Autotuning Polybench Benchmarks with Clang Loop Optimization Pragmas

Xingfu Wu, Michael Kruse, Prasanna Balaprakash, Brice Videau, Hal Finkel, and Paul Hovland
Argonne National Laboratory

Technical Report
April 16, 2020

This report focuses on autotuning polybench benchmarks with Clang loop optimization pragmas. We use the Clang loop tiling, interchange and/or jam pragmas as examples to illustrate the integration process about autotuning the pragma parameters to achieve the optimal performance. We summarize the general autotuning framework into the following steps shown in Figure 1:

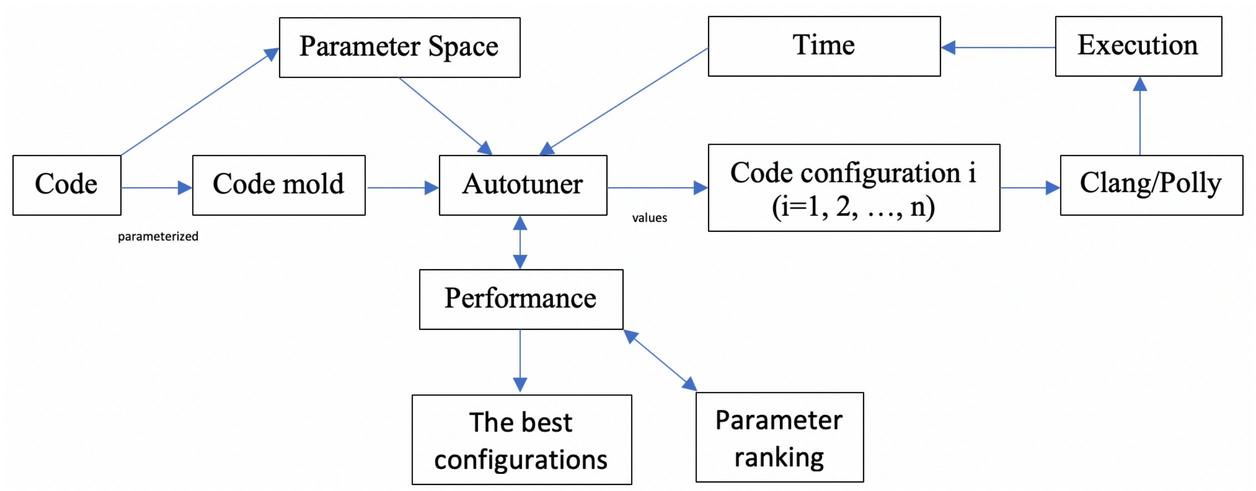

Figure 1. A Framework for Autotuning Clang Loop Pragmas

1) Analyze an application code to identify the important parameters which we try to focus on
2) Replace these parameters with the symbols like #P1, #P2, …, #Pn in the top-down order to generate another code with these symbols as a mold code
3) Define the value ranges of these symbols for input_space using autotuner (problem.py)
4) Autotuner assigns the values in the allowed ranges (using random forest as default) and replace these symbols in the mold code with them to generate a new code using the function plopper (plopper.py)
5) plopper compiles the code and executes it several times to get the average execution time (using a Perl script exe.pl; default: 1)
6) Autotuner writes the average time and the elapsed time with the parameters' values to the two output files: results.csv and results.json (both files are similar with different formats) The file results.csv looks like

```
p0,p1,p2,p3,p4,p5,objective,elapsed_sec
#pragma clang loop(j2) pack array(A) allocate(malloc), , ,32,100,128,0.85,4.602376937866211
 , , ,100,100,100,0.811,6.5058159828186035
 , , ,100,100,256,0.816,8.502590894699097
…
```

7) Repeat steps 4, 5 and 6 until it reaches the maximum number of evaluation times set (using the option --max-evals=maximum number; default: 100)
8) Process results.csv to find the smallest execution time and output the optimal configurations for the time (findMin.py)
9) Identify the most important features which impact the performance (proposed for further work) for the search improvement

Outline:

1. **Installation requirements**
2. **Survey of polybench benchmarks**
3. **Using autotune and ytopt to work on the benchmarks with Clang pragmas**
4. **Finding the optimal configurations for the smallest execution time**
5. **Identifying the important features which impact the performance**

1. Installation requirements

This autotuning framework requires the following components: SOLLVE Clang with polly and pragma-clang-loop, configspace, scikit-optimize, autotune and ytopt. See the following installation instructions for the details.

1.1 Installing the latest version of SOLLVE

Use SOLLVE official site: https://github.com/SOLLVE/llvm-project.git to install Clang with polly, and use *git checkout origin/pragma-clang-loop* to install the pragma-clang-loop.

1.2. Installing the latest version of autotune and ytopt

For the autotune and ytopt using ConfigSpace, the installation is as follows:

1) Create a conda env ytune
   conda create -n ytune python=3.7

2) pip install ConfigSpace

3) git clone https://github.com/pbalapra/scikit-optimize.git
   cd scikit-optimize
   pip install -e .

4) git clone https://github.com/ytopt-team/autotune.git
   cd autotune/
   git checkout version1
   pip install -e .

5) git clone https://github.com/ytopt-team/ytopt.git
   cd ytopt/
   pip install -e .

Under the current directory, test the benchmark example:

```
python -m ytopt.search.ambs --evaluator ray --problem
ytopt.benchmark.loopopt.problem.Problem
```

If this works, this environment for the autotuning framework is ready to use.

2. Survey of polybench benchmarks

PolyBench 4.2 (https://sourceforge.net/projects/polybench/) a benchmark suite of 30 numerical computations with static control flow, extracted from operations in various application domains (linear algebra computations (19), image processing (3), physics simulation (6), and data mining (2)). PolyBench has the following features:

- A single file, tunable at compile-time, used for the kernel instrumentation. It performs extra operations such as cache flushing before the kernel execution, and can set real-time scheduling to prevent OS interference.
- Non-null data initialization, and live-out data dump.
- Syntactic constructs to prevent any dead code elimination on the kernel.
- Parametric loop bounds in the kernels, for general-purpose implementation.
- Clear kernel marking, using pragma-based delimiters.

In this report, we use two benchmarks gemm and syr2k from linear algebra computations to illustrate how the autotuning framework works.

gemm entails the matrix multiplication C = alpha*A*B + beta*C; syr2k entails the matrix multiplication C = A*alpha*B+B*alpha*A+belta*C.

The package tile-interchange includes the following directories:

gemm	(gemm benchmark)
syr2k	(syr2k benchmark)
plopper	(code generation and compiling for both gemm and syr2k)

Before autotuning a benchmark, please check the compiler clang to see if these benchmarks can be compiled successfully using gemm.bat and syr2k.bat under the corresponding directories.

To autotune a benchmark, under the conda environment, use run.bat in the benchmark directory to test the autotuning framework. See the description of the framework for the details.

The run.bat looks like:

```
python -m ytopt.search.ambs --evaluator ray --problem problem.Problem --max-evals=200 --
learner RF  (Note: autotuning the benchmark)
```

python findMin.py (Note: find the configuration for optimal performance from results.csv)

3. Using autotune and ytopt to work on the benchmarks with Clang pragmas

We use the Clang loop tiling, interchange and/or jam pragmas as examples to illustrate the integration process about autotuning the pragma parameters to achieve the optimal performance. We worked on the polybench benchmarks for case studies.

3.1. Case Study: Polybench gemm (C = alpha*A*B + beta*C) with multiple loop transformations (loop tiling, interchange, and pack)

We apply multiple loop transformations to the polybench gemm for our case study.

The polybench gemm provides the following large datasets:

```
#  ifdef LARGE_DATASET
#   define NI 1000
#   define NJ 1100
#   define NK 1200
#  endif

#  ifdef EXTRALARGE_DATASET
#   define NI 2000
#   define NJ 2300
#   define NK 2600
#  endif
```

We use both datasets to test the performance on my laptop (3.1GHz Quad-core intel Core i7) with 16GB memory. For the original code, the following table show some results:

Compiler and options	Large	ExtraLarge
Gcc 7.2 -O3	0.760 s	7.676 s
Clang 10.0 -O3	0.729 s	6.651 s
Clang 10.0 -O3 with polly -std=c99 -fno-unroll-loops -O3 -mllvm -polly -mllvm -polly-process-unprofitable -mllvm -polly-use-llvm-names -ffast-math -march=native	0.157 s	1.065 s

For the tile, and interchange pragmas (gemm.c), we define the following parameters:

```
#P0
#P1
#P2
#pragma clang loop(i,j,k) tile sizes(#P3,#P4,#P5) floor_ids(i1,j1,k1) tile_ids(i2,j2,k2)
#pragma clang loop id(i)
 for (i = 0; i < _PB_NI; i++) {
   #pragma clang loop id(j)
   for (j = 0; j < _PB_NK; j++){
    #pragma clang loop id(k)
    for (k = 0; k < _PB_NJ; k++) {
         C[i][k] += alpha * A[i][j] * B[j][k];
    }
  }

}
```

For this case, we have to make sure that there is no dependence among P0, P1, P2, P3, P4 and P5 because we define them as categorical type with the choices of with them or without them.

Based on the defined parameters, we have the following parameter space (using ConfigSpace):

```
cs = CS.ConfigurationSpace(seed=1234)
p0= CSH.CategoricalHyperparameter(name='p0', choices=["#pragma clang loop(j2) pack array(A)
allocate(malloc)", " "], default_value=' ')
p1= CSH.CategoricalHyperparameter(name='p1', choices=["#pragma clang loop(i1) pack array(B)
allocate(malloc)", " "], default_value=' ')
p2= CSH.CategoricalHyperparameter(name='p2', choices=["#pragma clang loop(i1,j1,k1,i2,j2) interchange
permutation(j1,k1,i1,j2,i2)", " "], default_value=' ')
p3= CSH.OrdinalHyperparameter(name='p3', sequence=['4','8','16','20','32','50','64','80','96','100','128'],
default_value='96')
p4= CSH.OrdinalHyperparameter(name='p4', sequence=['4','8','16','20','32','50','64','80','100','128','2048'],
default_value='2048')
p5= CSH.OrdinalHyperparameter(name='p5', sequence=['4','8','16','20','32','50','64','80','100','128','256'],
default_value='256')

cs.add_hyperparameters([p0, p1, p2, p3, p4, p5])

#cond1 = CS.InCondition(p1, p0, ['#pragma clang loop(j2) pack array(A) allocate(malloc)'])
#cs.add_condition(cond1)

input_space = cs
```

The result output of the autotuning (results.csv) looks like:

```
p0,p1,p2,p3,p4,p5,objective,elapsed_sec
#pragma clang loop(j2) pack array(A) allocate(malloc), , ,32,100,128,0.85,4.602376937866211
 , , ,100,100,100,0.811,6.5058159828186035
 , , ,100,100,256,0.816,8.502590894699097
 , , ,100,128,50,0.817,10.510121822357178
 , , ,100,16,16,0.233,12.5169677734375
 , , ,100,128,100,0.825,14.51698613166809
 , , ,100,128,20,0.224,16.507138967514038
 , , ,128,128,80,0.821,18.506083011627197
 , , ,128,128,50,0.8,20.5277681350708
 , , ,100,16,20,0.245,22.513237953186035
 , , ,100,16,4,0.225,24.508893966674805
 , ,"#pragma clang loop(i1,j1,k1,i2,j2) interchange permutation(j1,k1,i1,j2,i2)",100,16,100,0.863,26.511728763580322
 , , ,100,16,256,0.778,28.51711893081665
 , , ,100,16,50,0.8,30.5142560005188
#pragma clang loop(j2) pack array(A) allocate(malloc), , ,100,16,16,0.236,32.511526107788086
 , ,"#pragma clang loop(i1,j1,k1,i2,j2) interchange permutation(j1,k1,i1,j2,i2)",100,16,16,1.208,35.54129505157471
 , ,"#pragma clang loop(i1,j1,k1,i2,j2) interchange permutation(j1,k1,i1,j2,i2)",50,16,16,1.296,38.56500601768494
…
```

3.2. Case Study: Polybench syr2k (C = A*alpha*B+B*alpha*A+belta*C) with multiple loop transformations (loop tiling, interchange, and pack)

We apply multiple loop transformations to the polybench syr2k for our case study.

The polybench syr2k provides the following large datasets:

```
#  ifdef LARGE_DATASET
#   define M 1000
#   define N 1200
#  endif

#  ifdef EXTRALARGE_DATASET
#   define M 2000
#   define N 2600
#  endif
```

For the tile, and interchange pragmas (syr2k.c), we define the following parameters:

```
#P0
#P1
#P2
#pragma clang loop(i,j,k) tile sizes(#P3,#P4,#P5) floor_ids(i1,j1,k1) tile_ids(i2,j2,k2)
#pragma clang loop id(i)
 for (i = 0; i < _PB_N; i++) {
   #pragma clang loop id(j)
   for (j = 0; j < _PB_M; j++) {
    #pragma clang loop id(k)
      for (k = 0; k <= i; k++)
      {
        C[i][k] += A[k][j]*alpha*B[i][j] + B[k][j]*alpha*A[i][j];
      }
   }
 }
```

For this case, we have to make sure that there is no dependence among P0, P1, P2, P3, P4 and P5 because we define them as categorical type with the choices of with them or without them.

Based on the defined parameters, we have the following parameter space (using ConfigSpace):

```
cs = CS.ConfigurationSpace(seed=1234)
p0= CSH.CategoricalHyperparameter(name='p0', choices=["#pragma clang loop(j2) pack array(A)
allocate(malloc)", " "], default_value=' ')
p1= CSH.CategoricalHyperparameter(name='p1', choices=["#pragma clang loop(i1) pack array(B)
allocate(malloc)", " "], default_value=' ')
p2= CSH.CategoricalHyperparameter(name='p2', choices=["#pragma clang loop(i1,j1,k1,i2,j2) interchange
permutation(j1,k1,i1,j2,i2)", " "], default_value=' ')
p3= CSH.OrdinalHyperparameter(name='p3', sequence=['4','8','16','20','32','50','64','80','96','100','128'],
default_value='96')
p4= CSH.OrdinalHyperparameter(name='p4', sequence=['4','8','16','20','32','50','64','80','100','128','2048'],
default_value='2048')
p5= CSH.OrdinalHyperparameter(name='p5', sequence=['4','8','16','20','32','50','64','80','100','128','256'],
default_value='256')

cs.add_hyperparameters([p0, p1, p2, p3, p4, p5])

#cond1 = CS.InCondition(p1, p0, ['#pragma clang loop(j2) pack array(A) allocate(malloc)'])
#cs.add_condition(cond1)

input_space = cs
```

The result output of the autotuning (results.csv) looks like:

```
p0,p1,p2,p3,p4,p5,objective,elapsed_sec
#pragma clang loop(j2) pack array(A) allocate(malloc), , ,32,100,128,1.087,4.4886720180511475
 , , ,100,100,100,0.739,6.387370824813843
 , , ,100,100,256,0.842,8.39160704612732
 , , ,100,128,50,0.866,10.407526016235352
 , , ,100,16,16,0.812,12.396085023880005
 , , ,100,128,100,0.749,14.402247190475464
 , , ,16,100,16,0.794,16.391496181488037
 , , ,128,100,32,0.741,18.388952016830444
 , , ,20,100,20,0.793,20.41989302635193
 , , ,128,100,16,0.775,22.396055936813354
 , , ,128,100,80,0.754,24.392914056777954
 , , ,128,100,4,0.712,26.398925065994263
 ,#pragma clang loop(i1) pack array(B) allocate(malloc),"#pragma clang loop(i1,j1,k1,i2,j2) interchange
permutation(j1,k1,i1,j2,i2)",128,100,100,0.696,28.419668912887573
 ,#pragma clang loop(i1) pack array(B) allocate(malloc), ,128,100,80,0.713,30.39665913581848
 , , ,128,100,256,0.809,32.39763402938843
 , ,"#pragma clang loop(i1,j1,k1,i2,j2) interchange permutation(j1,k1,i1,j2,i2)",4,100,100,0.801,34.39807415008545
…
```

4. Finding the optimal configurations for the smallest execution time

We use both datasets to test the performance on my laptop (3.1GHz Quad-core intel Core i7) with 16GB memory. Based on the output file results.csv, We wrote the Python script findMin.py to find the smallest execution time and output the best configurations for the time.

4.1. For the application gemm.c (C = alpha*A*B + beta*C), the summary output is:

Large dataset:
Performance summary based on 200 evaluations:
Min: 0.177 s
Max: 1.296 s
Mean: 0.325685 s
The best configurations (for the smallest time) of P0, P1, P2, P3, P4 and P5 is:

```
P0     P1     P2     P3     P4    P5      execution time         elapsed time

['''''' 100 2048 16 0.177 182.5524709224701]
```

Extralarge dataset:
Performance summary based on 200 evaluations:
Min: 1.041 s
Max: 9.557 s
Mean: 3.149525000000001 s
The best configurations (for the smallest time) of P0, P1, P2, P3, P4 and P5 is:

```
P0     P1     P2     P3     P4    P5      execution time         elapsed time

['''''' 100 128 8 1.041 72.74733471870421]
```

We compare the performance as shown in the following table:

Compiler and options	Large	ExtraLarge
Gcc 7.2 -O3	0.760 s	7.676 s
Clang 10.0 -O3	0.729 s	6.651 s
Clang 10.0 -O3 with polly -std=c99 -fno-unroll-loops -O3 -mllvm -polly -mllvm -polly-process-unprofitable -mllvm -polly-use-llvm-names -ffast-math -march=native	0.157 s	1.065 s
Clang 10.0 -O3 with polly and tiling (96, 2048, 256) and interchange -std=c99 -fno-unroll-loops -O3 -mllvm -polly -mllvm -polly-process-unprofitable -mllvm -polly-use-llvm-names -ffast-math -march=native	0.262 s	1.852 s
Clang 10.0 -O3 with polly and autotuning (100 2048 16) (100 128 8) -std=c99 -fno-unroll-loops -O3 -mllvm -polly -mllvm -polly-process-unprofitable -mllvm -polly-use-llvm-names -ffast-math -march=native	0.177 s	1.041s

4.2. For the application syr2k.c (C = A*alpha*B+B*alpha*A+belta*C), the summary output is:

Large dataset:

```
Performance summary based on 200 evaluations:
Min:  0.239 s
Max:  1.8530000000000002 s
Mean:  0.40743000000000046 s
The best configurations (for the smallest time) of P0, P1, P2, P3, P4 and P5 is:

P0     P1     P2     P3     P4    P5      execution time         elapsed time

['#pragma clang loop(j2) pack array(A) allocate(malloc)'
 '#pragma clang loop(i1) pack array(B) allocate(malloc)'
 '#pragma clang loop(i1,j1,k1,i2,j2) interchange permutation(j1,k1,i1,j2,i2)'
 128 128 100 0.239 62.40195393562317]
```

Extralarge dataset:

```
Performance summary based on 200 evaluations:
Min:  1.9069999999999998 s
Max:  16.202 s
Mean:  3.809889999999997 s
The best configurations (for the smallest time) of P0, P1, P2, P3, P4 and P5 is:

P0     P1     P2     P3     P4    P5      execution time         elapsed time

['#pragma clang loop(j2) pack array(A) allocate(malloc)'
 '#pragma clang loop(i1) pack array(B) allocate(malloc)'
 '#pragma clang loop(i1,j1,k1,i2,j2) interchange permutation(j1,k1,i1,j2,i2)'
 64 50 256 1.9069999999999998 924.4977309703827]
```

We compare the performance as shown in the following table:

Compiler and options	Large	ExtraLarge

Gcc 7.2 -O3	4.250 s	60.130 s
Clang 10.0 -O3	3.817 s	58.848 s
Clang 10.0 -O3 with polly -std=c99 -fno-unroll-loops -O3 -mllvm -polly -mllvm -polly-process-unprofitable -mllvm -polly-use-llvm-names -ffast-math -march=native	0.709 s	6.268 s
Clang 10.0 -O3 with polly and tiling (96, 2048, 256) and interchange -std=c99 -fno-unroll-loops -O3 -mllvm -polly -mllvm -polly-process-unprofitable -mllvm -polly-use-llvm-names -ffast-math -march=native	0.265 s	2.041 s
Clang 10.0 -O3 with polly and autotuning (128 128 100) (64 50 256) -std=c99 -fno-unroll-loops -O3 -mllvm -polly -mllvm -polly-process-unprofitable -mllvm -polly-use-llvm-names -ffast-math -march=native	0.239 s	1.907s

5. Identifying the important features which impact the performance

For further work, based on the output result file results.csv, we can identify which feature impacts the performance most so that the parameter search can be improved.