Westfälische Wilhelms-Universität Münster

ATF: A Generic Auto-Tuning Framework

Ari Rasch and Sergei Gorlatch

University of Münster, Germany

Auto-Tuning

What is Auto-Tuning?

Auto-tuning is an approach for automatically optimizing programs: values of performance-critical parameters (a.k.a. *tuning parameters*) are chosen by using an automatized search technique, e.g., the number of threads.

Why is Auto-Tuning useful?

- Manually choosing tuning parameter values is hard.
- Optimal values of tuning parameters (usually) differs over devices.

Simple Example: SAXPY in OpenCL

- SAXPY is a BLAS routine.
- It takes as its input a scalar a, and two input vectors x and y; it computes:

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

<u>SAXPY in OpenCL:</u>

- Each thread (a.k.a. work-item) performs SAXPY on a chunk of WPT-many elements.
- WPT (Work per Thread) is a <u>tuning parameter</u> of the SAXPY kernel.
- The threads are grouped in work-groups.
- The work-group size (a.k.a local size LS) is a further <u>tuning parameter</u> of the SAXPY kernel.

```
__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 ];
  }
}
```

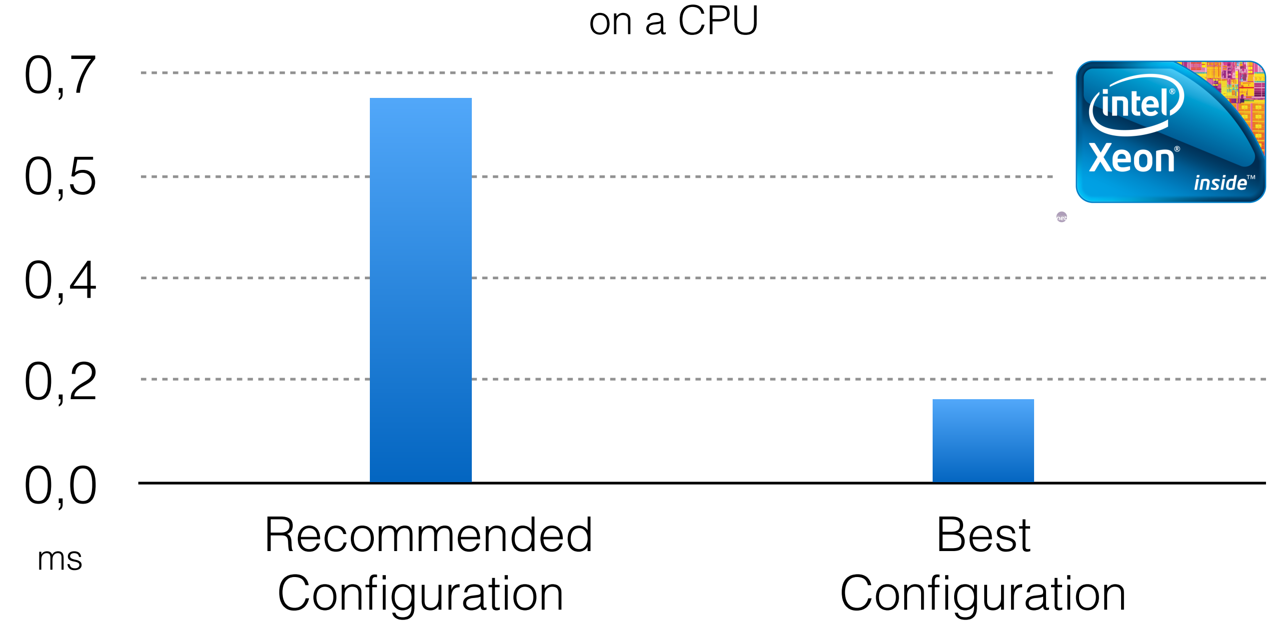
simplified saxpy kernel of the auto-tunable OpenCL BLAS library CLBlast

Simple Example: SAXPY in OpenCL

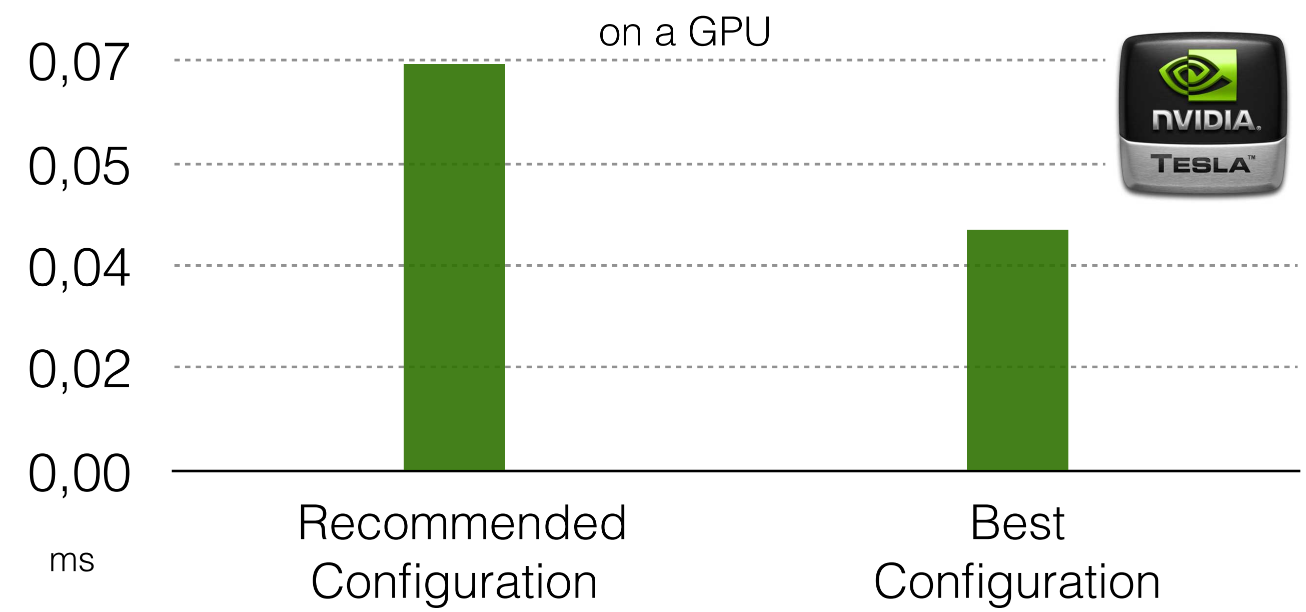
<u>Manually choosing tuning parameter values is hard:</u>

- <u>Intel's recommendation for CPU:</u>
  - start one work-groups for each of CPU's cores;
  - the local size should be 8 (or a multiple of 8) enabling SIMD vectorization.
- <u>NVIDIA's recommendation for GPU:</u>
  - start as many threads as possible to exploit GPU's massive parallelism;
  - the local size should be 32 (or a multiple of 32) to efficiently utilize GPU's *Warps.*

⇒ **Best tuning parameter values are not obvious!**



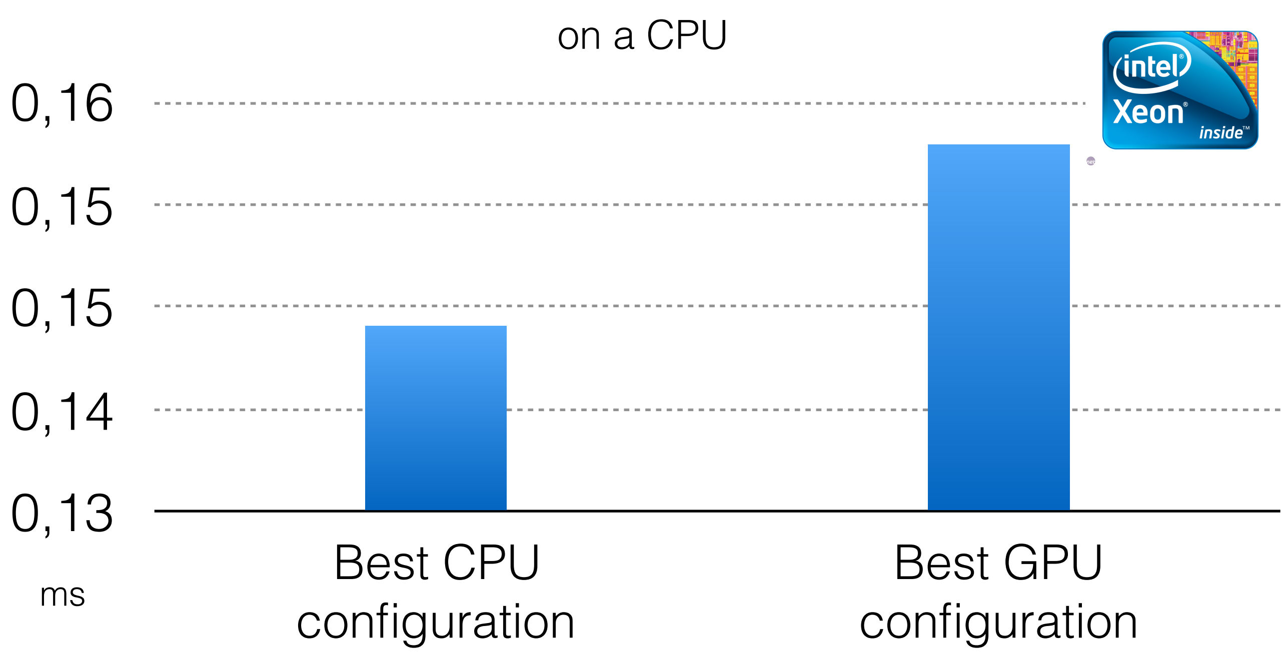
N = 400,000	Recommended	Best
WPT	6250	10
LS	8	2500

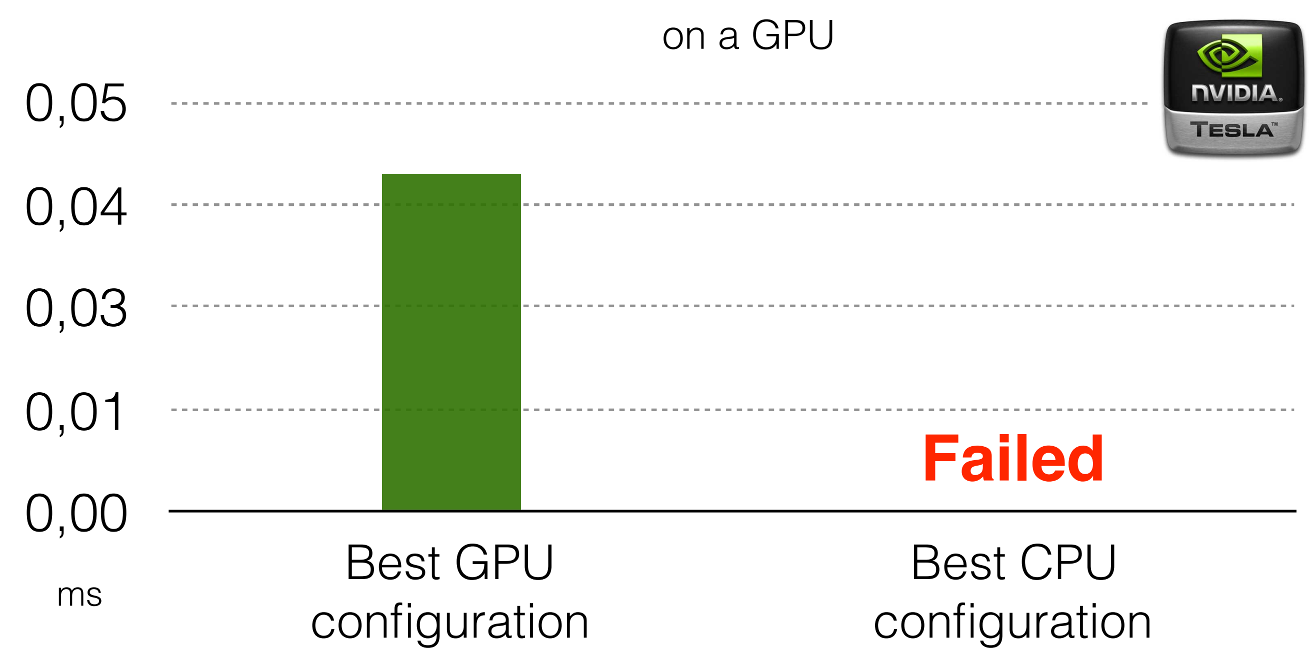


N = 400,000	Recommended	Best
WPT	1	4
LS	32	160

Simple Example:
SAXPY in OpenCL

<u>Optimal values of tuning parameters differs over devices:</u>





Configuration with best performance on GPU has not best performance on CPU.

Configuration with best performance on CPU fails on the GPU (unsupported work-group size).

⇒ Tuning-Parameter values have to be chosen specifically for each device!

Weaknesses of State-of-the-Art Approaches

Domain-specific approaches:

- ATLAS (linear algebra), PATUS (stencil), MILEPOST (compiler optimizations) → cannot be used for applications from other domains.

Generic approaches:

- <u>OpenTuner:</u> No support for auto-tuning parameters with interdependencies (e.g., a parameter has to divide another parameter).

- <u>CLTune:</u> Allows parameters with interdependencies but:

    1. restricted to: 1) only OpenCL, 2) only auto-tuning for runtime performance;

    2. only suitable for auto-tuning small parameter ranges usually not covering parameters' entire range → search space generation is time intensive.

The Auto-Tuning Framework (ATF)

ATF combines the following advantages over state-of-the-art auto-tuning approaches:

1. ATF is **generic** regarding:
    - programming language;
    - application domain;
    - tuning objective;
    - search technique.
2. ATF allows **dependencies between tuning parameters**.
3. ATF allows **significantly larger tuning parameter ranges**.
4. ATF is **arguably simpler** to use.

Illustration of ATF

- Illustration of ATF by a simple example: auto-tuning the OpenCL SAXPY kernel.
- For high performance, SAXPY has to be tuned specifically for a fixed user-defined input size N.
- The ATF user has to implement a C++ program using ATF's C++ API and perform the following three steps:

1. Step: Define the search space

- ATF search spaces are defined using tuning parameters, here:
  - WPT: a size_t parameter in [1,N] that divides N
  - LS: a size_t parameter in [1,N] that divides N/WPT

```
int main()
{
  std::string saxpy_kernel = /* SAXPY kernel's code as string */;
  int         N            = /* fixed user-defined input size */;

  auto WPT = atf::tp( "WPT",
                      atf::interval<size_t>(1,N),
                      atf::divides( N )
                    );
  auto LS  = atf::tp( "LS",
                      atf::interval<size_t>(1,N),
                      atf::divides( N/WPT )
                    );

  auto cf_saxpy = atf::cf::ocl(
                    { "NVIDIA", "Tesla K20c" },
                    saxpy_kernel,
                    inputs( atf::scalar<int>(N),   // N
                            atf::scalar<float>(),  // a
                            atf::buffer<float>(N), // x
                            atf::buffer<float>(N), // y
                          )
                    atf::glb_size( N/WPT ), atf::lcl_size( LS )
                  );

  auto best_config = atf::annealing( atf::duration<minutes>(10) )
                                   ( WPT, LS )
                                   ( cf_saxpy );
}
```

ATF program for auto-tuning SAXPY

Illustration of ATF

2. Step: Implement a cost function

- Cost function takes a configuration and yields a cost (e.g., program’s runtime).
- Here, we use ATF’s pre-implemented cost function for auto-tuning OpenCL in terms of runtime:
    - it is initialized with: i) target device, ii) kernel’s code, iii) kernels’ input arguments, iv) global/local size;
    - tuning parameters are replaced by values according to the input configuration;
    - it returns kernel’s runtime as cost.

3. Step: Start the tuning

- The tuning process is startet by choosing a search technique and pass to it:

    1) an abort condition,

    2) the tuning parameters

    3) the cost function

- The result is the best found configuration.

```
int main()
{
  std::string saxpy_kernel = /* SAXPY kernel's code as string */;
  int         N            = /* fixed user-defined input size */;

  auto WPT = atf::tp( "WPT",
                      atf::interval<size_t>(1,N),
                      atf::divides( N )
                    );
  auto LS  = atf::tp( "LS",
                      atf::interval<size_t>(1,N),
                      atf::divides( N/WPT )
                    );

  auto cf_saxpy = atf::cf::ocl(
                    { "NVIDIA", "Tesla K20c" },
                    saxpy_kernel,
                    inputs( atf::scalar<int>(N),    // N
                            atf::scalar<float>(),   // a
                            atf::buffer<float>(N),  // x
                            atf::buffer<float>(N),  // y
                          )
                    atf::glb_size( N/WPT ), atf::lcl_size( LS )
                  );

  auto best_config = atf::annealing( atf::duration<minutes>(10) )
                                   ( WPT, LS )
                                   ( cf_saxpy );
}
```

ATF program for auto-tuning SAXPY

Detailed Discussion of ATF

<u>1. Step:</u> Define the search space

General form of a tuning parameter:

```
atf::tp( /* name       */,
         /* range      */,
         /* constraint */
       );
```

Used to access parameter's value in a configuration , e.g., best_config[“LS”].

Boolean functions to filter the parameter's range; may contain tuning parameters, e.g.:
atf::divides(N/WPT)
(→ [&](auto LS){ N/WPT % LS == 0})

Can be either an:

1) <u>atf</u>::interval<T>(begin, end, step_size=1, generator=id), where generator:T->U

2) atf::set(val_1, … , val_n) or {val_1, … , val_n}

Detailed Discussion of ATF

2. Step: Implement a cost function

General form of a cost function:

```
T cost_function( atf::configuration config )
{
  // ...
}
```

- Input: a configuration
- Output: Element of type T (e.g., size_t) for which < is defined
- Output is interpreted as cost to minimize (e.g., program's runtime).
- Multi-Objective Tuning: e.g., auto-tune for runtime and then energy consumption → T=std::vector with < as lexicographical order
- ATF provides three pre-implemented cost functions, for:
  1. OpenCL,
  2. CUDA,
  3. arbitrary Programming languages that are not OpenCL or CUDA.

```
atf::cf::ocl(
  {“NVIDIA”, ”Tesla K20c”},
  saxpy,
  inputs( atf::scalar<int>(N),    // N
          atf::scalar<float>(),   // a
          atf::buffer<float>(N),  // x
          atf::buffer<float>(N),  // y
        )
  atf::glb_size( N/WPT ),
  atf::lcl_size( LS )
);
```

OpenCL

```
atf::cf::cuda(
  {”Tesla K20c”},
  saxpy_cuda,
  inputs( atf::scalar<int>(N),    // N
          atf::scalar<float>(),   // a
          atf::buffer<float>(N),  // x
          atf::buffer<float>(N),  // y
        )
  atf::grd_size( (N/WPT)/LS ),
  atf::blk_size( LS )
);
```

CUDA

```
atf::cf::gcf(
  /* path to source file    */
  /* path to compile script */
  /* path to run script     */
  /* path to log file       */
);
```

Generic Cost Function

Detailed Discussion of ATF

3. Step: Start the tuning

General schema to start the tuning:

```
atf::/* search technique */( /* abort condition   */ )
                           ( /* tuning parameters */ )
                           ( /* cost function     */ );
```

ATF provides three pre-implemented search techniques:

1. Exhaustive: finds provably best configuration, but probably at the cost of a long search time if the search space is large;
2. Simulated Annealing: effective for auto-tuning OpenCL/CUDA if search spaces are to large to be explored exhaustively;
3. OpenTuner: combines automatically various search techniques to yield a good tuning result on average for arbitrary applications.

ATF provides various abort conditions, e.g.:

- duration<D>(t): stops tuning after time interval t; D is an std::chrono::duration (seconds, minutes, etc.)
- cost(c): stops tuning when a configuration with a cost lower or equal than c has been found;
- speedup<D>(s,t): stops tuning when within last time interval t cost could not be lowered by a factor >=s;
- …

Comparison: ATF vs. CLTune

- <u>We show:</u> even though ATF is a generic approach, it works better for OpenCL than CLTune which is specifically designed for OpenCL.
- We use the example of auto-tuning SAXPY.

```
int main()
{
  const std::string saxpy = /* path to kernel of Listing 1   */;
  const size_t      N     = /* fixed user-defined input size */;

  cltune::Tuner tuner(1,0);
  auto id = tuner.AddKernel(saxpy, "saxpy", {N}, {1} );

  float a;
  auto  x = std::vector<float>(N);
  auto  y = std::vector<float>(N);

  const auto random_seed =
    std::chrono::system_clock::now().time_since_epoch().count();
  std::default_random_engine
    generator( static_cast<unsigned int>(random_seed) );
  std::uniform_real_distribution<float> distribution(-2.0f,2.0f);

  a = distribution(generator);
  for (auto &item: x) { item = distribution(generator); }
  for (auto &item: y) { item = distribution(generator); }

  tuner.AddArgumentScalar( N );
  tuner.AddArgumentScalar( a );
  tuner.AddArgumentInput( x );
  tuner.AddArgumentOutput( y );

  auto range = std::vector<size_t>( N );
  for( size_t i = 0; i < N ; ++i )
    range[ i ] = i;
  tuner.AddParameter( id, "LS" , range );
  tuner.AddParameter( id, "WPT", range );

  auto DividesN = []( std::vector<size_t> v )
                  {
                    return  N % v[0] == 0;
                  };
  auto DividesNDivWPT = []( std::vector<size_t> v )
                        {
                          return ( N / v[0] ) % v[1] == 0;
                        };

  tuner.AddConstraint( id, DividesN      , {"WPT"}        );
  tuner.AddConstraint( id, DividesNDivWPT, {"WPT", "LS"} );

  tuner.DivGlobalSize(id, {"WPT" } );
  tuner.MulLocalSize(id, {"LS"} );

  tuner.UseAnnealing( 1.0f/2048.0f , 4.0 );
  tuner.Tune();
  const auto parameters = tuner.GetBestResult();
}
```

Comparison: ATF vs. CLTune

```
int main()
{
  std::string saxpy_kernel = /* path to kernel of Listing 1   */;
  int          N             = /* fixed user-defined input size */;

  auto WPT = atf::tp( "WPT",
                      atf::interval<size_t>(1,N),
                      atf::divides( N )
                    );
  auto LS  = atf::tp( "LS",
                      atf::interval<size_t>(1,N),
                      atf::divides( N/WPT )
                    );

  auto cf_saxpy = atf::cf::ocl(
                    { "NVIDIA", "Tesla K20c" },
                    saxpy_kernel,
                    inputs( atf::scalar<int>(N),   // N
                            atf::scalar<float>(),  // a
                            atf::buffer<float>(N), // x
                            atf::buffer<float>(N), // y
                          )
                    atf::glb_size( N/WPT ), atf::lcl_size( LS )
                  );

  auto best_config = atf::annealing( atf::duration<minutes>(10) )
                                   ( WPT, LS )
                                   ( cf_saxpy );
}
```

```
int main()
{
  const std::string saxpy = /* path to kernel of Listing 1   */;
  const size_t      N     = /* fixed user-defined input size */;

  cltune::Tuner tuner(1,0);
  auto id = tuner.AddKernel(saxpy, "saxpy", {N}, {1} );

  float a;
  auto  x = std::vector<float>(N);
  auto  y = std::vector<float>(N);

  const auto random_seed =
    std::chrono::system_clock::now().time_since_epoch().count();
  std::default_random_engine
    generator( static_cast<unsigned int>(random_seed) );
  std::uniform_real_distribution<float> distribution(-2.0f,2.0f);

  a = distribution(generator);
  for (auto &item: x) { item = distribution(generator); }
  for (auto &item: y) { item = distribution(generator); }

  tuner.AddArgumentScalar( N );
  tuner.AddArgumentScalar( a );
  tuner.AddArgumentInput( x );
  tuner.AddArgumentOutput( y );

  auto range = std::vector<size_t>( N );
  for( size_t i = 0; i < N ; ++i )
    range[ i ] = i;
  tuner.AddParameter( id, "LS" , range );
  tuner.AddParameter( id, "WPT", range );

  auto DividesN = []( std::vector<size_t> v )
                  {
                    return  N % v[0] == 0;
                  };
  auto DividesNDivWPT = []( std::vector<size_t> v )
                        {
                          return ( N / v[0] ) % v[1] == 0;
                        };

  tuner.AddConstraint( id, DividesN      , {"WPT"}        );
  tuner.AddConstraint( id, DividesNDivWPT, {"WPT", "LS"} );

  tuner.DivGlobalSize(id, {"WPT" } );
  tuner.MulLocalSize(id, {"LS"} );

  tuner.UseAnnealing( 1.0f/2048.0f , 4.0 );
  tuner.Tune();
  const auto parameters = tuner.GetBestResult();
}
```

Comparison: ATF vs. CLTune

```
int main()
{
  std::string saxpy_kernel = /* path to kernel of Listing 1   */;
  int         N            = /* fixed user-defined input size */;

  auto WPT = atf::tp( "WPT",
                      atf::interval<size_t>(1,N),
                      atf::divides( N )
                    );
  auto LS  = atf::tp( "LS",
                      atf::interval<size_t>(1,N),
                      atf::divides( N/WPT )
                    );

  auto cf_saxpy = atf::cf::ocl(
                    { "NVIDIA", "Tesla K20c" },
                    saxpy_kernel,
                    inputs( atf::scalar<int>(N),   // N
                            atf::scalar<float>(),  // a
                            atf::buffer<float>(N), // x
                            atf::buffer<float>(N), // y
                          )
                    atf::glb_size( N/WPT ), atf::lcl_size( LS )
                  );

  auto best_config = atf::annealing( atf::duration<minutes>(10) )
                                   ( WPT, LS )
                                   ( cf_saxpy );
}
```

ATF allows easier expressing parameter dependencies

```
int main()
{
  const std::string saxpy = /* path to kernel of Listing 1   */;
  const size_t      N     = /* fixed user-defined input size */;

  cltune::Tuner tuner(1,0);
  auto id = tuner.AddKernel(saxpy, "saxpy", {N}, {1} );

  float a;
  auto  x = std::vector<float>(N);
  auto  y = std::vector<float>(N);

  const auto random_seed =
    std::chrono::system_clock::now().time_since_epoch().count();
  std::default_random_engine
    generator( static_cast<unsigned int>(random_seed) );
  std::uniform_real_distribution<float> distribution(-2.0f,2.0f);

  a = distribution(generator);
  for (auto &item: x) { item = distribution(generator); }
  for (auto &item: y) { item = distribution(generator); }

  tuner.AddArgumentScalar( N );
  tuner.AddArgumentScalar( a );
  tuner.AddArgumentInput( x );
  tuner.AddArgumentOutput( y );

  auto range = std::vector<size_t>( N );
  for( size_t i = 0; i < N ; ++i )
    range[ i ] = i;
  tuner.AddParameter( id, "LS" , range );
  tuner.AddParameter( id, "WPT", range );

  auto DividesN = []( std::vector<size_t> v )
                  {
                    return  N % v[0] == 0;
                  };
  auto DividesNDivWPT = []( std::vector<size_t> v )
                        {
                          return ( N / v[0] ) % v[1] == 0;
                        };

  tuner.AddConstraint( id, DividesN      , {"WPT"}       );
  tuner.AddConstraint( id, DividesNDivWPT, {"WPT", "LS"} );

  tuner.DivGlobalSize(id, {"WPT" } );
  tuner.MulLocalSize(id, {"LS"} );

  tuner.UseAnnealing( 1.0f/2048.0f , 4.0 );
  tuner.Tune();
  const auto parameters = tuner.GetBestResult();
}
```

Comparison: ATF vs. CLTune

```
int main()
{
  std::string saxpy_kernel = /* path to kernel of Listing 1   */;
  int         N            = /* fixed user-defined input size */;

  auto WPT = atf::tp( "WPT",
                      atf::interval<size_t>(1,N),
                      atf::divides( N )
                    );
  auto LS  = atf::tp( "LS",
                      atf::interval<size_t>(1,N),
                      atf::divides( N/WPT )
                    );

  auto cf_saxpy = atf::cf::ocl(
                    { "NVIDIA", "Tesla K20c" },
                    saxpy_kernel,
                    inputs( atf::scalar<int>(N),   // N
                            atf::scalar<float>(),  // a
                            atf::buffer<float>(N), // x
                            atf::buffer<float>(N), // y
                          )
                    atf::glb_size( N/WPT ), atf::lcl_size( LS )
                  );

  auto best_config = atf::annealing( atf::duration<minutes>(10) )
                                   ( WPT, LS )
                                   ( cf_saxpy );
}
```

```
int main()
{
  const std::string saxpy = /* path to kernel of Listing 1   */;
  const size_t      N     = /* fixed user-defined input size */;

  cltune::Tuner tuner(1,0);
  auto id = tuner.AddKernel(saxpy, "saxpy", {N}, {1} );

  float a;
  auto  x = std::vector<float>(N);
  auto  y = std::vector<float>(N);

  const auto random_seed =
    std::chrono::system_clock::now().time_since_epoch().count();
  std::default_random_engine
    generator( static_cast<unsigned int>(random_seed) );
  std::uniform_real_distribution<float> distribution(-2.0f,2.0f);

  a = distribution(generator);
  for (auto &item: x) { item = distribution(generator); }
  for (auto &item: y) { item = distribution(generator); }

  tuner.AddArgumentScalar( N );
  tuner.AddArgumentScalar( a );
  tuner.AddArgumentInput( x );
  tuner.AddArgumentOutput( y );

  auto range = std::vector<size_t>( N );
  for( size_t i = 0; i < N ; ++i )
    range[ i ] = i;
  tuner.AddParameter( id, "LS" , range );
  tuner.AddParameter( id, "WPT", range );

  auto DividesN = []( std::vector<size_t> v )
                {
                  return  N % v[0] == 0;
                };
  auto DividesNDivWPT = []( std::vector<size_t> v )
                      {
                        return ( N / v[0] ) % v[1] == 0;
                      };

  tuner.AddConstraint( id, DividesN      , {"WPT"}       );
  tuner.AddConstraint( id, DividesNDivWPT, {"WPT", "LS"} );

  tuner.DivGlobalSize(id, {"WPT" } );
  tuner.MulLocalSize(id, {"LS"} );

  tuner.UseAnnealing( 1.0f/2048.0f , 4.0 );
  tuner.Tune();
  const auto parameters = tuner.GetBestResult();
}
```

ATF allows easier setting the global/local size

Comparison: ATF vs. CLTune

```
int main()
{
  std::string saxpy_kernel = /* path to kernel of Listing 1   */;
  int         N            = /* fixed user-defined input size */;

  auto WPT = atf::tp( "WPT",
                      atf::interval<size_t>(1,N),
                      atf::divides( N )
                    );
  auto LS  = atf::tp( "LS",
                      atf::interval<size_t>(1,N),
                      atf::divides( N/WPT )
                    );

  auto cf_saxpy = atf::cf::ocl(
                    { "NVIDIA", "Tesla K20c" },
                    saxpy_kernel,
                    inputs( atf::scalar<int>(N),    // N
                            atf::scalar<float>(),   // a
                            atf::buffer<float>(N),  // x
                            atf::buffer<float>(N),  // y
                          )
                    atf::glb_size( N/WPT ), atf::lcl_size( LS )
                  );

  auto best_config = atf::annealing( atf::duration<minutes>(10) )
                                   ( WPT, LS )
                                   ( cf_saxpy );
}
```

ATF allows a broader range of global/local sizes

```
int main()
{
  const std::string saxpy = /* path to kernel of Listing 1   */;
  const size_t      N     = /* fixed user-defined input size */;

  cltune::Tuner tuner(1,0);
  auto id = tuner.AddKernel(saxpy, "saxpy", {N}, {1} );

  float a;
  auto  x = std::vector<float>(N);
  auto  y = std::vector<float>(N);

  const auto random_seed =
    std::chrono::system_clock::now().time_since_epoch().count();
  std::default_random_engine
    generator( static_cast<unsigned int>(random_seed) );
  std::uniform_real_distribution<float> distribution(-2.0f,2.0f);

  a = distribution(generator);
  for (auto &item: x) { item = distribution(generator); }
  for (auto &item: y) { item = distribution(generator); }

  tuner.AddArgumentScalar( N );
  tuner.AddArgumentScalar( a );
  tuner.AddArgumentInput( x );
  tuner.AddArgumentOutput( y );

  auto range = std::vector<size_t>( N );
  for( size_t i = 0; i < N ; ++i )
    range[ i ] = i;
  tuner.AddParameter( id, "LS" , range );
  tuner.AddParameter( id, "WPT", range );

  auto DividesN = []( std::vector<size_t> v )
                  {
                    return  N % v[0] == 0;
                  };
  auto DividesNDivWPT = []( std::vector<size_t> v )
                        {
                          return ( N / v[0] ) % v[1] == 0;
                        };

  tuner.AddConstraint( id, DividesN      , {"WPT"}       );
  tuner.AddConstraint( id, DividesNDivWPT, {"WPT", "LS"} );

  tuner.DivGlobalSize(id, {"WPT" } );
  tuner.MulLocalSize(id, {"LS"} );

  tuner.UseAnnealing( 1.0f/2048.0f , 4.0 );
  tuner.Tune();
  const auto parameters = tuner.GetBestResult();
}
```

Comparison: ATF vs. CLTune

```
int main()
{
  std::string saxpy_kernel = /* path to kernel of Listing 1   */;
  int         N            = /* fixed user-defined input size */;

  auto WPT = atf::tp( "WPT",
                      atf::interval<size_t>(1,N),
                      atf::divides( N )
                    );
  auto LS  = atf::tp( "LS",
                      atf::interval<size_t>(1,N),
                      atf::divides( N/WPT )
                    );

  auto cf_saxpy = atf::cf::ocl(
                    { "NVIDIA", "Tesla K20c" },
                    saxpy_kernel,
                    inputs( atf::scalar<int>(N),   // N
                            atf::scalar<float>(),  // a
                            atf::buffer<float>(N), // x
                            atf::buffer<float>(N), // y
                          )
                    atf::glb_size( N/WPT ), atf::lcl_size( LS )
                  );

  auto best_config = atf::annealing( atf::duration<minutes>(10) )
                                   ( WPT, LS )
                                   ( cf_saxpy );
}
```

ATF allows easier generating random input data

```
int main()
{
  const std::string saxpy = /* path to kernel of Listing 1   */;
  const size_t      N     = /* fixed user-defined input size */;

  cltune::Tuner tuner(1,0);
  auto id = tuner.AddKernel(saxpy, "saxpy", {N}, {1} );

  float a;
  auto  x = std::vector<float>(N);
  auto  y = std::vector<float>(N);

  const auto random_seed =
    std::chrono::system_clock::now().time_since_epoch().count();
  std::default_random_engine
    generator( static_cast<unsigned int>(random_seed) );
  std::uniform_real_distribution<float> distribution(-2.0f,2.0f);

  a = distribution(generator);
  for (auto &item: x) { item = distribution(generator); }
  for (auto &item: y) { item = distribution(generator); }

  tuner.AddArgumentScalar( N );
  tuner.AddArgumentScalar( a );
  tuner.AddArgumentInput( x );
  tuner.AddArgumentOutput( y );

  auto range = std::vector<size_t>( N );
  for( size_t i = 0; i < N ; ++i )
    range[ i ] = i;
  tuner.AddParameter( id, "LS" , range );
  tuner.AddParameter( id, "WPT", range );

  auto DividesN = []( std::vector<size_t> v )
                  {
                    return  N % v[0] == 0;
                  };
  auto DividesNDivWPT = []( std::vector<size_t> v )
                        {
                          return ( N / v[0] ) % v[1] == 0;
                        };

  tuner.AddConstraint( id, DividesN      , {"WPT"}       );
  tuner.AddConstraint( id, DividesNDivWPT, {"WPT", "LS"} );

  tuner.DivGlobalSize(id, {"WPT" } );
  tuner.MulLocalSize(id, {"LS"} );

  tuner.UseAnnealing( 1.0f/2048.0f , 4.0 );
  tuner.Tune();
  const auto parameters = tuner.GetBestResult();
}
```

Comparison: ATF vs. CLTune

```
int main()
{
  std::string saxpy_kernel = /* path to kernel of Listing 1   */;
  int         N            = /* fixed user-defined input size */;

  auto WPT = atf::tp( "WPT",
                      atf::interval<size_t>(1,N),
                      atf::divides( N )
                    );
  auto LS  = atf::tp( "LS",
                      atf::interval<size_t>(1,N),
                      atf::divides( N/WPT )
                    );

  auto cf_saxpy = atf::cf::ocl(
                    { "NVIDIA", "Tesla K20c" },
                    saxpy_kernel,
                    inputs( atf::scalar<int>(N),   // N
                            atf::scalar<float>(),  // a
                            atf::buffer<float>(N), // x
                            atf::buffer<float>(N), // y
                          )
                    atf::glb_size( N/WPT ), atf::lcl_size( LS )
                  );

  auto best_config = atf::annealing( atf::duration<minutes>(10) )
                                   ( WPT, LS )
                                   ( cf_saxpy );
}
```

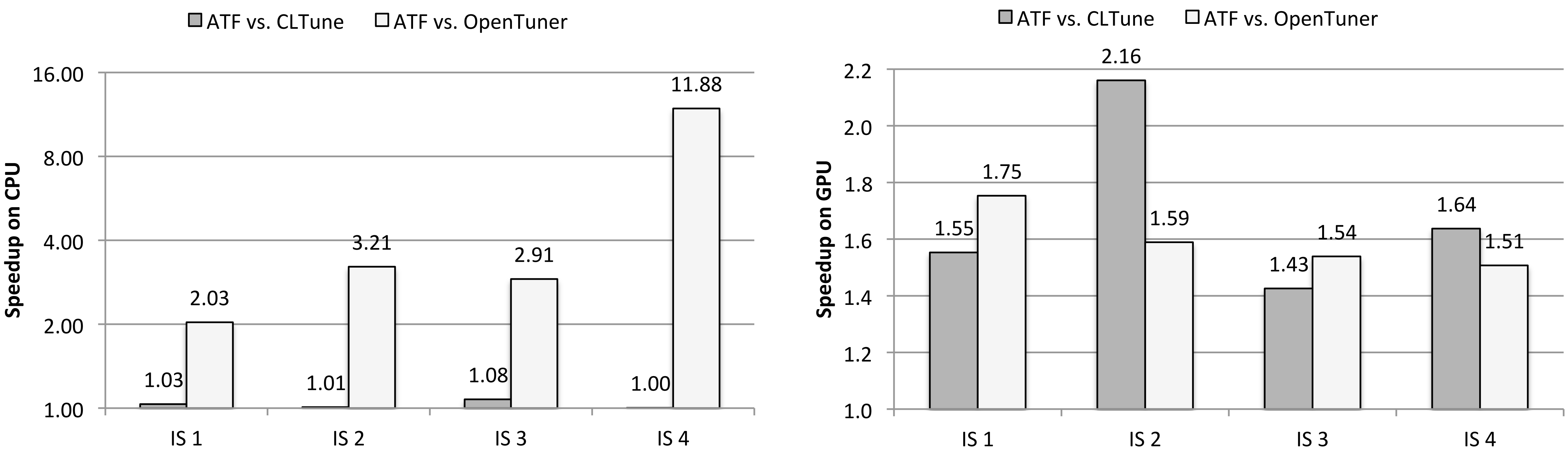
ATF supports more abort conditions

```
int main()
{
  const std::string saxpy = /* path to kernel of Listing 1   */;
  const size_t      N     = /* fixed user-defined input size */;

  cltune::Tuner tuner(1,0);
  auto id = tuner.AddKernel(saxpy, "saxpy", {N}, {1} );

  float a;
  auto  x = std::vector<float>(N);
  auto  y = std::vector<float>(N);

  const auto random_seed =
    std::chrono::system_clock::now().time_since_epoch().count();
  std::default_random_engine
    generator( static_cast<unsigned int>(random_seed) );
  std::uniform_real_distribution<float> distribution(-2.0f,2.0f);

  a = distribution(generator);
  for (auto &item: x) { item = distribution(generator); }
  for (auto &item: y) { item = distribution(generator); }

  tuner.AddArgumentScalar( N );
  tuner.AddArgumentScalar( a );
  tuner.AddArgumentInput( x );
  tuner.AddArgumentOutput( y );

  auto range = std::vector<size_t>( N );
  for( size_t i = 0; i < N ; ++i )
    range[ i ] = i;
  tuner.AddParameter( id, "LS" , range );
  tuner.AddParameter( id, "WPT", range );

  auto DividesN = []( std::vector<size_t> v )
                  {
                    return  N % v[0] == 0;
                  };
  auto DividesNDivWPT = []( std::vector<size_t> v )
                        {
                          return ( N / v[0] ) % v[1] == 0;
                        };

  tuner.AddConstraint( id, DividesN      , {"WPT"}       );
  tuner.AddConstraint( id, DividesNDivWPT, {"WPT", "LS"} );

  tuner.DivGlobalSize(id, {"WPT" } );
  tuner.MulLocalSize(id, {"LS"} );

  tuner.UseAnnealing( 1.0f/2048.0f , 4.0 );
  tuner.Tune();
  const auto parameters = tuner.GetBestResult();
}
```

Experimental Results

- We demonstrate: ATF provides better tuning results for GEMM (GEneral Matrix Multiplication) than OpenTuner and CLTune.
- As concrete GEMM implementation, we use the OpenCL kernel XgemmDirect which is part of the CLBlast library that uses CLTune for auto-tuning.
- XgemmDirect is used for accelerating important applications, e.g., the state-of-the-art deep learning framework Caffe [Jia et al, 2014].
- XgemmDirect has 10 tuning parameter, e.g., the tile size WGD and the loop unrolling factor KWID.
- The tuning parameters have various dependencies (16 in total), e.g, KWID has to divide WGD.
- We study four pairs of matrix input sizes (IS) with significance in 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: 50×500 and 500×64

Experimental Results

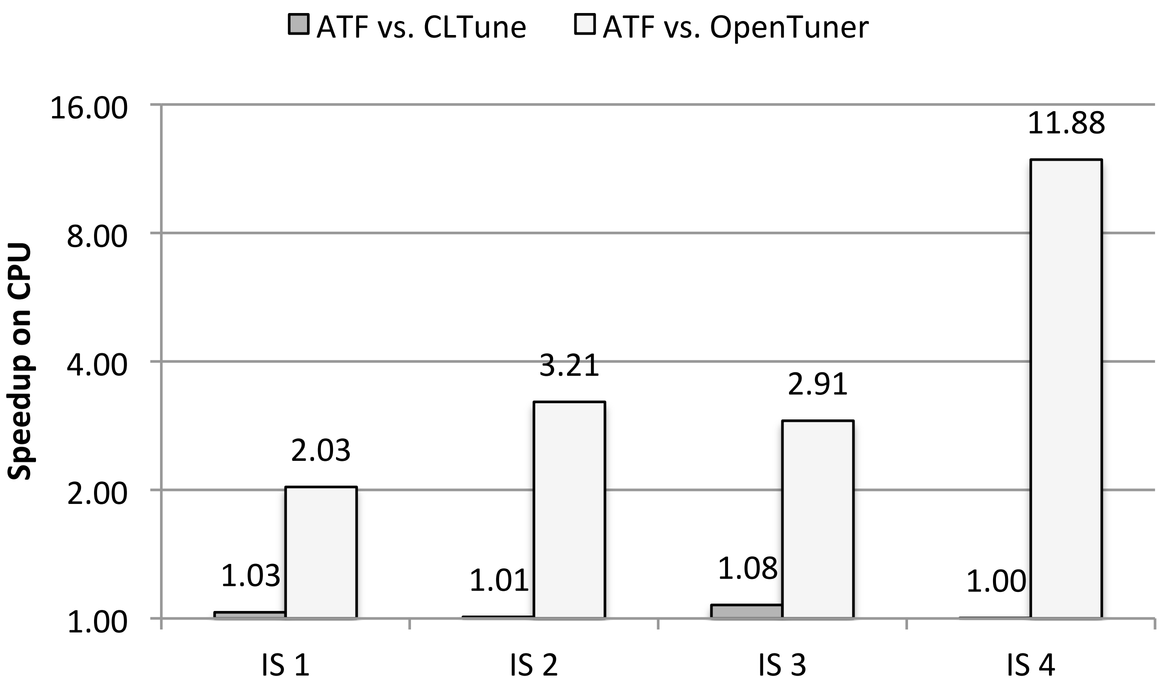


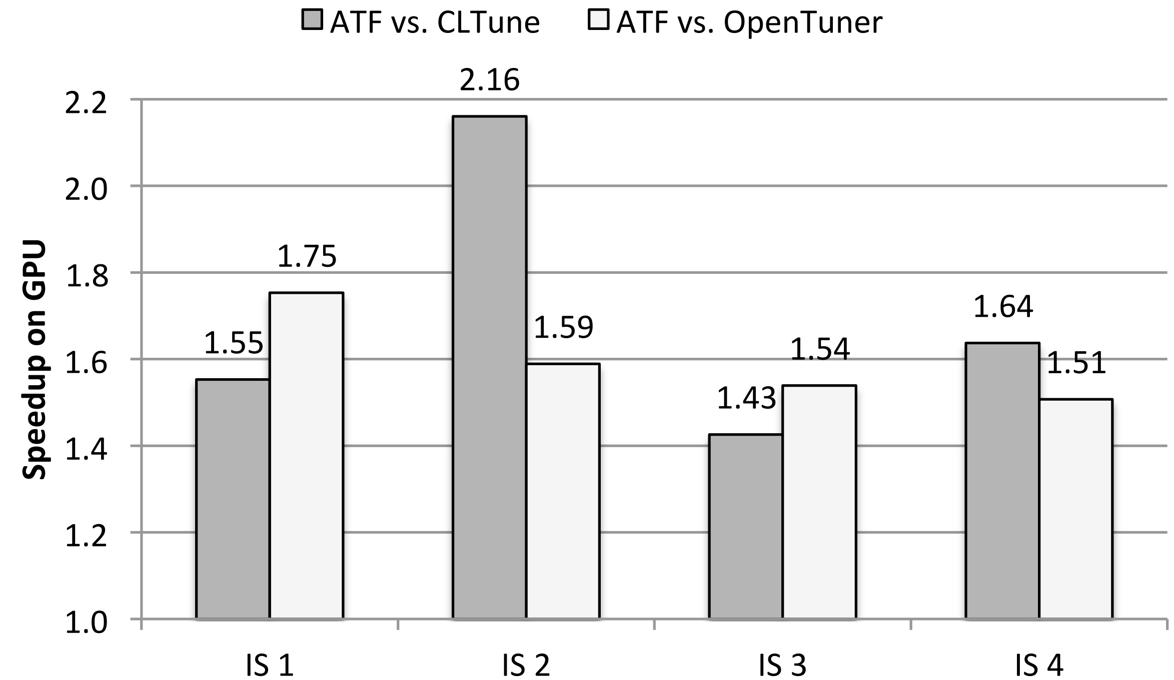
Speedup of XgemmDirect auto-tuned by ATF compared to XgemmDirect auto-tuned by CLTune and OpenTuner.

ATF vs CLTune:

- Speedups of up to **1.08x** on CPU and **2.16x** on GPU:
  - CLBlast uses artificially limited tuning parameter ranges to shorten time-intensive search space generation.
  - Limitations cause search space to be empty for result matrices MxN where M or N are not divisible by 8: the tile size (WGD) is constrained to divide M and N, but is limited to { 8, 16, 32 } (instead of {1,…,min(M,N)}).
  - CLBlast has to rely on device-optimized parameter values optimized for the average matrix size 256x256.
- Removing the artificial limitations causes significant time for search space generation: for small 32x32 matrices, we aborted CLTune after 3 hours; ATF requires less than 2 seconds ➝ ATF filters parameter ranges while CLTune filters the (large) search space.

Experimental Results





Speedup of XgemmDirect auto-tuned by ATF compared to XgemmDirect auto-tuned by CLTune and OpenTuner.

<u>ATF vs OpenTuner:</u>

- Speedups of up to **11.88x** on CPU and **1.75x** on GPU:
    - OpenTuner uses unconstrained search space and is not able to find valid configurations.
    - Search space size for IS 4: 10^{13} unconstrained (OpenTuner) vs. and 10^{5} constrained (ATF).
    - XgemmDirect has to rely on its default tuning parameter values → chosen to yield a good performance on average on various devices and for different input sizes.

Summary

- Auto-tuning simplifies optimizing programs by automatically choosing suitable values of tuning parameters.
- ATF combines four advantages over the state-of-the-art auto-tuning approaches:

  1. ATF is **generic** regarding the programming language, application domain, tuning objective, and search technique.

  2. ATF allows **dependencies between tuning parameters**, thus enabling to auto-tune a broader class of applications.

  3. ATF allows **significantly larger tuning parameter ranges** thus does not require artificially limiting its tuning parameters' ranges.

  4. ATF is **arguably simpler** to use, thus making auto-tuning appealing to common application developers.

- ATF significantly accelerates the performance of GEMM.

Appendix

Search space generation:

ATF

```
for( val_1 : tp_1.range )
  if( constraint_1( val_1 ) == true )
   .
    .
     .
      for( val_n : tp_n.range )
        if( constraint_n( val_n ) == true )
        {
          search_space.add( val_1, ... , val_n );
        }
```

CLTune

```
for( val_1 : tp_1.range )
 .
  .
   .
    for( val_n : tp_n.range )
    {
      for( c : constraints )
        if( c( val_1, ... , val_n )
          search_space.add( val_1, ... , val_n );
    }
```

Appendix

Interface search_technique:

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