2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems

ATF: A Generic Auto-Tuning Framework

Ari Rasch
University of Muenster
Einsteinstr. 62
Muenster, Germany
Email: a.rasch@wwu.de

Michael Haidl
University of Muenster
Einsteinstr. 62
Muenster, Germany
Email: m.haidl@wwu.de

Sergei Gorlatch
University of Muenster
Einsteinstr. 62
Muenster, Germany
Email: gorlatch@wwu.de

***Abstract*—We describe the *Auto-Tuning Framework (ATF)* — a novel generic approach for automatic program optimization by choosing the most suitable values of program parameters, such as number of parallel threads, tile sizes, etc. Our framework combines four advantages over the state-of-the-art auto-tuning: i) it is generic regarding the programming language, application domain, tuning objective (e.g., 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, e.g., when one parameter is divisible by another parameter; iii) it allows tuning parameters with substantially larger ranges by implementing an optimized search space generation process; and iv) its interface is arguably simpler than the interfaces of current auto-tuning frameworks. We demonstrate ATF's efficacy by comparing it to the state-of-the-art auto-tuning approaches OpenTuner and CLTune, showing better tuning results with less programmer's effort.**

I. 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 multi-core CPUs and many-core GPUs which are often programmed in OpenCL [11] and/or CUDA [14], the programmer has to choose values for performance-critical parameters (a.k.a. *tuning parameters*), e.g., the number of threads (a.k.a. work-items in OpenCL) and the number of thread groups (work-groups in OpenCL and blocks in CUDA).

Auto-tuning is an approach to automatically find optimal values of tuning parameters. To auto-tune a program, the programmer has to identify program's tuning parameters and to perform the following three steps: 1) generate an application-specific *search space* of parameter configurations, i.e., sets of tuning parameter values, 2) implement a *cost function* for estimating program's cost (e.g., its runtime) for a configuration in terms of the target objective, and 3) explore the generated search space using a search technique to find a configuration providing minimal cost. Auto-tuning has been successfully applied in different areas, including ATLAS [20] for linear algebra routines, PATUS [6] for stencil computations, MILEPOST [7] for compiler optimizations, CHiLL [5] and Orio [8] for loop operations, Active Harmony [19] for runtime systems, and Apollo [2] for execution policies. These approaches implement auto-tuning *specifically* for their target domains and cannot be used for other domains.

OpenTuner [1] is a recent approach that is *generic* regarding the application domain: given an arbitrary, user-provided cost function and a specification of the tuning parameters (i.e., their names and the ranges of possible values), OpenTuner automatically generates a search space and explores it in terms of a user-defined objective by using pre-implemented search techniques. However, OpenTuner does not provide mechanisms for expressing dependencies between parameters. For example, the important routine *GEMM (GEneral Matrix Multiplication)* [13] when implemented in OpenCL has various tuning parameters (tile size, work-group size, etc.) which have dependencies between them, e.g., some parameter must be divisible by another parameter [15]. Due to these dependencies, OpenTuner is not capable of tuning GEMM. The OpenTuner community has offered workarounds, e.g., re-designing the user program so that its tuning parameters become independent [9], or setting a penalty value for configurations where the dependencies fail [3]. However, the former workaround usually requires a significant effort from the user, and the latter may cause a poor tuning result as we demonstrate for GEMM in Section VI. Moreover, OpenTuner is optimized for auto-tuning programs with large parameter ranges and, thus, it does not provide search techniques for small ranges, e.g., exhaustive search which finds the provably best result.

CLTune [16] is an 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 objective: it is restricted to auto-tuning OpenCL in terms of runtime performance only. Moreover, CLTune is applicable only for parameters with small ranges due to the time-intensive process of search space generation when tuning parameters are interdependent. Thus, parameter ranges have to be artificially limited by the user, requiring expert knowledge from the user and often leading to non-optimal solutions [1]. We demonstrate that even when such limitations are implemented by an expert, CLTune is still not suitable for auto-tuning GEMM for practice-relevant input sizes as used in the context of deep learning [10].

In this paper, we propose the *Auto-Tuning Framework (ATF)* which combines the following advantages over the state-of-the-art auto-tuning approaches:

i) ATF allows auto-tuning programs in arbitrary programming languages and of arbitrary application domains, using a user-defined search technique and objective; its pre-implemented search techniques suite programs with

978-1-5386-2588-0/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCC-SmartCity-DSS.2017.9

both small and large tuning parameter ranges;

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 due to pre-implemented cost functions for OpenCL, CUDA, and other programming languages different from OpenCL and CUDA.

II. Illustration of ATF

We illustrate the usage of ATF by a simple example: auto-tuning the `saxpy` kernel of the *CLBlast* library [15]. The kernel is written in OpenCL; it takes as its input the input 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 the `saxpy` kernel. For simplicity, we removed switching between single and double precision, as well as using OpenCL vector data types. The kernel is executed on a device (e.g., 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 (e.g., 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 chunk 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 `saxpy` (line 10). OpenCL requires work-items to be grouped in so-called *work-groups*. The number of work-items per work-group is called *local size* (`LS`) which is a further tuning parameter, i.e, it has also to be chosen specifically for the target device. The local size is set in the OpenCL host code when invoking the kernel on a device.

```
1  __kernel void saxpy( const          int      N,
2                       const          float    a,
3                       const __global float*   x,
4                             __global float*   y
5                     )
6  {
7    for( int w = 0; w < WPT; ++w ) {
8      const int index = w * get_global_size(0)
9                          + get_global_id(0);
10
11     y[ index ] += a * x[ index ];
12   }
13 }
```

Listing 1: Simplified `saxpy` kernel from CLBlast.

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 OpenCL specification [11] requires the local size to divide the *global size* — the total number of work-items — which in case of the `saxpy` kernel is `N/WPT`. Analogously 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 optimizing the `saxpy` kernel in Listing 1. For high performance, we auto-tune the two parameters `WPT` and `LS` specifically for a fixed, user-defined input size `N` (line 4). The ATF program (Listing 2) is written in C++ and performs the three steps for auto-tuning, as explained in the following.

```
1  int main()
2  {
3    std::string saxpy_kernel = /* path to kernel of Listing 1   */;
4    int         N            = /* fixed user-defined input size */;
5
6    auto WPT = atf::tp( "WPT",
7                        atf::interval<size_t>(1,N),
8                        atf::divides( N )
9                      );
10   auto LS  = atf::tp( "LS",
11                       atf::interval<size_t>(1,N),
12                       atf::divides( N/WPT )
13                     );
14
15   auto cf_saxpy = atf::cf::ocl(
16                     { "NVIDIA", "Tesla K20c" },
17                     saxpy_kernel,
18                     inputs( atf::scalar<int>(N),   // N
19                             atf::scalar<float>(),  // a
20                             atf::buffer<float>(N), // x
21                             atf::buffer<float>(N), // y
22                           )
23                    atf::glb_size( N/WPT ), atf::lcl_size( LS )
24                   );
25
26   auto best_config = atf::annealing( atf::duration<minutes>(10) )
27                                    ( WPT, LS )
28                                    ( cf_saxpy );
29 }
```

Listing 2: ATF program for auto-tuning the `saxpy` kernel.

Step 1: Generate the Search Space

ATF automatically generates the search space. For this, the user describes the application-specific search space using tuning parameters; in this example: 1) the work-per-thread `WPT` (line 6-9) which is a `size_t` parameter in the interval $[1, N]$ (line 7) that has to divide the input size N (line 8), and 2) the local size `LS` (line 10-13) — a `size_t` parameter in $[1, N]$ (line 11) that divides the global size `N/WPT` (line 12). The generated search space comprises all valid parameter configurations, i.e., for which the parameters' constraints (line 8, 12) are satisfied.
The general form of an ATF tuning parameter is as follows:

```
atf::tp( /* name       */,
         /* range      */,
         /* 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 used to get a parameter's value conveniently out of a configuration. For example, `best_config["LS"]` gets the value of `LS` out of the best found configuration `best_config` in line 26.

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 `atf::interval<T>(begin, end, step_size, generator)` represents the values of an arbitrary fundamental type `T` (e.g., `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 domain-specific parameter ranges and can be any arbitrary C++ callable (i.e., a lambda, a function pointer, or a functor) with the input type `T` and an arbitrary fundamental output type `T'`. When a generator function is used, the range type changes automatically to `T'`, and the interval comprises the elements `generator(i)` for each i from `begin` to `end`. For example, `atf::interval<size_t>(1, 10, [](size_t i){ return std::pow(2,i); })` represents the first ten powers of 2. Here, the generator function is a lambda that takes a `size_t` parameter i and returns the i-th power of 2. For small ranges, it is more convenient to state a range's elements explicitly. For this, ATF provides `atf::set(val_1,...,val_n)` — a set that comprises the elements `val_1,...,val_n`. For convenience, a set can be expressed also as an `std::initializer_list` of the form {`val_1, ... val_n`}. To enable auto-tuning parameters comprising user-defined values, sets may also comprise values of an `enum` type.

Constraints are a major feature of ATF; they enable filtering a tuning parameter's range. A constraint can be any arbitrary C++ callable that takes a value of the parameter's range and returns a value of the type `bool`: values for which the constraint returns `false` are filtered out of the range. Constraints can be used to conveniently define dependencies between tuning parameters: e.g., in line 12 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, `atf::divides( N/WPT )` returns the lambda function `[&]( size_t LS ){ return (N/WPT) % LS == 0; }`. We refer to `atf::divides` as a *constraint alias* which is offered by ATF for programmer's convenience. Currently, ATF provides six constraint aliases: `divides`, `is_multiple_of`, `less_than`, `greater_than`, `equal`, and `unequal`. Further aliases can be easily added to ATF by the user. Constraints can be combined using the logical operators `&&` and `||`.

Step 2: Implement a Cost Function

The user implements a cost function that takes a configuration of tuning parameter values and returns a value of a type for which operator < is defined, e.g., `size_t`. ATF interprets cost function's return value (e.g., program's runtime) as the configuration's cost that has to be minimized.

We enable *multi-objective tuning*, e.g., minimizing first runtime and then energy consumption, by allowing cost functions to have an arbitrary, user-defined return type. For example, to auto-tune for both, runtime performance and low energy consumption, the user chooses pairs as return type. Pairs comprise runtime (e.g, in ms) and energy consumption (e.g., in microjoules) and < is defined as lexicographical order, i.e.: 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'`.

For user's convenience, ATF provides a pre-implemented cost function `atf::cf::ocl` (Listing 2, line 15-24) for auto-tuning OpenCL kernels in terms of runtime performance. In this example, the cost function (line 15) is initialized with the Tesla K20c device of system's NVIDIA platform (line 16) — this choice is arbitrary and could also be any other of system's OpenCL-compatible devices. As the kernel's input, we use the input size N (line 18), a floating point number for a that is randomly generated by ATF (line 19), and two N-sized buffers for x and y that are also filled with random floating point numbers (line 20-21) — random data is the default input when auto-tuning OpenCL kernels. The kernel's global and local size are set in line 23 using ATF's functions `atf::glb_size` and `atf::lcl_size`, correspondingly. The initialized cost function `cf_saxpy` (line 15) then takes configurations comprising concrete values for `WPT` and `LS`, and it returns the `saxpy` kernel's runtime for that concrete `WPT` and `LS`. For this, `cf_saxpy` replaces in kernel's source code the tuning parameters' names by their corresponding values in the input configuration using the OpenCL preprocessor, and it uses pre-implemented OpenCL host code that invokes the kernel with the passed global/local size, and measures and returns the kernel's runtime using the OpenCL profiling API. To avoid the usually time-intensive host-to-device/device-to-host data transfers, we upload data only once during cost function's initialization, and we refrain from downloading the data — auto-tuning is performed on random data, as the computed result is not needed. Optionally, ATF's OpenCL cost function can support error checking for the computed results.

ATF provides also a pre-implemented cost function for auto-tuning CUDA kernels. It is based on the NVIDIA NVRTC library [17] and used analogously to the ATF's OpenCL cost function, with the only difference that platform's name is omitted, because CUDA targets NVIDIA devices only.

Moreover, ATF provides a generic cost function to simplify auto-tuning programs written in an arbitrary programming language, using an arbitrary objective. This function is initialized with: 1) the path to program's source file, 2) paths to two user-provided scripts for compiling and running the program, and optionally 3) the path to a log file to which the user program writes its cost that ATF should minimize; if no log file is stated, ATF automatically measures and uses program's runtime as cost. For multi-objective tuning, the auto-tuned program writes comma-separated costs (e.g., as pairs) to the log file; ATF then minimizes these costs using the lexicographical order, or, alternatively, a user-defined order.

Step 3: Explore the Search Space

The user explores the search space (line 26-28) by choosing a search technique (in this example: simulated annealing [12]) and passing to it: i) an *abort condition*, here `atf::duration` which causes to stop exploration after

10 minutes (line 26), ii) the tuning parameters (line 27), and iii) the cost function (line 28).

Currently, ATF allows the user to choose among three pre-implemented 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 effective for auto-tuning OpenCL and CUDA applications if search spaces are too large to be explored exhaustively [16], and 3) the OpenTuner search which automatically combines various well-proven search techniques, e.g., many variants of Nelder-Mead search (a.k.a. simplex method) and Torczon hillclimbers, to yield a good tuning result on average for applications with large search spaces [1]. For domain-specific requirements, the user can extend ATF by new search techniques as demonstrated in Section IV.

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

1) `duration<D>(t)`: stops exploration after a user-defined time interval `t`; here, `D` is an `std::chrono::duration` and states 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 $\leq$ `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 $\geq$`s`;
6) `speedup(s,n)`: stops when within the last `n` tested configurations the cost could not be lowered by a factor $\geq$`s`.

If no abort condition is passed, 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 easily added to ATF.

III. Comparison: ATF vs. CLTune

We now demonstrate that even though ATF is a generic approach, it is expressive and easy to use. For this, we compare the generic ATF to the framework CLTune [16] that is specifically designed for auto-tuning OpenCL.

In the following, we compare the ATF program for `saxpy` in Listing 2 with the CLTune program for `saxpy` in Listing 3 taken from [4]. We demonstrate that ATF's constraints conveniently express parameter dependencies and that ATF allows tuning parameters of arbitrary types, making ATF more flexible and less memory-intensive. Moreover, we demonstrate that OpenCL's global and local sizes can be expressed in ATF as common arithmetic expressions, thus further contributing to ATF's usability and expressiveness.

In Listing 3, the user chooses a target device, sets the kernel, and prepares the kernel's input data (line 6-24). Afterwards, he defines the `saxpy`-specific tuning parameters (line 26-39) and then starts the tuning process (line 41-43). We now discuss each of the three steps required for auto-tuning.

```
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 )
28     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 auto-tuning the `saxpy` kernel.

Step 1: Generate the Search Space

CLTune allows the user to express parameter interdependencies by boolean 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: the user uses boolean functions to constrain the parameter ranges (Listing 2, line 8, 12), rather than the search space which can become very large. This design decision enables ATF to enormously accelerate the process of search space generation and, consequently, to auto-tune parameters with significantly larger ranges. Moreover, CLTune requires abstractions by vectors for using tuning parameters in boolean functions (Listing 3, line 32, 33) while ATF does not require such abstractions, thus making its use more convenient.

In CLTune, tuning parameters have to be of type `size_t`. In contrast, ATF allows tuning parameters to be of an arbitrary fundamental type (e.g., `bool`, `float`, etc.) and also of type `enum` for user-defined types, making ATF suitable for a broad range of applications with differently-typed tuning parameters. Furthermore, ATF reduces memory utilization since `size_t` has a higher memory consumption than `integer` or `bool`.

Step 2: Implement a Cost Function

ATF tunes program's parameters with respect to an arbitrary cost function, thus allowing to auto-tune programs in arbitrary programming languages for an arbitrary objective (e.g., high performance and/or low energy consumption). In constrast, CLTune is only suitable for auto-tuning OpenCL programs and only in terms of runtime performance. ATF provides a pre-implemented cost function that allows conveniently auto-tuning OpenCL programs (Listing 2, line 15-24). We argue that using ATF for OpenCL is more convenient than CLTune, for the following reasons.

In CLTune, the global and local size 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 to conveniently define the global/local size as an arithmetic expression containing tuning parameters (Listing 2, line 23); the ATF user need not set the global and local size to initial values and modify them later using special functions to make them dependent on tuning parameters. The generality of ATF makes it also 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, i.e., CLBlast has to use a simplified global size for auto-tuning its GEMM kernel, thus causing non-optimal tuning results as we demonstrate in Section VI.

Our OpenCL cost function can generate random input data of arbitrary fundamental data types (e.g., `float`) as usually required for auto-tuning OpenCL kernels. For example, `atf::scalar<T>()` (Listing 2, line 19) generates a random value of type `T` and passes it to the kernel, and `atf::buffer<T>(N)` (line 20-21) generates and passes to the kernel a buffer comprising `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 using concrete input data, he can use `atf::scalar(a)` to pass the scalar `a` of an arbitrary fundamental type to the kernel, and `atf::buffer(vec)` where `vec` is an arbitrary `std::vector<T>` for a fundamental data type `T`.

ATF allows the user to choose a device directly by its platform and device name (Listing 2, line 16), without the inconvenient interactions with the OpenCL API requiring knowledge about special OpenCL flags and performing string operations. In contrast, the CLTune user chooses the target device by its platform's and device's id (Listing 3, line 6) which are usually determined via the OpenCL API using the platform and device name and are dependent on system's configuration, i.e.: CLTune programs may require reconfiguration/recompiling when system's configuration changes, e.g., a new OpenCL implementation is installed, a new device added, etc.

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 [1]. Moreover, ATF allows an arbitrary objective and also multi-objective tuning (see Section II), 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, e.g., to stop the tuning depending on the tuning result (`cost` and `speedup`), and also allows to combine abort conditions by logical operators to meet complex user requirements.

IV. 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:

```
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 argument; it is called by ATF once 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, e.g., to free allocated memory, etc. During exploration, ATF performs repeatedly the following two steps until the chosen abort condition is satisfied: 1) it takes a configuration using the function `get_next_config`, determines the configuration's cost by using the user-provided or pre-implemented cost-function, and 2) it reports the returned cost back to the search technique using `report_cost`.

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.

A. 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 for each call a new configuration within the search space.

B. Simulated Annealing search

For simulated annealing which has proven to be effective for exploring OpenCL and CUDA search spaces [16], the implementation of `initialize` and `finalize` is straightforward: 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 back 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 [16].

C. OpenTuner search

The OpenTuner framework [1] implements various search techniques, e.g., Nelder-Mead and Torczon hillclimbers, selected automatically for a concrete search space. As discussed above, the OpenTuner is not suitable for auto-tuning parameters with dependencies between them. We employ the OpenTuner's search engine as an ATF's search technique, because ATF's search space contains per construction only configurations where parameter dependencies are satisfied (see Section II). For this, we define the OpenTuner tuning parameter `TP` with a range of integers from 1 to `S`, where `S` is the search space size and `TP` is 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 [18]. 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` in each call takes 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 the 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.

V. Advanced ATF Usage

Applications with many tuning parameters (e.g., the matrix multiplication GEMM has 10 tuning parameters) usually comprise different groups of interdependent parameters, allowing ATF to generate search space faster, e.g., 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 comprising values 1 and 2. The constraint of `tp_2`, i.e., `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` make together a group of interdependent parameters, and `tp_3` and `tp_4` comprise a further group. For each group, the corresponding part of the search space can be generated in parallel: parameters `tp_1` and `tp_2` do not influence possible values for `tp_3` and `tp_4`, and vice versa.

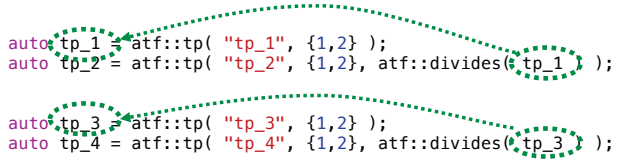

Fig. 1: Example of tuning parameters that allow parallel search space generation.

Currently, ATF cannot automatically determine dependencies between parameters: the user has to group interdependent parameters explicitly using the *grouping function* `G(...)`:

```
atf::/* search technique */( /* abort condition         */ )
                            ( G(tp_1, tp_2) , G(tp_3, tp_4) )
                            ( /* cost function           */ );
```

ATF then generates the search space in parallel using one thread per dependent parameter group. Our parallel implementation is based on the Standard C++ Threading Library.

VI. Experimental Results

In this section, we demonstrate that ATF provides better tuning results for GEMM (GEneral Matrix Multiplication) than the state-of-the-art auto-tuning approaches CLTune and OpenTuner.

As concrete GEMM implementation, we take the `XgemmDirect` kernel; it is part of the auto-tunable OpenCL BLAS library CLBlast which uses CLTune for auto-tuning. The kernel is optimized for small matrix sizes of up to $2^{10} \times 2^{10}$ and is used to accelerate important applications, e.g., the state-of-the-art deep learning framework Caffe [10]. It has 10 tuning parameters, with the following ranges for $N \times N$ input matrices [15]:

- 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 $\{\texttt{true}, \texttt{false}\}$.

Here, `WGD` represents the tile size and `KWID` the loop unrolling factor. The parameters have various interdependencies (17 in total), e.g., `KWID` has to divide `WGD`, and `WGD` has to divide result matrix's number of rows and number of columns, respectively.

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 K20m 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 with CLTune version 2.6.0 and OpenTuner version 0.7.0. The `XgemmDirect` kernel is extracted from CLBlast version 0.11.0.

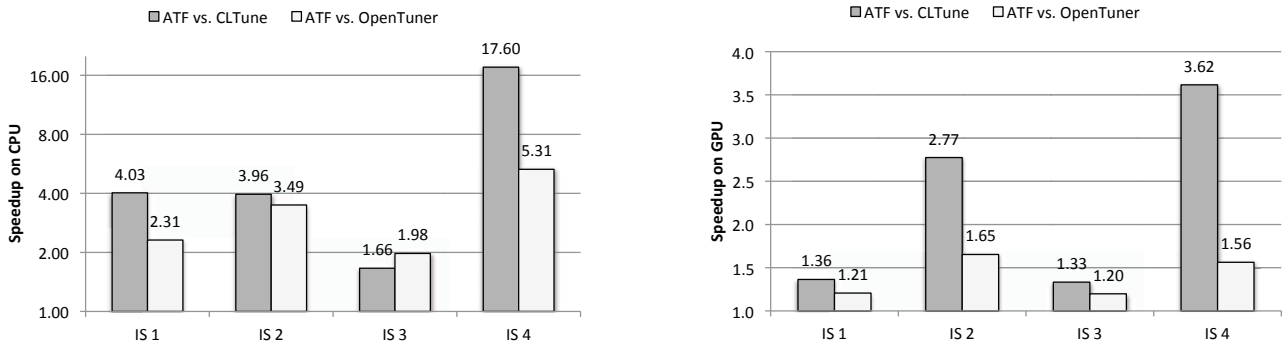

Fig. 2: Speedup (higher is better) of the `XgemmDirect` kernel auto-tuned by ATF over auto-tuning by CLTune and OpenTuner on Intel CPU (left) and NVIDIA GPU (right), using four different input sizes IS1-IS4.

Figure 2 shows the measured speedup of the `XgemmDirect` kernel auto-tuned by ATF as compared to the kernel auto-tuned with CLTune and OpenTuner, correspondingly. We study four pairs of matrix input sizes (IS) that are heavily used in Caffe [10], e.g., 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 auto-tuning `XgemmDirect` [15], and we implement the OpenTuner program for this kernel according to [3], 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.

A. ATF vs. CLTune

In Figure 2, we observe that, in comparison to CLTune, ATF improves `XgemmDirect`'s runtime by factors from $1.66\times$ to $17.60\times$ on the CPU (left part of the figure, logarithmic scale), and from $1.33\times$ to $3.62\times$ 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 [15]. Due to this constraint, the range limitation of `WGD` causes search space to be empty for the matrix sizes used in deep learning: either 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 (i.e., 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 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 II. In contrast, CLTune iterates over the entire, unconstrained search space and then filters 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 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 size more generally than CLTune (see Section III). 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 III). 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 size 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 $12.85\times$ to $17.60\times$ on the CPU, and from $2.89\times$ to $3.62\times$ on the GPU.

B. ATF vs. OpenTuner

In Figure 2, comparing ATF and OpenTuner, we can observe that ATF speedups the `XgemmDirect` kernel by factors

from $1.98\times$ to $5.31\times$ on the CPU (left), and by factors from $1.20\times$ to $1.65\times$ 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 — i.e., 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 has determined with CLTune. This is because the default parameter values are small, e.g., `WGD=8` and `KWID=1`, causing a high parallelization of `XgemmDirect` for the special input sizes as used in deep learning.

VII. 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 which are written in an arbitrary programming language and which belong to an arbitrary application domain; tuning parameters may be interdependent. The user can auto-tune for an arbitrary objective (e.g., high runtime performance and/or low energy consumption) and choose among three pre-implemented search techniques, thus targeting search spaces of different size. For domain-specific user requirements, ATF can be easily extended by further search techniques. We demonstrate that ATF is easy to use, thus making auto-tuning appealing to common application developers. Our experimental results show that ATF provides better tuning results for General Matrix Multiplication (GEMM) written in OpenCL compared to CLTune which is currently used for auto-tuning GEMM in OpenCL on important input sizes as used in the application area of deep learning.

Acknowledgements

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

References

[1] Ansel J, Kamil S et al. (2014) OpenTuner: An Extensible Framework for Program Autotuning. In: Proc. of the 23rd Int. Conf. on Parallel Architectures and Compilation, ACM, pp 303–316

[2] Beckingsale D, Pearce O et al.(2017) Apollo: Reusable Models for Fast, Dynamic Tuning of Input-Dependent Code. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp 307–316

[3] Bruel P, Amars M et al. (2017) Autotuning CUDA Compiler Parameters for Heterogeneous Applications using the OpenTuner Framework. Concurrency and Computation: Practice and Experience pp 1–13

[4] Cedric Nugteren (2016) CLTune Issue 48. URL github.com/CNugteren/CLTune/issues/48

[5] Chen C, Chame J et al. (2008) CHiLL: A Framework for Composing High-Level Loop Transformations. Tech. rep., Technical Report 08-897, U. of Southern California

[6] Christen M, Schenk O et al. (2011) PATUS: A Code Generation and Autotuning Framework For Parallel Iterative Stencil Computations on Modern Microarchitectures. In: Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International, IEEE, pp 676–687

[7] Fursin G, Kashnikov Y et al (2011) Milepost GCC: Machine Learning Enabled Self-tuning Compiler. International Journal of Parallel Programming 39(3):296–327

[8] Hartono A, Norris B et al. (2009) Annotation-Based Empirical Performance Tuning Using Orio. In: Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, IEEE, pp 1–11

[9] Jason Jansel (2016) OpenTuner Issue 87. URL github.com/jansel/opentuner/issues/87

[10] Jia Y, Shelhamer E et al. (2014) Caffe: Convolutional Architecture for Fast Feature Embedding . In: Proceedings of the 22nd ACM international conference on Multimedia, ACM, pp 675–678

[11] Khronos OpenCL Working Group (2017) The OpenCL Specification. URL khronos.org/opencl/

[12] Kirkpatrick S, Gelatt CD et al (1983) Optimization by Simulated Annealing. Science 220(4598):671–680

[13] Netlib (2016) BLAS. URL netlib.org/blas/

[14] Nickolls J, Buck I et al. (2008) Scalable Parallel Programming With CUDA. Queue 6(2):40–53

[15] Nugteren C (2017) CLBlast: A Tuned OpenCL BLAS Library. arXiv preprint arXiv:170505249

[16] Nugteren C, Codreanu V (2015) CLTune: A Generic Auto-Tuner for OpenCL Kernels. In: Embedded Multicore/Many-core Systems-on-Chip (MCSoC), IEEE, pp 195–202

[17] NVIDIA (2017) NVRTC. URL docs.nvidia.com/cuda/nvrtc/

[18] Python Software Foundation (2017) Python Embedding API. URL docs.python.org

[19] Ţăpuş C, Chung IH et al (2002) Active Harmony: Towards Automated Performance Tuning. In: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, IEEE Computer Society Press, pp 1–11

[20] Whaley RC, Dongarra JJ (1998) Automatically Tuned Linear Algebra Software. In: Proc. of the 1998 ACM/IEEE Conf. on Supercomputing, IEEE Computer Society, pp 1–27