Universität
Münster


Richard
Schulze


Sergei
Gorlatch

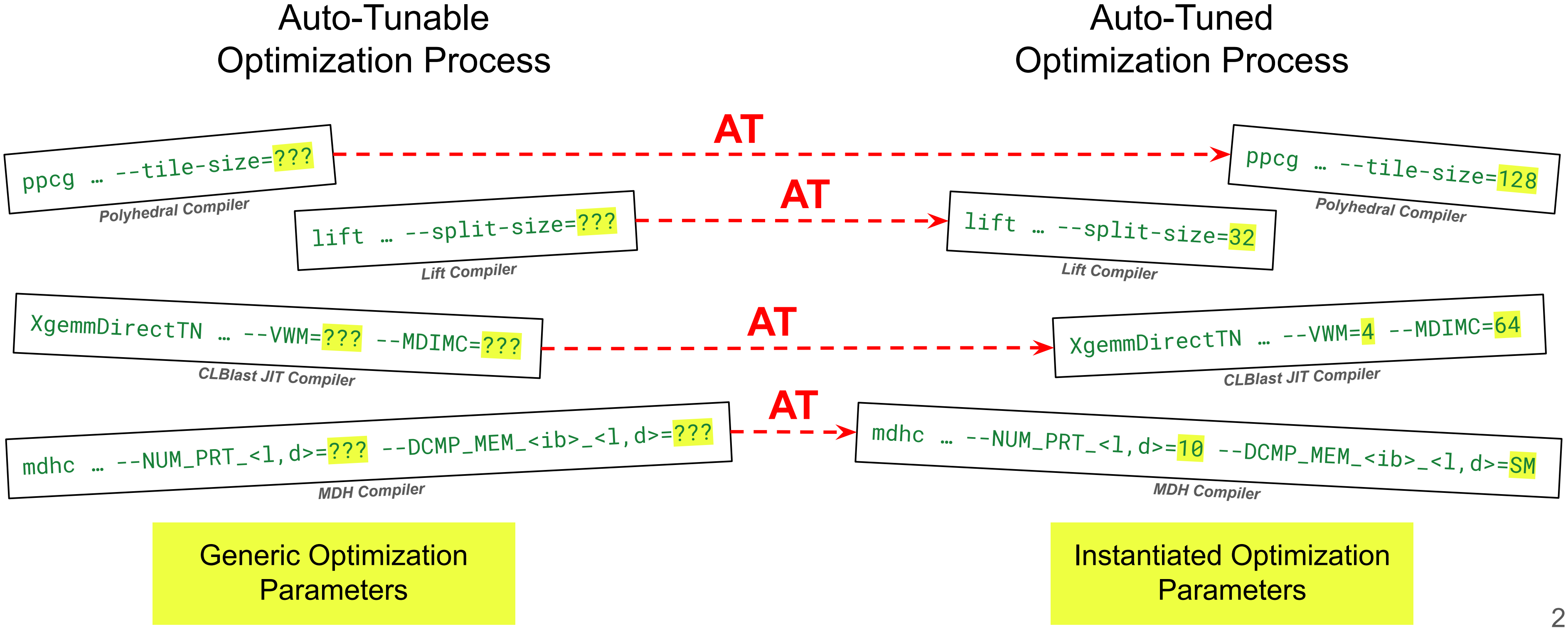

Ari
Rasch

pyATF: Constraint-Based Auto-Tuning in Python

Richard Schulze, Sergei Gorlatch, Ari Rasch

What is *Auto-Tuning*?

Auto-Tuning (AT) automatically finds optimized values of performance-critical parameters:



Auto-Tunable Optimization Process

Auto-Tuned Optimization Process

`ppcg … --tile-size=???` **AT** → `ppcg … --tile-size=128`

Polyhedral Compiler

`lift … --split-size=???` **AT** → `lift … --split-size=32`

Lift Compiler

`XgemmDirectTN … --VWM=??? --MDIMC=???` **AT** → `XgemmDirectTN … --VWM=4 --MDIMC=64`

CLBlast JIT Compiler

`mdhc … --NUM_PRT_<l,d>=??? --DCMP_MEM_<ib>_<l,d>=???` **AT** → `mdhc … --NUM_PRT_<l,d>=10 --DCMP_MEM_<ib>_<l,d>=SM`

MDH Compiler

Generic Optimization Parameters

Instantiated Optimization Parameters

What are *Constraints*?

Independent Parameters

Parameters:

`tile_size ∈ {1,...,128}`

`mem_region ∈ {GLB,LCL,PRV}`

`num_threads ∈ {1,...,1024}`

Constraints:

`<none>`

Configurations:

`{ (1,GLB,1) ,..., (128,PRV,1024) }`

Any combination of parameter's values represents a valid parameter configuration

Interdependent Parameters

Parameters:

`tile_size_1 ∈ {1,...,128}`

`tile_size_2 ∈ {1,...,128}`

`tile_size_3 ∈ {1,...,128}`

Constraints:

`tile_size_3 | tile_size_2 | tile_size_1`

Configurations:

{ (1,1,1) , ~~(1,1,2)~~ ,..., (128,128,128) }

tile_size_2* not multiple of *tile_size_3

Only combinations that satisfy the constraints represent valid configurations

State-of-the-Art Auto-Tuners

Framework	Year	API	Constr.	Targets
OpenTuner [PACT'14]	2014	Python	╳	*
CLTune [MCSoC'15]	2015	C++	(✔)	OpenCL
Kernel Tuner	2019	Python	(✔)	OpenCL, CUDA, …
HyperMapper [MASCOTS'19]	2019	JSON	╳	*
KTT	2020	C++	(✔)	OpenCL, CUDA, …
ytopt	2021	Python	(✔)	*
ATF [TACO'21]	2021	DSL	✔	*
BaCO [ASPLOS'23]	2023	JSON	✔	*

✓ : strong support for constraints
(✓) : limited support for constraints
X : no support for constraints
* : support for arbitrary prog. lang.

State-of-the-Art auto-tuners differ in their
API & support for constraints & target programming languages

Goal of this Work

pyATF-Website

Framework	Year	API	Constr.	Targets
OpenTuner [PACT'14]	2014	Python	✗	*
CLTune [MCSoC'15]	2015	C++	(✔)	OpenCL
Kernel Tuner	2019	Python	(✔)	OpenCL, CUDA, …
HyperMapper [MASCOTS'19]	2019	JSON	✗	*
KTT	2020	C++	(✔)	OpenCL, CUDA, …
ytopt	2021	Python	(✔)	*
ATF [TACO'21]	2021	DSL	✔	*
BaCO [ASPLOS'23]	2023	JSON	✔	*
pyATF (this work)	**2024**	**Python**	✔	*

python™

Auto-Tuning Framework (ATF)

Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF)

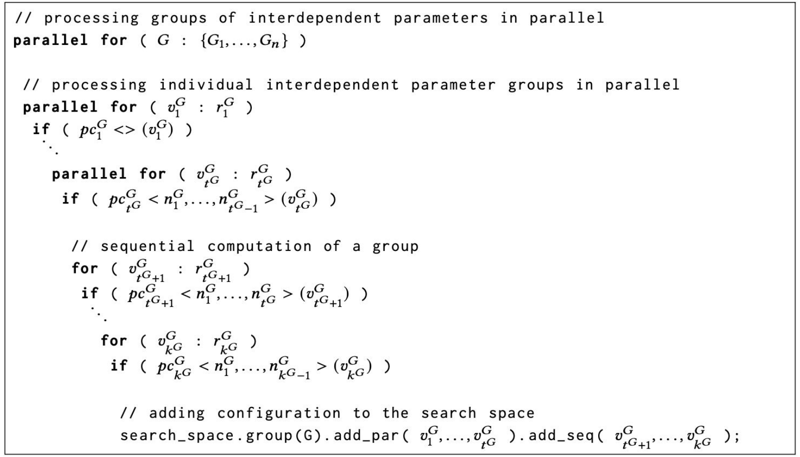
ARI RASCH and RICHARD SCHULZE, University of Muenster, Germany
MICHEL STEUWER, University of Edinburgh, United Kingdom
SERGEI GORLATCH, University of Muenster, Germany **ACM TACO 2021**

Combine advantages of related approaches &
hide them behind a *convenient interface for auto-tuning*

Recap: The ATF Approach

pyATF-Website

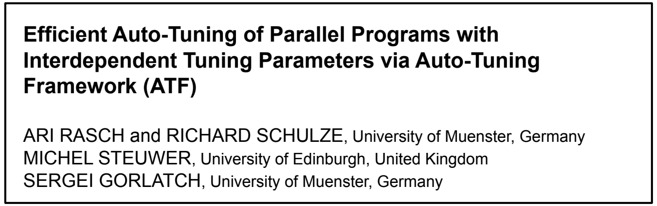
ATF introduces novel processes to
Generating & Storing & Exploring
the search spaces of constrained tuning
parameters, based on a novel
Constraint Design & Search Space Structure



```
// processing groups of interdependent parameters in parallel
parallel for ( $G$ : $\{G_1,\dots,G_n\}$ )

 // processing individual interdependent parameter groups in parallel
 parallel for ( $v^G_1$ : $r^G_1$ )
  if ( $pc^G_1 <> (v^G_1)$ )
   ⋱
    parallel for ( $v^G_{t^G}$ : $r^G_{t^G}$ )
     if ( $pc^G_{t^G} < n^G_1,\dots,n^G_{t^G-1} > (v^G_{t^G})$ )

      // sequential computation of a group
      for ( $v^G_{t^G+1}$ : $r^G_{t^G+1}$ )
       if ( $pc^G_{t^G+1} < n^G_1,\dots,n^G_{t^G} > (v^G_{t^G+1})$ )
        ⋱
         for ( $v^G_{k^G}$ : $r^G_{k^G}$ )
          if ( $pc^G_{k^G} < n^G_1,\dots,n^G_{k^G-1} > (v^G_{k^G})$ )

           // adding configuration to the search space
           search_space.group(G).add_par( $v^G_1,\dots,v^G_{t^G}$ ).add_seq( $v^G_{t^G+1},\dots,v^G_{k^G}$ );
```

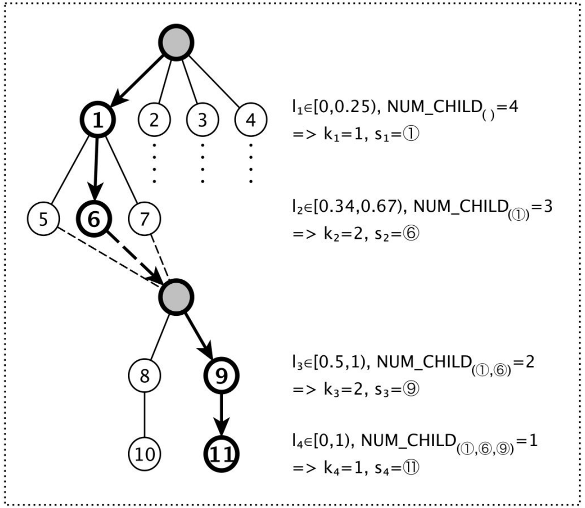
Novel
Constraint Design

ACM TACO 2021



Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF)

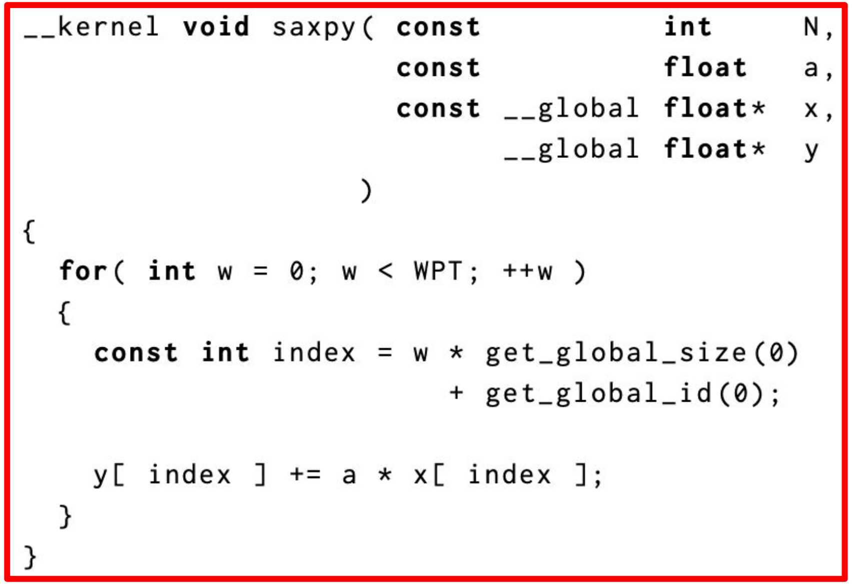
ARI RASCH and RICHARD SCHULZE, University of Muenster, Germany
MICHEL STEUWER, University of Edinburgh, United Kingdom
SERGEI GORLATCH, University of Muenster, Germany



*Novel **CoT***
Search Space Structure

pyATF-Website

Illustration of pyATF



```
__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 ];
  }
}
```

OpenCL SAXPY

- We illustrate pyATF by showing how it is used for auto-tuning SAXPY in OpenCL

- SAXPY has two tuning parameters:
  - `WPT: [1,N]`, has to divide `N`
  - `LS:  [1,N]`, has to divide `N/WPT`

- pyATF works in three steps, described in the following

Illustration of pyATF

pyATF-Website

```
# Input Size
N = 1000

# Step 1: Generate the Search Space
WPT = TP( 'WPT'                      ,
          Interval( 1,N )            ,
          lambda WPT: N % WPT == 0 )
LS  = TP( 'LS'                             ,
          Interval( 1,N )                  ,
          lambda WPT,LS: (N/WPT) % LS == 0 )

# Step 2: Implement a Cost Function
saxpy_code = "..."
N = np.int32(N)
a = np.float32(np.random.random())
x = np.random.rand(N).astype(np.float32)
y = np.random.rand(N).astype(np.float32)
cf = opencl.CostFunction( saxpy_code )         \
           .platform_id( 0 )                   \
           .device_id( 0 )                     \
           .kernel_args( N, a,x,y )            \
           .glb_size( lambda WPT,LS: N/WPT )   \
           .lcl_size( lambda LS: LS          )

# Step 3: Explore the Search Space
config = Tuner().tuning_parameters( WPT,LS )     \
                .search_technique( AUCBandit() ) \
                .tune( cf, Evaluations(50) )
```

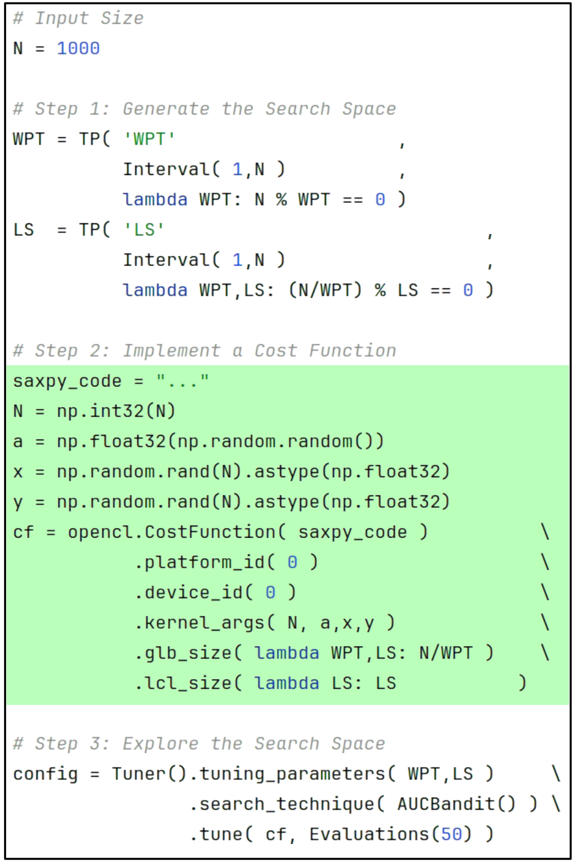
pyATF program for SAXPY

Step 1: Generate the Search Space

- The search space is generated using tuning parameters:
  - `WPT:` `[1,N]`, has to divide N
  - `LS:` `[1,N]`, has to divide `N/WPT`
- A pyATF tuning parameter consists of a
  - name: any arbitrary identifier
  - range: either `Interval` or `Set`
  - constraint: any arbitrary Python callable
- Special features:
  - `Interval` may have a generator function, e.g., `Interval( 1,10 , pow2)`, for `pow2 = lambda i: 2**i`, to get the first ten powers of two
  - Constraint functions may contain tuning parameters, e.g., `WPT` in constraint of `LS`

Illustration of pyATF

pyATF-Website



```
# Input Size
N = 1000

# Step 1: Generate the Search Space
WPT = TP( 'WPT'                          ,
          Interval( 1,N )                ,
          lambda WPT: N % WPT == 0 )
LS  = TP( 'LS'                                   ,
          Interval( 1,N )                        ,
          lambda WPT,LS: (N/WPT) % LS == 0 )

# Step 2: Implement a Cost Function
saxpy_code = "..."
N = np.int32(N)
a = np.float32(np.random.random())
x = np.random.rand(N).astype(np.float32)
y = np.random.rand(N).astype(np.float32)
cf = opencl.CostFunction( saxpy_code )          \
           .platform_id( 0 )                    \
           .device_id( 0 )                      \
           .kernel_args( N, a,x,y )             \
           .glb_size( lambda WPT,LS: N/WPT )    \
           .lcl_size( lambda LS: LS           )

# Step 3: Explore the Search Space
config = Tuner().tuning_parameters( WPT,LS )     \
                .search_technique( AUCBandit() ) \
                .tune( cf, Evaluations(50) )
```

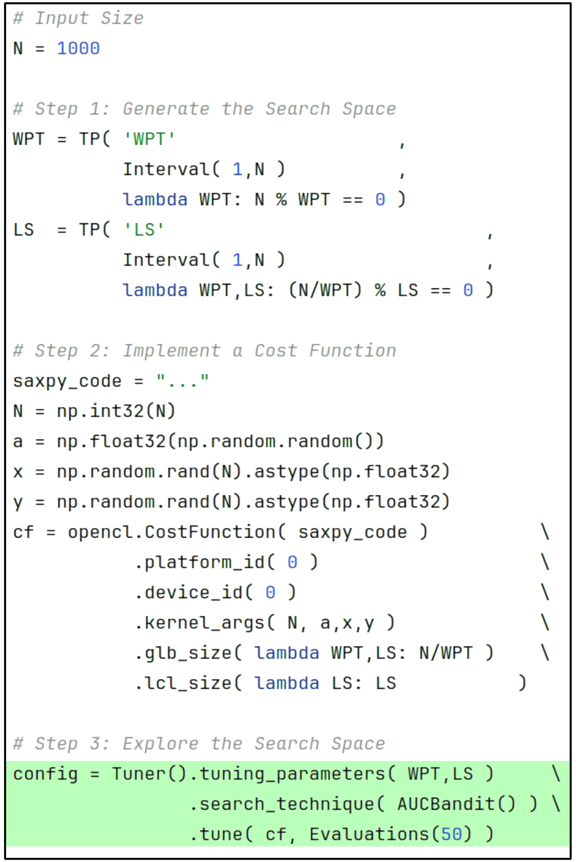
pyATF program for SAXPY

Step 2: Implement a Cost Function

- pyATF allows any arbitrary Python callable as cost function that takes as input a particular configuration of tuning parameter values and returns the cost to be minimized (e.g., runtime and/or energy consumption)
- pyATF provides pre-implemented cost functions for:
    - OpenCL
    - CUDA
    - arbitrary programming languages

Illustration of pyATF

pyATF-Website



```
# Input Size
N = 1000

# Step 1: Generate the Search Space
WPT = TP( 'WPT'                    ,
          Interval( 1,N )          ,
          lambda WPT: N % WPT == 0 )
LS  = TP( 'LS'                             ,
          Interval( 1,N )                  ,
          lambda WPT,LS: (N/WPT) % LS == 0 )

# Step 2: Implement a Cost Function
saxpy_code = "..."
N = np.int32(N)
a = np.float32(np.random.random())
x = np.random.rand(N).astype(np.float32)
y = np.random.rand(N).astype(np.float32)
cf = opencl.CostFunction( saxpy_code )           \
           .platform_id( 0 )                     \
           .device_id( 0 )                       \
           .kernel_args( N, a,x,y )              \
           .glb_size( lambda WPT,LS: N/WPT )     \
           .lcl_size( lambda LS: LS            )

# Step 3: Explore the Search Space
config = Tuner().tuning_parameters( WPT,LS )      \
                .search_technique( AUCBandit() ) \
                .tune( cf, Evaluations(50) )
```

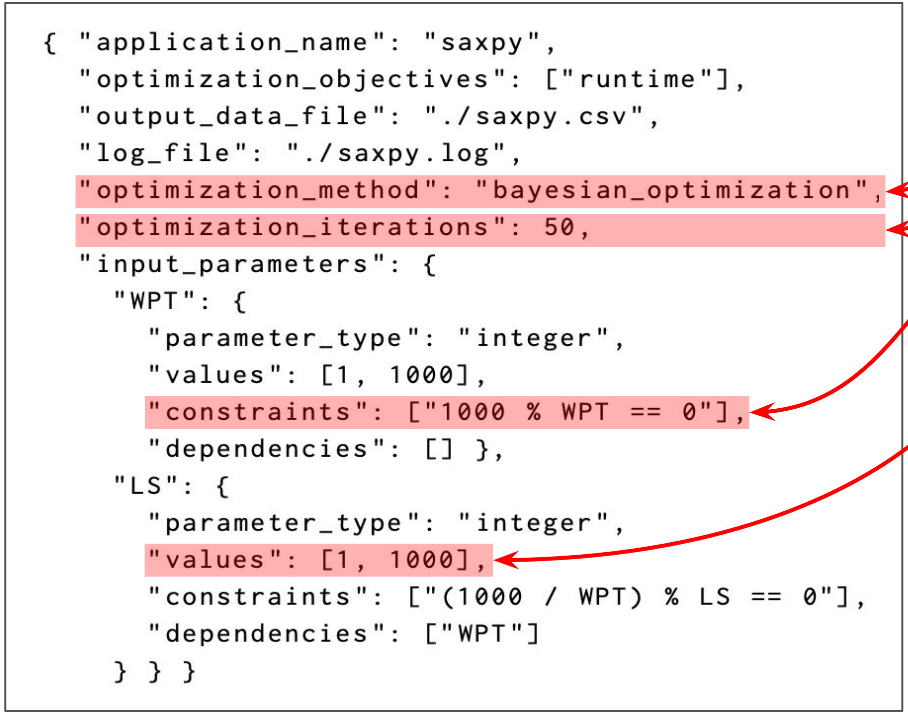
pyATF program for SAXPY

Step 3: Explore the Search Space

- pyATF provides different kinds of pre-implemented search techniques & abort conditions:
  - **basic search techniques:**
    1) Differential Evolution, 2) Pattern Search, 3) Simulated Annealing, 4) Torczon, 5) Exhaustive, 6) Random
  - **meta search techniques:**
    1) AUC Bandit, 2) Round Robin
  - **abort conditions based on tuning time:**
    1) Duration, 2) Evaluations, 3) Fraction
  - **abort conditions based on tuning result:**
    1) Cost
  - **abort conditions based on both:**
    1) Speedup, 2) arbitrarily complex combinations via logical operators
- pyATF is designed generically: new search techniques & abort conditions can easily be added

Interface: pyATF vs. **BaCO**

pyATF-Website



```
{ "application_name": "saxpy",
  "optimization_objectives": ["runtime"],
  "output_data_file": "./saxpy.csv",
  "log_file": "./saxpy.log",
  "optimization_method": "bayesian_optimization",
  "optimization_iterations": 50,
  "input_parameters": {
    "WPT": {
      "parameter_type": "integer",
      "values": [1, 1000],
      "constraints": ["1000 % WPT == 0"],
      "dependencies": [] },
    "LS": {
      "parameter_type": "integer",
      "values": [1, 1000],
      "constraints": ["(1000 / WPT) % LS == 0"],
      "dependencies": ["WPT"]
    } } }
```

BaCO program for SAXPY

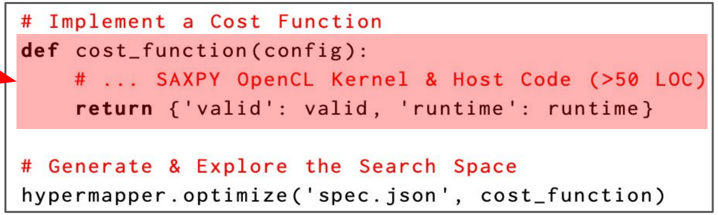
designed toward Bayesian Optimization

only two kinds of abort conditions

limited to expressions in NumExpr

no Set type &
only supports one type of generator function

no pre-implemented cost functions

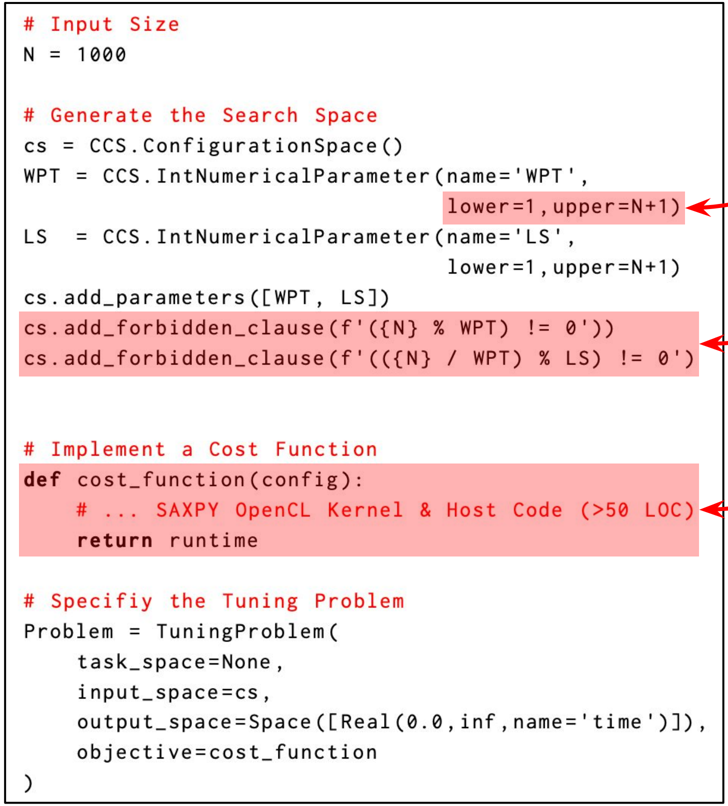


```
# Implement a Cost Function
def cost_function(config):
    # ... SAXPY OpenCL Kernel & Host Code (>50 LOC)
    return {'valid': valid, 'runtime': runtime}

# Generate & Explore the Search Space
hypermapper.optimize('spec.json', cost_function)
```

BaCO Python execution program

Interface: pyATF vs. ytopt

pyATF-Website



```
# Input Size
N = 1000

# Generate the Search Space
cs = CCS.ConfigurationSpace()
WPT = CCS.IntNumericalParameter(name='WPT',
                                lower=1,upper=N+1)
LS  = CCS.IntNumericalParameter(name='LS',
                                lower=1,upper=N+1)
cs.add_parameters([WPT, LS])
cs.add_forbidden_clause(f'({N} % WPT) != 0'))
cs.add_forbidden_clause(f'(({N} / WPT) % LS) != 0')


# Implement a Cost Function
def cost_function(config):
    # ... SAXPY OpenCL Kernel & Host Code (>50 LOC)
    return runtime

# Specifiy the Tuning Problem
Problem = TuningProblem(
    task_space=None,
    input_space=cs,
    output_space=Space([Real(0.0,inf,name='time')]),
    objective=cost_function
)
```

ytopt program for SAXPY

no Set type &
only supports one type of generator function

constraints defined on configurations
(severely limits ytopt's auto-tuning efficiency)

no pre-implemented cost functions

one kind of abort condition only

supports only Bayesian Optimization

Experimental Results

pyATF-Website

Case Studies:

1. `CCSD(T)` → Quantum Chemistry
2. `CONV` → Stencil
3. `PRL` → Data Mining
4. `GEMM` → Deep Learning

Data Characteristics:

1. `CCSD(T)` → `TCCG`
2. `CONV` → `ImageNet`
3. `PRL` → `EKR`
4. `GEMM` → `ResNet50`

Architectures:

NVIDIA Ampere A100 GPU

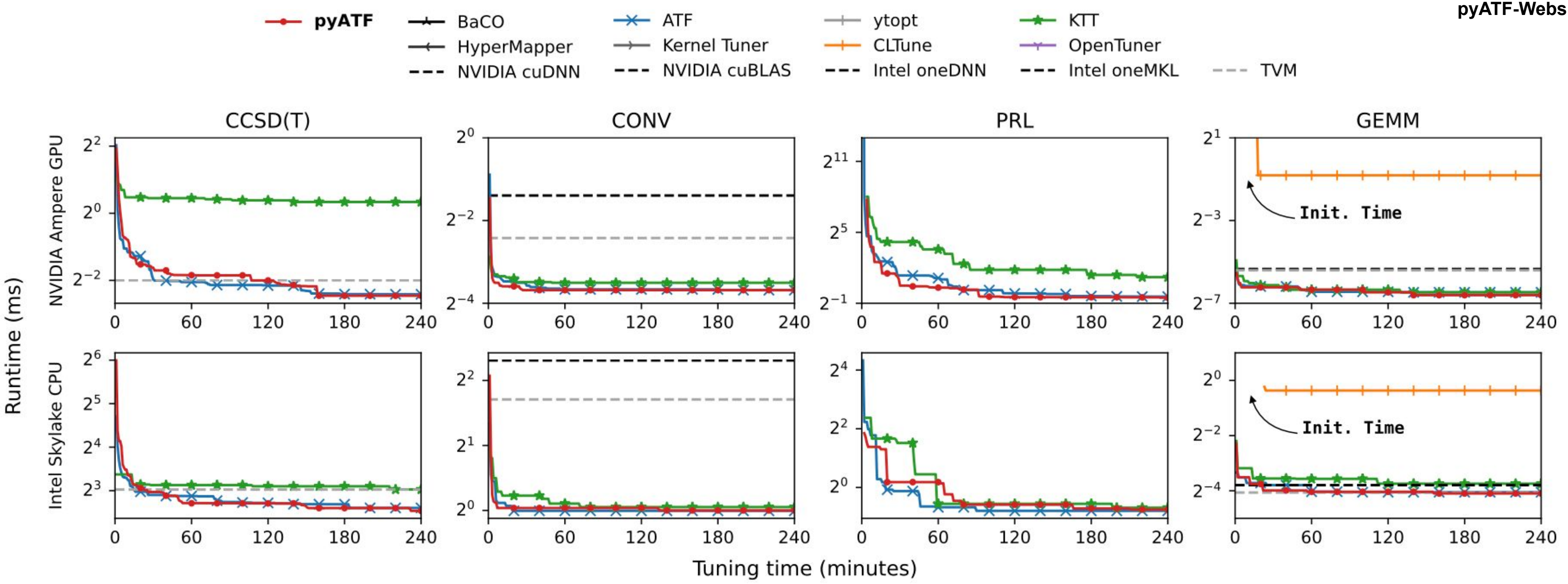
Intel Skylake Xeon Gold CPU

Search Space Characteristics:

	Name	**Domain**	**Input Size**	**#TP**	**Min. RS**	**Max. RS**	**Avg. RS**	**|SP|**	**FT**
1	`CCSD(T)`	`Quantum Chemistry`	$24 \times 16 \times 16 \times 24 \times 16 \times 16 \times 24$	39	2	24	15.46	$8.81 * 10^{18}$	$4.41 * 10^{-25}$
2	`CONV`	`Image Processing`	4288×2848	15	2	4288	2379.33	$2.03 * 10^{8}$	$2.33 * 10^{-29}$
3	`PRL`	`Data Mining`	1024×1024	14	2	1024	586.14	$2.31 * 10^{7}$	$1.33 * 10^{-19}$
4	`GEMM`	`Deep Learning`	$1 \times 1000 \times 2048$	19	1	2048	642.89	$1.08 * 10^{8}$	$6.34 * 10^{-21}$

Experimental Results

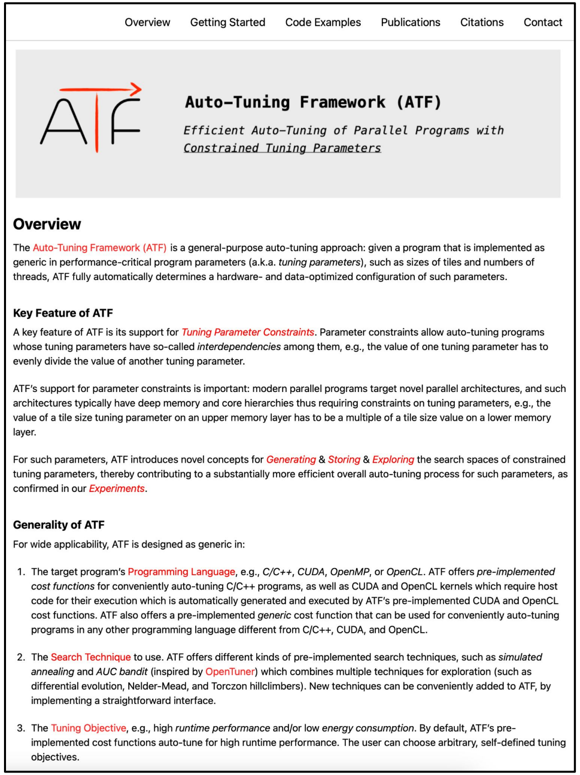
pyATF-Website



Even though pyATF relies on an easy-to-use, Python-based user interface, <u>it achieves high-quality auto-tuning results</u>

Conclusion

pyATF-Website



Auto-Tuning Framework (ATF)

Efficient Auto-Tuning of Parallel Programs with Constrained Tuning Parameters

Overview

The Auto-Tuning Framework (ATF) is a general-purpose auto-tuning approach: given a program that is implemented as generic in performance-critical program parameters (a.k.a. *tuning parameters*), such as sizes of tiles and numbers of threads, ATF fully automatically determines a hardware- and data-optimized configuration of such parameters.

Key Feature of ATF

A key feature of ATF is its support for *Tuning Parameter Constraints*. Parameter constraints allow auto-tuning programs whose tuning parameters have so-called *interdependencies* among them, e.g., the value of one tuning parameter has to evenly divide the value of another tuning parameter.

ATF's support for parameter constraints is important: modern parallel programs target novel parallel architectures, and such architectures typically have deep memory and core hierarchies thus requiring constraints on tuning parameters, e.g., the value of a tile size tuning parameter on an upper memory layer has to be a multiple of a tile size value on a lower memory layer.

For such parameters, ATF introduces novel concepts for *Generating* & *Storing* & *Exploring* the search spaces of constrained tuning parameters, thereby contributing to a substantially more efficient overall auto-tuning process for such parameters, as confirmed in our *Experiments*.

Generality of ATF

For wide applicability, ATF is designed as generic in:

1. The target program's Programming Language, e.g., *C/C++*, *CUDA*, *OpenMP*, or *OpenCL*. ATF offers *pre-implemented cost functions* for conveniently auto-tuning C/C++ programs, as well as CUDA and OpenCL kernels which require host code for their execution which is automatically generated and executed by ATF's pre-implemented CUDA and OpenCL cost functions. ATF also offers a pre-implemented *generic* cost function that can be used for conveniently auto-tuning programs in any other programming language different from C/C++, CUDA, and OpenCL.
2. The Search Technique to use. ATF offers different kinds of pre-implemented search techniques, such as *simulated annealing* and *AUC bandit* (inspired by OpenTuner) which combines multiple techniques for exploration (such as differential evolution, Nelder-Mead, and Torczon hillclimbers). New techniques can be conveniently added to ATF, by implementing a straightforward interface.
3. The Tuning Objective, e.g., high *runtime performance* and/or low *energy consumption*. By default, ATF's pre-implemented cost functions auto-tune for high runtime performance. The user can choose arbitrary, self-defined tuning objectives.

https://atf-tuner.org

- pyATF introduces a productive user-interface, meeting real-world demands (e.g., regarding abort conditions)
- pyATF combines major advantages of state-of-the-art approaches:
  1. implemented in Python
  2. supports constraints
  3. supports arbitrary programming languages
- pyATF works immediately out-of-the-box, is publicly available and open source, can be conveniently installed via Python's package manager `pip`, and is extensively documented and illustrated on its website
- pyATF already successfully used!

GitHub https://github.com/atf-tuner/pyATF

We have a talk at C4ML!
Room: Willow, Time: 3:30pm

Questions?

pyATF-Website

GitHub

https://github.com/atf-tuner/pyATF

https://richardschulze.net
r.schulze@uni-muenster.de

Richard Schulze

https://arirasch.net
a.rasch@uni-muenster.de

Ari Rasch