei Gorlatch**

University of Münster, Einsteinstraße 62, Münster, Germany

Correspondence
Ari Rasch, University of Münster, Einsteinstraße 62, Münster, Germany.
Email: a.rasch@uni-muenster.de

Funding information
BMBF project HPC2SE; DFG Cluster CiM

Summary

We describe the *Auto-Tuning Framework (ATF)* — a simple-to-use, generic approach and its implementation, as a framework for automatic program optimization by choosing the most suitable values of program parameters such as the number of parallel threads, tile sizes, etc. ATF combines four major advantages over the state-of-the-art auto-tuning: i) it is generic regarding the programming language, application domain, tuning objective (eg, high performance and/or low energy consumption), and search technique; ii) it can auto-tune a broader class of applications by allowing tuning parameters to be interdependent, eg, when one parameter is divisible by another parameter; iii) it allows tuning parameters to have substantially larger ranges by implementing an optimized search space generation process; and iv) it is arguably simpler to use, eg, the ATF user prepares an application for auto-tuning by annotating its source code with simple tuning directives. We demonstrate ATF's efficacy by comparing it to the state-of-the-art auto-tuning approaches, OpenTuner and CLTune; ATF shows better tuning results with less programmer's effort.

KEYWORDS
auto-tuning, CLBlast, CLTune, CUDA, dependent tuning parameters, GEMM, many-core, multi-core, multi-objective auto-tuning, OpenCL, OpenTuner, tuning parameter constraints

1 | MOTIVATION AND RELATED WORK

In order to achieve high performance and/or low energy consumption, programs have to be optimized for different hardware architectures. For programs on multi-core CPUs and many-core GPUs, the programmer has to choose values for performance-critical parameters (a.k.a. *tuning parameters*), eg, the number of parallel threads, how they are combined in groups, etc.

Auto-tuning is an approach to automatically find the optimal values of tuning parameters. To auto-tune a program, the programmer has to identify the program's tuning parameters and perform the following three steps: 1) generate the application-specific *search space*, which comprises the parameter configurations, ie, sets of tuning parameter values; 2) implement a *cost function* for estimating the program's cost for a particular configuration in terms of the target objective(s), eg, runtime performance and/or low energy consumption; and 3) explore the search space to find a configuration providing minimal cost using an automatized *search technique*. Auto-tuning systems have been successfully applied in different application areas, including ATLAS[1] for linear algebra routines, PATUS[2] for stencil computations, MILEPOST[3] for compiler optimizations, CHiLL[4] and Orio[5] for loop operations, Active Harmony[6] for runtime systems, and Apollo[7] for execution policies. The common feature of these approaches is that they are not generic, ie, they implement auto-tuning specifically for their target domains and cannot be used for applications from other domains.

OpenTuner[8] is a recent approach that is generic regarding the application domain, eg, given the specification of the tuning parameters (ie, their names and the ranges of possible values) and an arbitrary user-provided cost function, OpenTuner automatically generates a search space and explores it by using pre-implemented search techniques. However, OpenTuner does not provide mechanisms for expressing possible dependencies between tuning parameters. For example, the important linear algebra routine *GEMM (General Matrix Multiplication)*[9], when implemented in OpenCL, has various tuning parameters (tile size, work-group size, etc.), which have dependencies between them, eg, some parameter must be divisible by another parameter.[10] Due to these dependencies, OpenTuner is not capable of auto-tuning GEMM. For this restriction, the OpenTuner community has offered workarounds, eg, re-designing the user program so that its tuning parameters become independent[11] or setting a penalty value for

TABLE 1 Feature Comparison of OpenTuner, CLTune, and ATF

	OpenTuner	CLTune	ATF
Arbitrary programming language	✓		✓
Arbitrary application domain	✓	✓	✓
Arbitrary tuning objective	✓		✓
Arbitrary search technique		✓	✓
Dependent tuning parameters		✓	✓
Large parameter ranges	✓		✓
Directive-based auto-tuning			✓
Automatic cost function generation		✓	✓

configurations where the constraints are not met.[12] However, the first workaround usually requires a significant effort from the user, while the second may cause a poor tuning result as we demonstrate for GEMM in Section 7. Moreover, OpenTuner is optimized for programs whose tuning parameters have large ranges, and thus, it does not provide search techniques for small ranges, eg, exhaustive search that finds the probably best result.

CLTune[13] is a generic auto-tuning framework for OpenCL that allows tuning parameters to be interdependent — the search space can be filtered by user-defined boolean functions. However, CLTune is not generic regarding the programming language and tuning objective, ie, it is restricted to auto-tuning only OpenCL programs and only in terms of the runtime performance. Moreover, CLTune is applicable only for parameters with small ranges due to the time-intensive process of search space generation when the tuning parameters are interdependent. Thus, parameter ranges have to be artificially limited by the user of CLTune, requiring from him expert knowledge and often leading to non-optimal solutions.[8] We will demonstrate in this paper that even when such limitations are implemented by an expert, CLTune is still not suitable for auto-tuning the GEMM routine for practice-relevant input sizes that are used, eg, in the context of deep learning. [14]

Moreover, both approaches — OpenTuner and CLTune — require from the user specific programming skills for auto-tuning, eg, in Python (OpenTuner) or in C++ (CLTune), making auto-tuning hard for common application developers.

In this paper, we propose the *Auto-Tuning Framework (ATF)*, which combines the following advantages over the state-of-the-art auto-tuning approaches as summarized in Table 1.

i) ATF allows auto-tuning programs written in arbitrary programming languages and of arbitrary application domains, using a user-defined tuning objective and search technique; its pre-implemented search techniques suite programs with both small and large ranges of tuning parameters.
ii) ATF allows dependencies between tuning parameters thus enabling to auto-tune a broader class of applications.
iii) ATF allows substantially larger parameter ranges by optimizing the process of search space generation.
iv) ATF is arguably simpler to use; the user annotates the program's source code with simple tuning directives rather than having to implement an auto-tuning program, eg, in Python or C++. Moreover, for programs with a large number of tuning parameters, directives can be passed to ATF as a JSON file that is well suited to be generated automatically. Furthermore, ATF can automatically generate the cost function for the user.

2 | ILLUSTRATION OF ATF

We illustrate the usage of ATF by a simple example, ie, auto-tuning the `saxpy` kernel of the popular *CLBlast* library.[10] The kernel is written in OpenCL; it takes as its input the input vector size *N*, a floating point value *a*, and two *N*-sized vectors *x* and *y* of floating point values, and it computes

$$y[i] = a * x[i] + y[i],$$

for all $i \in [1, N]$.

Listing 1 shows a slightly simplified code of the `saxpy` kernel; for simplicity, we removed the switching between single precision and double precision as well as the usage of OpenCL's vector data types. The kernel is executed on a device (eg, a GPU) in parallel by several *Work-Items (WIs)* — the OpenCL term for thread. Each WI computes a chunk of `WPT`-many elements of the result vector; `WPT` stands for *Work-Per-Thread* — the CLBlast terminology for the chunk size — and is a tuning parameter; the programmer has to replace it textually (eg, using the OpenCL preprocessor) by a concrete value that is optimized for the target hardware, or he uses an auto-tuning tool to automatically determine and replace `WPT` by an optimized value. The WIs iterate over their corresponding chunks of the input (line 7) and compute in each iteration the index of the input elements of *x* and *y* (line 8) to be used in the computation of *y* (line 10). OpenCL requires WIs to be grouped in the so-called *work-groups*. The number of WIs per work-group is called the *local size* (`LS`), which is a further tuning parameter, ie, it has also be chosen specifically for the target device. The local size is set in the host code — C++ code that is required in OpenCL for invoking a kernel on a device.

For the correctness of the `saxpy` kernel, `WPT` must divide the input size `N`, such that each WI processes an equal-sized chunk of the input. Moreover, the two tuning parameters `WPT` and `LS` are interdependent; the OpenCL specification[15] requires the local size `LS` to divide the *global size* — the total number of work-items — which in case of the `saxpy` kernel, is `N/WPT`. Analogous to the local size, the global size is passed from the host code when invoking the kernel.

Listing 2 demonstrates how ATF is used for auto-tuning the `saxpy` kernel from Listing 1. The user annotates kernel's source code (line 25) with *ATF tuning directives* (line 1-23); they specify the auto-tuning process, ie, the tuning parameters, the cost function, and the search technique, and are explained in the following. The annotated source code is passed to ATF which then automatically generates and executes a specific auto-tuning program for `saxpy` according to the directives; the program yields as its result the best found configuration for `saxpy`'s two tuning parameters in the form of a JSON file — a common file format for storing human-readable name-value pairs. Since the ranges of `WPT` and `LS` are dependent on the input size — both parameters are in the interval from `1` to `N` where `N` is the input vector size — we auto-tune the `saxpy` kernel specifically for a fixed user-defined input size. The input size is passed by the user as the first command line argument to the generated auto-tuning program, as denoted by `$1` in line 1.

To perform the three steps for auto-tuning, ATF uses the tuning directives of Listing 2 as discussed in the following.

```
__kernel void saxpy( const          int      N,
                     const          float    a,
                     const __global float*   x,
                           __global float*   y
                   )
{
  for( int w = 0; w < WPT; ++w ) {
    const int index = w * get_global_size(0)
                        + get_global_id(0);

    y[ index ] += a * x[ index ];
  }
}
```

Listing 1 Simplified `saxpy` kernel from CLBlast

```
1   #atf::var::N $1
2
3   #atf::tp name       "WPT"
4            range      interval<size_t>( 1,N )
5            constraint divides( N )
6
7   #atf::tp name       "LS"
8            range      interval<size_t>( 1,N )
9            constraint divides( N/WPT )
10
11  #atf::ocl::platform "NVIDIA"
12  #atf::ocl::device   "Tesla K20"
13
14  #atf::ocl::input scalar<int>( N )   // N
15  #atf::ocl::input scalar<float>()    // a
16  #atf::ocl::input buffer<float>( N ) // x
17  #atf::ocl::input buffer<float>( N ) // y
18
19  #atf::ocl::global_size N/WPT
20  #atf::ocl::local_size  LS
21
22  #atf::search_technique annealing
23  #atf::abort_conition   duration<minutes>( 10 )
24
25  // saxpy kernel's code of Listing 1
```

Listing 2 The `saxpy` kernel of Listing1 annotated with ATF tuning directives

Step 1: Generate the search space

ATF automatically generates the search space according to the tuning parameters specified by the user. In Listing 2, the tuning parameters are: 1) the Work-Per-Thread `WPT` (line 3-5), which is a `size_t` parameter whose range is in the interval $[1, N]$ (line 4) that has to divide the input size N (line 5), and 2) the local size `LS` (line 7-9) — a `size_t` parameter in $[1, N]$ (line 8) that divides the global size `N/WPT` (line 9), correspondingly. The generated search space comprises the set of all valid parameter configurations, ie, configurations for which the parameters' constraints (line 5, 9) are satisfied.

The general form of an ATF tuning parameter is as follows:

```
#atf::tp name       /* name       */
         range      /* range      */
         constraint /* constraint */
```

It has a *name*, a *range*, and a *constraint*, which we explain in the following.

A name is a tuning parameter's unique identifier. Each occurrence of a tuning parameter's name is eventually replaced by ATF in the user program's source code by a hardware-optimized value.

A tuning parameter's range specifies the valid values of the parameter. It can be defined as either an interval or a set. An interval `interval<T>( begin, end, step_size, generator )` represents the values of an arbitrary primitive type `T` (eg, `bool`, `integer`, or `float`) from `begin` to `end` using the optional `step_size` that has a default value of `1`. The `generator` is also optional; it allows defining special parameter ranges, eg, of powers of two, and can be any arbitrary lambda function (a.k.a. anonymous function) with the input type `T` and arbitrary primitive output type `T'`. We denote lambda functions using a simplified C++ syntax, we declare a function's input parameters in parentheses together with their corresponding type, and we state the function's body in curly braces. For example, `(int i){return pow(2,i);}` is the lambda function that takes an integer `i` and returns the `i`-th power of `2`. When a generator function is used, the range type automatically changes to `T'`, and the interval comprises the values captured by `generator(i)` for each i from `begin` to `end`. For example, `interval<size_t>( 1, 10, atf::pow(2) )` represents the first ten powers of 2 — a step size is not passed and thus implicitly set by ATF to 1. Here, `atf::pow(2)` is a *generator function alias*; ATF automatically replaces the alias by the lambda function for generating powers of `2` as demonstrated above. Currently, ATF provides two generator function aliases: `atf::pow(N)` for generating the powers of an arbitrary integer `N` and `atf::is_multiple_of(N)` for generating a sequence of numbers that are multiples of `N`. Further generator function aliases can be easily added to ATF by the user; he adds the alias' name and the corresponding lambda function as a string to the corresponding ATF configuration file.

For small ranges, it is more convenient to state their elements explicitly. ATF provides for such ranges the range type set: the user states range's elements explicitly in curly braces, ie, {`val_1`, … , `val_n`}. Sets may also comprise user-defined values by denoting them as strings.

Constraints are a major feature of ATF; they enable filtering a tuning parameter's range. A constraint is a lambda function that takes a value of the parameter's range and returns a value of type `bool`, ie, values for which the constraint returns `false` are automatically filtered out of the range by ATF. Constraints can be used to conveniently define dependencies between tuning parameters, eg, in line 9 of Listing 2, we use the tuning parameter `WPT` in the constraint of the tuning parameter `LS` in order to ensure that `LS` divides `N/WPT`. Here, `divides( N/WPT )` is a *constraint alias*, which ATF automatically replaces by the lambda function `(size_t LS){ return (N/WPT) % LS == 0;}` in its initialization phase. Currently, ATF provides six constraint aliases: `divides`, `is_multiple_of`, `less_than`, `greater_than`, `equal`, and `unequal`. Constraints can be combined using the logical operators `&&` and `||`. The user can extend ATF easily by further constraint aliases; for this, he adds alias' name and the corresponding lambda function as a string to the corresponding configuration file.

Step 2: implement a cost function

ATF automatically generates the cost function according to the user-defined tuning directives. OpenCL requires OpenCL host code for its execution. ATF provides special directives for OpenCL; these allow ATF to automatically generate the host code according to the target OpenCL platform, device, kernel's input arguments, and global and local sizes. In our example in Listing 2, we use the Tesla K20 device of the system's OpenCL NVIDIA platform (line 11-12) — this could be, of course, any other OpenCL-compatible device. As the kernel's input, we use the input size N (line 14), a random floating point number for a (line 15) — a random number is automatically generated by ATF since no value is explicitly stated for a — and two N-sized buffers for x and y that are also filled with random floating point numbers (line 16-17) — random data is the default input when auto-tuning OpenCL kernels. The kernel's global and local sizes are set in line 19-20. The automatically generated cost function encapsulates kernel's source code, and it takes as input a configuration comprising concrete values for `WPT` and `LS`; it returns the `saxpy` kernel's runtime for concrete `WPT` and `LS`. For this, the cost function replaces in kernel's source code the tuning parameters' names by their corresponding values in the input configuration using the standard OpenCL preprocessor. Moreover, it uses a pre-implemented OpenCL host code for invoking the kernel with the passed global/local size and for measuring and returning the kernel's runtime using the standard OpenCL profiling API. To avoid the usually time-intensive host-to-device/device-to-host data transfers during auto-tuning, the cost function uploads data only once in its initialization phase and refrains from downloading the data — auto-tuning is performed on random data, and the computed result is not needed. Optionally, ATF provides directives for error checking the computed results, which we do not discuss here for brevity.

ATF also provides special directives for auto-tuning CUDA kernels. These are used by ATF to generate NVIDIA NVRTC host code,[16] which executes CUDA kernels. The ATF's CUDA directives are analogous to those for OpenCL with the only difference that platform's name (line 11) is omitted because CUDA targets NVIDIA devices only.

For programs that are not written in OpenCL or CUDA, ATF provides *language-independent directives* allowing to auto-tune programs in arbitrary programming languages: 1) `#atf::compile_script` followed by the path to a user-provided script that compiles the user's program, eg, a bash script and 2) `#atf::run_script` followed by the path to a user-provided script for running the compiled program. Optionally, the user can state the path to a *cost file* — a text file to which user's program will write its cost that ATF should minimize, eg, its energy consumption — by using the directive `#atf::cost_file`. If no cost file is specified, ATF automatically measures and uses program's runtime as cost. For multi-objective tuning, eg, auto-tuning for runtime together with energy consumption, the auto-tuned program writes pairs comprising runtime (in ms) and energy consumption (in microjoules) to the cost file; ATF minimizes these costs using the lexicographical order, ie, configuration `c` has a lower cost than configuration `c'` if either `c` has a lower runtime than `c'` or when `c` and `c'` have the same runtime and `c` causes a lower energy consumption than `c'`. The user can easily use a self-defined order by defining it as a string in the corresponding configuration file.

Step 3: Explore the search space

ATF automatically explores the search space that was generated in Step 1. For this, the user chooses one of ATF's pre-implemented *search techniques* and an *abort condition*. In this example, we use the simulated annealing search[17] (line 22) and the abort condition `atf::duration` (line 23), which causes to stop the tuning after 10 minutes.

Currently, ATF allows the user to choose among three search techniques: 1) exhaustive search, which finds the provably best configuration, but probably at the cost of a long search time; 2) simulated annealing, which has proven to be efficient for auto-tuning OpenCL and CUDA applications if search spaces are too large to be explored exhaustively[13]; and 3) OpenTuner's search engine which automatically combines various well-proven search techniques, eg, many variants of Nelder-Mead search and Torczon hillclimbers, to yield a good tuning result on average for arbitrary applications with large search spaces.[8] For domain-specific user requirements, the user can extend ATF by new search techniques as we demonstrate in Section 5.

To stop the exploration process, ATF offers the following six abort conditions:

1) `duration<D>(t)`: stops the tuning after a user-defined time interval `t`; here, `D` is the time unit: `seconds`, `minutes`, etc;
2) `evaluations(n)`: stops after `n` tested configurations;
3) `fraction(f)`: stops after `f*S` tested configurations, where `f` is a floating point value in [0, 1] and `S` the search space size;
4) `cost(c)`: stops when a configuration with a cost ≤`c` has been found;
5) `speedup<D>(s,t)`: stops when within the last time interval `t` the cost could not be lowered by a factor ≥`s`;
6) `speedup(s,n)`: stops when within the last `n` tested configurations the cost could not be lowered by a factor ≥`s`.

If no abort condition is specified, then ATF uses `evaluations(S)`, where `S` is the search space size. Abort conditions can be combined by using the logical operators `&&` and `||`. New abort conditions can be added easily to ATF by implementing a straightforward C++ interface.

3 | COMPARISON: ATF VERSUS CLTune

We now demonstrate that even though ATF is a generic approach, it is more expressive and easier to use for auto-tuning OpenCL than CLTune[13] that is specifically designed for OpenCL. In particular, we show that ATF's constraints conveniently express parameters' dependencies and that ATF allows tuning parameters of arbitrary types, making ATF more flexible and less memory-intensive than CLTune. Moreover, we demonstrate that the CLTune user has to use special functions for making the global and local size dependent on tuning parameters. In contrast, the ATF user expresses the global and local sizes as common arithmetic expressions built of tuning parameters and constants, thus further contributing to ATF's usability and expressiveness.

For comparison, we use the example of `saxpy`: we compare the ATF annotations for auto-tuning `saxpy` in Listing 2 with the CLTune program for `saxpy` shown in Listing 3 taken from.[18] In Listing 3, the user of CLTune chooses a target device, the OpenCL kernel, and prepares the kernel's input data (line 6-24). Then, he defines the `saxpy`-specific tuning parameters (line 26-39) and starts the tuning process (line 41-43). We compare ATF and CLTune for each of the three steps of auto-tuning.

Step 1: Generate the search space

CLTune allows the user to express parameter interdependencies by using C++ lambda functions — in this example: `DividesN` and `DividesNDivWPT` — which are used to filter the search space (Listing 3, line 32-36). ATF follows a different approach, ie,the user constrains the parameters' ranges (Listing 2, line 5, 9) rather than the search space, which may become very large. This design decision enables ATF to enormously accelerate the process of search space generation and, consequently, to tune parameters with significantly larger ranges as we show in Section 7. Moreover, CLTune requires using vectors for specifying constraints that are dependent on tuning parameters (Listing 3, line 32-33), making its use

```
1   int main()
2   {
3     const std::string saxpy = /* path to kernel of Listing 1   */;
4     const size_t      N     = /* fixed user-defined input size */;
5
6     cltune::Tuner tuner(1,0);
7     auto id = tuner.AddKernel(saxpy, "saxpy", {N}, {1} );
8
9     float a;
10    auto  x = std::vector<float>(N);
11    auto  y = std::vector<float>(N);
12
13    const auto random_seed =
        std::chrono::system_clock::now().time_since_epoch().count();
14    std::default_random_engine
        generator( static_cast<unsigned int>(random_seed) );
15    std::uniform_real_distribution<float> distribution(-2.0f,2.0f);
16
17    a = distribution(generator);
18    for (auto &item: x) { item = distribution(generator); }
19    for (auto &item: y) { item = distribution(generator); }
20
21    tuner.AddArgumentScalar( N );
22    tuner.AddArgumentScalar( a );
23    tuner.AddArgumentInput( x );
24    tuner.AddArgumentOutput( y );
25
26    auto range = std::vector<size_t>( N );
27    for( size_t i = 0; i < N ; ++i )
        range[ i ] = i;
29    tuner.AddParameter( id, "LS" , range );
30    tuner.AddParameter( id, "WPT", range );
31
32    auto DividesN = []( std::vector<size_t> v )
                      {
                        return  N % v[0] == 0;
                      };
33    auto DividesNDivWPT = []( std::vector<size_t> v )
                            {
                              return ( N / v[0] ) % v[1] == 0;
                            };
34
35    tuner.AddConstraint( id, DividesN      , {"WPT"}       );
36    tuner.AddConstraint( id, DividesNDivWPT, {"WPT", "LS"} );
37
38    tuner.DivGlobalSize(id, {"WPT" } );
39    tuner.MulLocalSize(id, {"LS"} );
40
41    tuner.UseAnnealing( 1.0f/2048.0f , 4.0 );
42    tuner.Tune();
43    const auto parameters = tuner.GetBestResult();
44  }
```

Listing 3 CLTune program for tuning `saxpy`

inconvenient. In contrast, ATF refrains from using vectors by passing a constraint to a particular tuning parameter (Listing 2, line 5, 9) and by allowing tuning parameters in constraints' definitions (line 9).

In CLTune, tuning parameters have to be of type `size_t`. In contrast, ATF allows tuning parameters of an arbitrary fundamental type (eg, also `bool` and `float`), making ATF suitable for a broad range of applications with differently-typed parameters. Furthermore, ATF reduces memory utilization since `size_t` has a higher memory consumption than, eg, `integer` or `bool`. For example, the GEMM routine of CLBlast introduced in Section 7 has ten tuning parameters of type `integer` and `bool`; using `size_t` for these parameters increases memory utilization by a factor > 2.

Step 2: Implement a cost function

ATF enables auto-tuning programs in arbitrary programming languages using an arbitrary tuning objective by providing language-independent tuning directives for automatically generating the cost function (see Section 2). In contrast, CLTune is only suitable for auto-tuning programs written in OpenCL and only in terms of runtime performance. ATF provides OpenCL-specific tuning directives for auto-tuning OpenCL programs (Listing 2, line 11-20). We argue that the usage of ATF for OpenCL is better than CLTune due to the following reasons.

In CLTune, the values of the global and local sizes have to be passed to the function `AddKernel` (Listing 3, line 7). Since these two values are usually dependent on tuning parameters, CLTune provides special functions `DivGlobalSize` (line 38) and `MulLocalSize` (line 39) to override the initial value by dividing/multiplying the global/local size with a tuning parameter's value. In contrast, ATF handles this more generally and allows defining the global/local size as common arithmetic expressions that may contain tuning parameters (Listing 2, line 19-20). The ATF user need not set the global and local sizes to initial values and modify them later using special functions to make them dependent on tuning parameters. The generality of ATF makes it especially more expressive; for example, in the CLBlast library, GEMM kernel's global size is computed as an arithmetic expression comprising tuning parameters and constants — this cannot be expressed in CLTune, ie, CLBlast has to use a simplified global size for auto-tuning its GEMM kernel, which causes non-optimal tuning results as we demonstrate in Section 7.

ATF's automatically generated OpenCL cost function can generate random input data of arbitrary fundamental data types (eg, `float`) as usually required for auto-tuning OpenCL kernels. For example, `scalar<T>()` (Listing 2, line 15) represents a random value of type `T` that is passed by the cost function to the kernel, and `atf::buffer<T>(N)` (line 16-17) represents a buffer containing `N` random elements of `T`. In comparison, the CLTune user is responsible for generating input data (Listing 3, line 13-19). If the ATF user aims at auto-tuning for concrete input data, then he uses `scalar(a)` for the scalar `a` of an arbitrary fundamental type (Listing 2, line 14), and `atf::buffer<T>( file_name(“./path/to/file”) )` where `./path/to/file` is the path to a text file that contains the buffer's values line by line.

Step 3: Explore the search space

In contrast to CLTune, ATF provides as an optional search technique the OpenTuner's search engine with its broad choice of search methods, enabling effective exploration especially for large search spaces.[8] Moreover, ATF allows an arbitrary objective and multi-objective tuning (see Section 2), while CLTune is restricted to exploring its search space only in terms of runtime performance as objective.

CLTune uses as abort condition testing a user-defined number of configurations while ATF offers further abort conditions, eg, to stop the tuning depending on the tuning result (`cost` and `speedup`), and allows to combine abort conditions by logical operators to meet complex user requirements.

4 | COMPARISON: ATF VERSUS OpenTuner

OpenTuner is optimized for auto-tuning programs whose tuning parameters have no interdependencies. In the following, we compare OpenTuner and ATF, and we show that ATF efficiently auto-tunes programs of OpenTuner's target application class, ie, without interdependent tuning parameters. For comparison, we use the OpenTuner program that searches over the space of GNU Compiler Collection's (GCC) optimization options[8] in order to minimize the compiled program's runtime. In this example, our application program is `raytracer`, which is provided by OpenTuner. [8]

Listing 4 shows the OpenTuner program for auto-tuning GCC's optimization options for raytracer. The user defines one tuning parameter per option (line 8-19, 23-38), — 322 in total — by overriding the `manipulator` function (line 23) of OpenTuner's class `MeasurementInterface` (line 21) and by using a so-called OpenTuner *configuration manipulator* (line 28); The OpenTuner provides a straightforward Python script for extracting the GCC's options used in line 8-19. To define the cost function, the user overrides OpenTuner's `run` function (line 40-61); in this function, he has to explicitly construct a GCC command with the optimization options according to the input configuration (line 45-55). Moreover, this function compiles `raytracer` using the constructed GCC command and runs, measures, and returns the compiled `raytracer`'s runtime (line 57-61).

Listing 5 shows the ATF directives for performing the same auto-tuning task of auto-tuning GCC's optimization flags. For programs with a large number of tuning parameters — in this example: 322 parameters — ATF takes directives also as a JSON file instead of source code annotations since the JSON file format is well suited to be generated automatically and widely supported in all common programming languages. For generating the JSON file of Listing 5, we use a straightforward Python script; it is a slightly modified version of the Python script that OpenTuner provides and uses

```
 1  import opentuner
 2  from opentuner import ConfigurationManipulator
 3  from opentuner import EnumParameter
 4  from opentuner import IntegerParameter
 5  from opentuner import MeasurementInterface
 6  from opentuner import Result
 7
 8  GCC_FLAGS = [
 9    'align-functions', 'align-jumps', 'align-labels',
10    'branch-count-reg', 'branch-probabilities',
11    # ... (176 total)
12  ]
13
14  # (name, min, max)
15  GCC_PARAMS = [
16    ('early-inlining-insns', 0, 1000),
17    ('gcse-cost-distance-ratio', 0, 100),
18    # ... (145 total)
19  ]
20
21  class GccFlagsTuner(MeasurementInterface):
22
23    def manipulator(self):
24      """
25      Define the search space by creating a
26      ConfigurationManipulator
27      """
28      manipulator = ConfigurationManipulator()
29      manipulator.add_parameter(
30        IntegerParameter('opt_level', 0, 3))
31      for flag in GCC_FLAGS:
32        manipulator.add_parameter(
33          EnumParameter(flag,
34                        ['on', 'off', 'default']))
35      for param, min, max in GCC_PARAMS:
36        manipulator.add_parameter(
37          IntegerParameter(param, min, max))
38      return manipulator
39
40    def run(self, desired_result, input, limit):
41      """
42      Compile and run a given configuration then
43      return performance
44      """
45      cfg = desired_result.configuration.data
46      gcc_cmd = 'g++ apps/raytracer.cpp -o ./tmp.bin'
47      gcc_cmd += ' -O{0}'.format(cfg['opt_level'])
48      for flag in GCC_FLAGS:
49        if cfg[flag] == 'on':
50          gcc_cmd += ' -f{0}'.format(flag)
51        elif cfg[flag] == 'off':
52          gcc_cmd += ' -fno-{0}'.format(flag)
53      for param, min, max in GCC_PARAMS:
54        gcc_cmd += ' --param {0}={1}'.format(
55          param, cfg[param])
56
57      compile_result = self.call_program(gcc_cmd)
58      assert compile_result['returncode'] == 0
59      run_result = self.call_program('./tmp.bin')
60      assert run_result['returncode'] == 0
61      return Result(time=run_result['time'])
62
63  if __name__ == '__main__':
64    argparser = opentuner.default_argparser()
65    GccFlagsTuner.main(argparser.parse_args())
```

Listing 4 OpenTuner program for auto-tuning GCC's flags for raytracer

for extracting the GCC's options in Listing 4, line 8-19. Our only modification is that the script outputs its result in the ATF syntax rather than in the OpenTuner syntax. The ATF directives in Listing 5 comprise the tuning parameters (line 2-23) without the boilerplate code for defining a configuration manipulator, inheriting from a class and overriding a function, as required in OpenTuner's approach (Listing 4, line 23-38). Moreover, ATF frees the user from the burden of implementing a cost function in Python in which user has to explicitly construct the GCC command with the options according to the input configuration: ATF automatically generates the cost function that replaces tuning parameters' names in user program's source code — in this example: a bash script comprising the GCC command shown in Listing 6 — by the corresponding values in the input configuration, and it automatically compiles, runs, and measures the runtime of the compiled `raytracer`. For generating the bash script in Listing 6, we use again a modified version of OpenTuner's Python script for extracting GCC's options, ie, it straightforwardly generates the GCC command with options that have parametrized values, eg, we use as value for option `early-inlining-insns` the parameter name `early-inlining-insns_val` (Listing 6, line 4) instead of a concrete integer value. For generating the cost function, ATF only requires from the user the path to the bash script in Listing 6 (Listing 5, line 25), the command for compiling `raytracer`, which is again the bash script (line 26), and the command for running the compiled `raytracer` program (line 27).

Note that ATF automatically generates a cost function that substitutes tuning parameters' names in the source code written in an arbitrary programming language, and it also automatically runs and measures the modified program. In contrast, OpenTuner requires from the user to explicitly implement the cost function in which user has to construct the target program's source code and to explicitly compile and run the constructed

```
{
  "tuning_parameters" :
  {
    { "name"  : "opt_level",
      "range" : {"-O0", "-O1", "-O2", "-O3" }
    },

    { "name"  : "align_functions",
      "range" : { "-falign_functions", "-fno-align_functions"}
    },
    { "name"  : "align-jumps",
      "range" : { "-falign-jumps", "-fno-align-jumps" }
    },
    // ... (176 total)

    { "name"  : "early-inlining-insns_val",
      "range" : "interval<int>( 0, 1000 )"
    },
    { "name"  : "gcse-cost-distance-ratio_val",
      "range" : "interval<int>( 0, 100  )"
    },
    // ... (145 total)
  },

  "program_source" : "./bash_script.sh",
  "compile_script" : "./bash_script.sh",
  "run_script"     : "./tmp.bin"

  "search_technique" : "open_tuner",
  "abort_condition"  : "duration<minutes>(30)",
}
```

Listing 5 ATF directives for auto-tuning GCC's flags (JSON file).

```
#!/bin/bash
g++ apps/raytrayer.cpp -o ./tmp.bin opt_level
  align-functions align-jumps                      # ... 176 total
  --param early-inlining-insns=early-inlining-insns_val
  --param gcse-cost-distance-ratio
      =gcse-cost-distance-ratio_val                # ... 145 total
```

Listing 6 GCC's flags that are auto-tuned by ATF according to the directives in Listing 5 (bash script).

program as discussed above. This approach usually requires a great effort from the user if the target application is not a GCC command and programming skills in Python. We demonstrate in Section 7 that even though ATF provides an easier user interface, it produces the tuning results of the same quality as OpenTuner.

5 | SEARCH TECHNIQUES IN ATF

ATF provides three search techniques: 1) exhaustive search, 2) simulated annealing, and 3) OpenTuner search. All of them implement the same generic interface, ie,

```
class search_technique
{
  void          initialize( search_space sp );
  void          finalize();
  configuration get_next_config();
  void          report_cost( size_t cost );
}
```

Here, function `initialize` takes the search space as an argument; it is called by ATF before starting the exploration process to initialize the search technique for the passed search space. Function `finalize` is the counterpart of `initialize`; it is called after the exploration process, eg, to free allocated memory, etc. During exploration, ATF performs repeatedly the following two steps until the chosen abort condition (eg, a specific duration) is satisfied: 1) it takes a configuration using the function `get_next_config`, measures the configuration's cost by using the automatically generated cost function, and 2) it reports the measured cost using the `report_cost` function.

We now discuss the implementation of these four functions for each of ATF's three search techniques. Further search techniques can be added to ATF by implementing the `search_technique` interface.

5.1 | Exhaustive search

The exhaustive search iterates straightforwardly over the search space. The implementation of `finalize` and `report_cost` is void; function `initialize` stores a reference to the passed search space, and `get_next_config` returns straightforwardly for each call a new configuration within the search space.

5.2 | Simulated annealing

For simulated annealing, which has proven to be effective for exploring OpenCL and CUDA search spaces,[13] the implementation of `initialize` and `finalize` is straightforward, ie, the memory for intermediate results is allocated/deallocated, etc. Function `get_next_config` returns in each call a random neighbor c' of the current configuration c, and the corresponding runtime t' of c' is reported using function `report_cost`. The configuration c' becomes the new current configuration with probability

$$P(t, t', T) = e^{-(t'-t)*T^{-1}}$$

if $t' \geq t$ and 1 otherwise. Here, t and t' represent the runtimes of configurations c and c', and T is the so-called annealing temperature. The value $T = 4$ was reported as suitable for OpenCL and CUDA.[13]

5.3 | OpenTuner's search engine

The OpenTuner framework[8] implements various search techniques, eg, Nelder-Mead and Torczon hillclimbers, which are selected automatically for a concrete search space. As discussed above, OpenTuner is not suitable for auto-tuning parameters with dependencies between them. We employ the OpenTuner's search engine as one of possible ATF's search techniques because ATF's search space contains per construction only configurations where parameters' dependencies are already satisfied (see Section 2). For this, we define the OpenTuner's parameter `TP` with a range of integers from 1 to `S`, where `S` is the search space size, and we use `TP` as the index for configurations within the ATF search space.

The implementation of the `search_technique` class' functions for OpenTuner is as follows. We embed OpenTuner's Python interface in C++ by using the Python-provided C++ embedding API.[19] In the `initialize` function, we initialize the Python interpreter and embed straightforwardly the OpenTuner interface, which we then use to define the OpenTuner tuning parameter `TP`. Function `get_next_config` takes in each call from OpenTuner a new prediction for `TP` and returns the configuration with index `TP` within the ATF search space. The returned configuration is evaluated by ATF using the user-provided cost function, and the corresponding cost is passed to OpenTuner by calling the `report_cost` function, which then calls the corresponding function of the OpenTuner interface. The `finalize` function destructs the Python embedding API.

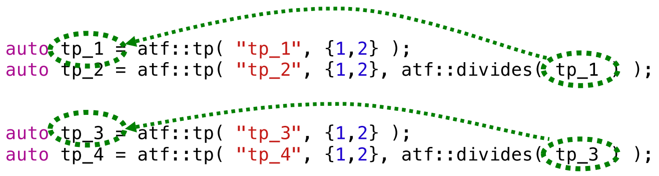
In case of auto-tuning independent tuning parameters, ie., when the tuning parameters have no constraints, the ATF search space coincides with that of OpenTuner. We then use for each ATF tuning parameter a corresponding OpenTuner tuning parameter with the same range of values since the OpenTuner promises good results when using multiple tuning parameters.

6 | PARALLEL SEARCH SPACE GENERATION

Applications with many tuning parameters (eg, the matrix multiplication GEMM has 10 tuning parameters) usually comprise groups of interdependent parameters. ATF uses this in order to generate search space faster, in parallel using C++ multithreading.

Figure 1 shows a simple (artificial) example of four tuning parameters `tp_1`, … , `tp_4`. For simplicity, each parameter has the same small range of values 1 and 2. The constraint of `tp_2`, ie, `atf::divides(tp_1)`, uses `tp_1`, and analogously, the constraint of `tp_4` uses `tp_3`; the parameters `tp_1` and `tp_3` have no constraints. Therefore, `tp_1` and `tp_2` build together a group of interdependent parameters, and `tp_3` and `tp_4` build another group. For each group, the corresponding part of the search space can be generated in parallel because parameters `tp_1` and `tp_2` do not influence possible values for `tp_3` and `tp_4` and vice versa.

ATF can easily determine the dependencies between tuning parameters by straightforwardly analyzing their constraints. Thus, ATF can generate the search space in parallel by using the Standard C++ Threading Library with one thread per group of interdependent parameters.



```
auto tp_1 = atf::tp( "tp_1", {1,2} );
auto tp_2 = atf::tp( "tp_2", {1,2}, atf::divides( tp_1 ) );

auto tp_3 = atf::tp( "tp_3", {1,2} );
auto tp_4 = atf::tp( "tp_4", {1,2}, atf::divides( tp_3 ) );
```

FIGURE 1 Example of tuning parameters that allow optimized search space generation

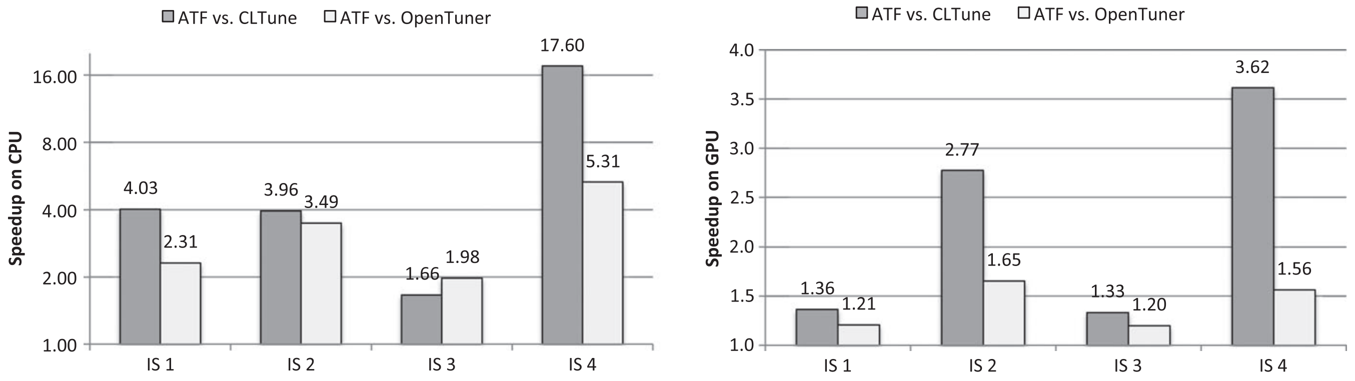


FIGURE 2 Speedup (higher is better) of the `XgemmDirect` kernel auto-tuned by ATF over auto-tuning by CLTune and OpenTuner on Intel CPU (left) and NVIDIA GPU (right), using four different input sizes IS1-IS4

7 | EXPERIMENTAL RESULTS

In this section, we demonstrate that ATF provides significantly better tuning results as compared to CLTune and OpenTuner for the important routine GEMM (General Matrix Multiplication),[9] which is a cornerstone of the current deep learning implementations. Moreover, we demonstrate that ATF provides the same good tuning results for the application classes that are specifically targeted by OpenTuner and CLTune.

For evaluation, we use a dual-socket system equipped with two Intel Xeon E5-2640 v2 8-core CPUs, tacted at 2GHz with 128 GB main memory and hyper-threading enabled, as well as a NVIDIA Tesla K20 m GPU. We perform experiments using both the CPU and GPU as OpenCL devices. The dual-socket CPU is represented in OpenCL as a single device with 32 compute units, corresponding to the overall 2×16 logical cores in the system. We compare CLTune version 2.6.0 with OpenTuner version 0.7.0. The `XgemmDirect` kernel is extracted from CLBlast version 0.11.0.

7.1 | Case study: General Matrix Multiplication

As a concrete GEMM implementation, we take the `XgemmDirect` kernel from the auto-tunable OpenCL BLAS library CLBlast,[10] which uses CLTune for auto-tuning. The kernel is optimized for small matrix sizes of up to $2^{10} \times 2^{10}$, and it is used to accelerate important applications, eg, the state-of-the-art deep learning framework Caffe.[14] It has 10 tuning parameters, with the following ranges for $N \times N$ input matrices[10]:

- 6 integer parameters `WGD`, `MDIMCD`, `NDIMCD`, `MDIMAD`, `NDIMBD`, and `KWID`, each with a range $\{1, \ldots, N\}$;
- 2 integer parameters `VWMD` and `VWND`, each with a range $\{1, 2, 4, 8\}$;
- 2 boolean parameters `PADA` and `PADB`, each with a range {`true`,`false`}.

Here, `WGD` represents the tile size, and `KWID` denotes the loop unrolling factor. The parameters have various interdependencies (17 in total), eg, `KWID` divides `WGD`, and `WGD` divides the result matrix's number of rows and number of columns.

Figure 2 shows the measured speedup of the `XgemmDirect` kernel auto-tuned by ATF as compared to the kernel auto-tuned by CLTune and OpenTuner, correspondingly. We study four pairs of matrix input sizes (IS) that are heavily used in the Caffe framework, eg, in Caffe's sample `siamese`, and thus are of great importance in the context of deep learning:

- IS 1: 20×1 and 1×576;
- IS 2: 20×25 and 25×576;
- IS 3: 50×1 and 1×64;
- IS 4: 10×64 and 64×500;

We employ the CLTune program that CLBlast uses for tuning `XgemmDirect`,[10] and we implement the OpenTuner program for this kernel according to the work of Bruel et al,[12] where we use the unconstrained search space; we report a penalty value in case of a configuration for which `XgemmDirect`'s constraints are not satisfied.

In Figure 2, we observe that in comparison to CLTune, ATF improves `XgemmDirect`'s runtime by factors from 1.66× to 17.60× on the CPU (left part of the figure, logarithmic scale) and from 1.33× to 3.62× on the GPU (right part of the figure). The reason is that CLBlast artificially limits CLTune's tuning parameter ranges apparently because of CLTune's time-intensive process of search space generation. For example, the tile size `WGD` is limited to $\{8, 16, 32\}$ and is constrained to divide result matrix's number of rows and number of columns.[10] Due to this constraint, the range limitation of `WGD` causes the search space to be empty for the matrix sizes used in deep learning, ie, either the result matrix's number of rows or number of columns is not a multiple of 8. Therefore, the `XgemmDirect` kernel relies on CLTune's device-optimized values for its tuning parameters (ie, optimized for the average matrix input size of 256×256), thus causing a poor performance. The higher speedup of ATF on the CPU as compared to GPU is because `XgemmDirect`'s limited tuning parameter ranges comprise values that are rather optimal for the GPUs' architecture than for CPUs.

We tried to improve the CLTune program by removing the artificial limitations on the parameters' ranges. However, even for the multiplication of small 32 × 32 matrices, the search space generation by CLTune takes too much time — we aborted after 3 hours — while ATF requires less than 1 second for generating its search space. The reason is that ATF filters out invalid configurations by iterating over the constrained ranges of tuning parameters as described in Section 2. In contrast, CLTune iterates over the unconstrained search space, which is usually very large, to filter out the invalid configurations. For the routine's maximal supported matrix size $2^{10} \times 2^{10}$, the unconstrained space of all possible configurations has a prohibitively huge size of more than 10^{19} configurations, while the constrained search space in ATF comprises only nearly 10^{7} configurations.

Moreover, ATF allows refraining from some of the constraints used in CLTune's program for `XgemmDirect` since ATF allows expressing the OpenCL global and local sizes more generally than CLTune (see Section 3). For example, in CLTune, the constraints on the tile size ensure that the local size divides evenly the global size. However, in CLBlast, the global size is automatically adapted to a multiple of the local size — this is done by performing arithmetic operations between tuning parameters and constants, which cannot be expressed in CLTune (see Section 3). In contrast, ATF allows to express the global and local size as common arithmetic expressions that may contain tuning parameters and, consequently, to use the global and local sizes that CLBlast uses for its `XgemmDirect` kernel. Thus, in our ATF program, we can refrain from CLTune's constraints for the global and local size, which enables ATF to generate and explore a larger search space of valid configurations. The larger search space leads to better tuning results for `XgemmDirect` since it comprises configurations that provide high performance and are not comprised by CLTune's search space. For example, in case of the input size `IS4`, the larger search space improves ATF's speedup from 12.85× to 17.60× on the CPU and from 2.89× to 3.62× on the GPU.

In Figure 2, comparing ATF and OpenTuner, ATF speedups the `XgemmDirect` kernel by factors from 1.98× to 5.31× on the CPU (left), and by factors from 1.20× to 1.65× on the GPU (right). This is because OpenTuner is optimized for unconstrained search spaces and, thus, is not able to find a valid configuration even after 10 000 evaluated configurations since valid configurations make only a tiny fraction of `XgemmDirect`'s search space. For example, for the input size IS 4, the unconstrained search space of OpenTuner has a size of 10^{13} while the number of valid configurations is 10^{6}— it corresponds exactly to the constrained search space of ATF which cannot be expressed in OpenTuner — ,ie, the probability of choosing a valid configuration is 10^{-7}. Consequently, OpenTuner is not suitable for auto-tuning the `XgemmDirect` kernel, and the kernel has to rely on its tuning parameters' default values, which are neither optimized for the target device nor for the input size; they are chosen to yield a good performance on average on various devices and for different input sizes.

Surprisingly, in most cases, `XgemmDirect`'s performance is better when using its default tuning parameter values as compared to using its device-optimized tuning parameter values that CLBlast hast determined with CLTune. This is because the default parameter values are small, eg, `WGD=8` and `KWID=1`, causing a high parallelization of `XgemmDirect` for the special input sizes as used in deep learning.

7.2 | Case study: 2D Convolution

In this section, we demonstrate that ATF provides tuning results of competitive quality as compared to CLTune for CLTune's target application class. For demonstration, we use the 2D Convolution sample that is provided by CLTune — it is written in OpenCL and has various tuning parameters, eg, the work per thread and the loop unrolling factor.[13] We compare the tuning results of the CLTune program for 2D Convolution[13] with the results of a corresponding ATF program.

Figure 3 shows how 2D Convolution's runtime evolves on the GPU during auto-tuning with CLTune and ATF. We show the runtime of the so far best found configuration after each tested configuration by CLTune and ATF, correspondingly. According to the abort condition of the CLTune program, we test a fraction 1.6% of search space, ie, 53 configurations, in each tuning run. For ATF, we demonstrate tuning results for both of its heuristic search techniques: simulated annealing, and OpenTuner. We do not use ATF's exhaustive search technique since we test only a small fraction of the search space in a tuning run.

We can observe that ATF reaches the same good performance for 2D Convolution as CLTune. This is because ATF implements and uses the simulated annealing search which is effective for exploring OpenCL search spaces and is also used by CLTune. ATF reaches also good tuning results for its OpenTuner search technique. This is because of the high efficacy of the OpenTuner search when used as an ATF search technique. The OpenTuner explores ATF's constrained search space which comprises only valid configurations. We also tried to compare against the OpenTuner framework as a whole, ie, when not used as an ATF search technique; however, OpenTuner was not able to find valid configurations and thus tuning fails. OpenTuner uses the unconstrained search space of which a vast fraction of 99.98% are invalid configurations.

We obtained analogous tuning results for 2D convolution on the CPU which we do not discuss for brevity.

7.3 | Case study: GCC's flags

We demonstrate that ATF provides the same good tuning results as compared to OpenTuner for OpenTuner's target application class, ie, applications without interdependent tuning parameters. As our case study, we use the example of auto-tuning GCC compiler's optimization options discussed in

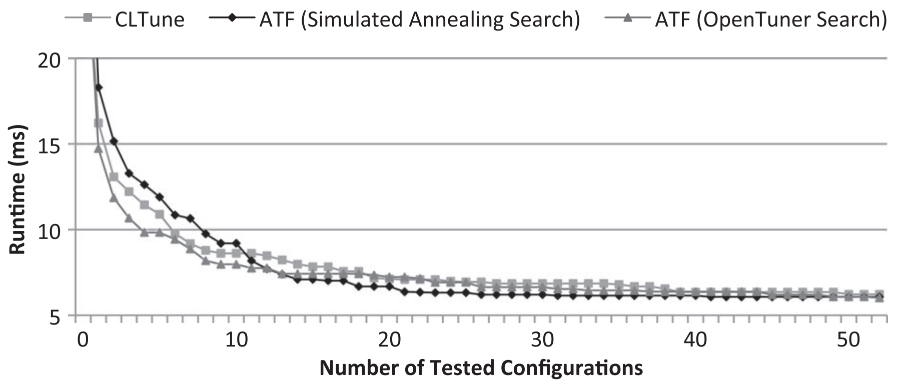


FIGURE 3 Evolution of 2D Convolution's runtime (lower is better) on the GPU with increasing tuning time (measured in number of tested configurations) when auto-tuned by 1) CLTune, 2) ATF with its simulated annealing search technique, and 3) ATF with its OpenTuner search technique. We show the median runtime of 30 tuning runs

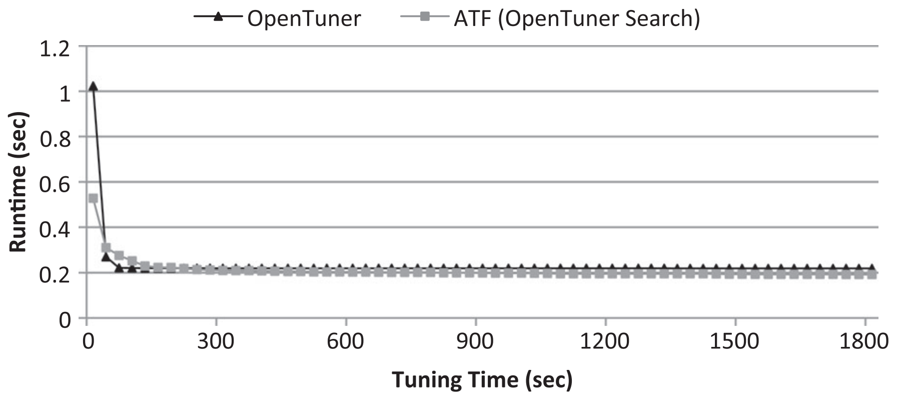


FIGURE 4 Evolution of raytracer's runtime (lower is better) with increasing tuning time (measured in seconds) when auto-tuned by 1) OpenTuner and 2) ATF with its OpenTuner search technique. We show the median runtime of 30 tuning runs

Section 4. For evaluation, we use for OpenTuner the program in Listing 4, and for ATF, we use the program in Listing 5. We stop each tuning run after 30 minutes of tuning time according to the OpenTuner program's abort condition. We are not able to compare against CLTune since it is designed for auto-tuning OpenCL programs only.

In Figure 4, we observe that ATF and OpenTuner reach the same performance for raytracer after 30 minutes of tuning. This is because the GCC's options have no interdependencies, ie, they can be auto-tuned independently of each other and thus, the search spaces of ATF and OpenTuner coincide. Since we integrated the OpenTuner's search engine into ATF, we reach the same tuning results for OpenTuner and ATF.

We do not show tuning results for ATF with its simulated annealing search since the results were poor. We reach a median runtime of > 1 second for raytracer. This is due to the GCC's options vast search space size of 10^{806} configurations requiring advanced search techniques for exploration such as provided by OpenTuner.

8 | CONCLUSION

In this paper, we present ATF — a highly generic framework for program auto-tuning that has several advantages as compared to the state-of-the-art approaches. ATF can auto-tune programs written in an arbitrary programming language and belonging to an arbitrary application domain; tuning parameters are allowed to be interdependent. The user can auto-tune for an arbitrary objective (eg, high runtime performance and/or low energy consumption) and choose among three pre-implemented search techniques, targeting both small and large search spaces; for domain-specific user requirements, ATF can be easily extended by further search techniques. We demonstrate that ATF is easy to use. The ATF user annotates the source code with simple tuning directives rather than implements a tuning program, eg, in Python or C++ — which makes auto-tuning appealing to common application developers. Our experimental results show that ATF provides better tuning results than the state-of-the-art approaches CLTune and OpenTuner for General Matrix Multiplication (GEMM) written in OpenCL on important input sizes as used in deep learning. Moreover, we demonstrate that ATF provides the tuning results of the same good quality for the application classes that are specifically targeted by CLTune and OpenTuner.

ACKNOWLEDGMENTS

This work has been supported by the BMBF project HPC2SE and the DFG Cluster CiM. We thank NVIDIA Corporation for donating the hardware used in our experiments.

ORCID

Ari Rasch http://orcid.org/0000-0002-0286-0755

REFERENCES

1. Clint WR, Dongarra JJ. Automatically tuned linear algebra software. Paper presented at: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing; 1998; Washington, DC.
2. Christen M, Schenk O, Burkhart H. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. Paper presented at: 2011 IEEE International Parallel Distributed Processing Symposium; 2011; Anchorage, AK.
3. Fursin G, Kashnikov Y, Memon AW, et al. Milepost GCC: machine learning enabled self-tuning compiler. *Int J Parallel Prog*. 2011;39(3):296-327.
4. Chen C, Chame J, Hall M. CHILL: A framework for composing high-level loop transformations. [Technical Report 08-897]. Los Angeles, CA; 2008:136-150.
5. Hartono A, Norris B, Sadayappan P. Annotation-based empirical performance tuning using Orio. Paper presented at: 2009 IEEE International Symposium on Parallel Distributed Processing; 2009; Rome, Italy.
6. Ţăpuş C, Chung I-H, Hollingsworth JK. Active harmony: towards automated performance tuning. Paper presented at: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, IEEE Computer Society Press; 2002; Baltimore, MD.
7. Beckingsale D, Pearce O, Laguna I, Gamblin T. Apollo: Reusable models for fast, dynamic tuning of input-dependent code. Paper presented at: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS); 2017; Orlando, FL.
8. Ansel J, Kamil S, Veeramachaneni K, et al. OpenTuner: An extensible framework for program autotuning. Paper presented at: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation; 2014; Edmonton, Canada.
9. Netlib BLAS; 2016.
10. Nugteren C. CLBLast: A tuned openCL BLAS library; 2017.
11. Jason J. OpenTuner issue 87; 2016.
12. Bruel P, Amaris M, Goldman A. Autotuning CUDA compiler parameters for heterogeneous applications using the opentuner framework. *Concurrency Computat Pract Exper*. 2017;29(22).
13. Nugteren C, Codreanu V. CLTUne: A generic auto-tuner for OpenCL kernels. Paper presented at: 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip; 2015; Turin, Italy.
14. Jia Y, Shelhamer E, Donahue J, et al. Caffe: Convolutional architecture for fast feature embedding. Paper presented at: Proceedings of the 22nd ACM International Conference on Multimedia; 2014; Orlando, FL.
15. Khronos openCL working group. The OpenCL Specification; 2017.
16. NVIDIA. NVRTC; 2017.
17. Kirkpatrick S, Gelatt CD, Vecchi MP, et al. Optimization by simulated annealing. *Science*. 1983;220(4598):671-680.
18. Nugteren C. CLTune issue 48; 2016.
19. Python Software Foundation. Python Embedding API; 2017.

How to cite this article: Rasch A, Gorlatch S. ATF: A generic directive-based auto-tuning framework. *Concurrency Computat Pract Exper*. 2018;e4423. https://doi.org/10.1002/cpe.4423