AutoTuneTMP: Auto Tuning in C++ With Runtime Template Metaprogramming

University of Stuttgart
Cluster of Excellence in Simulation Technology

David Pfander
Dirk Pflüger

David.Pfander@ipvs.uni-stuttgart.de
Dirk.Pflueger@ipvs.uni-stuttgart.de

Motivation

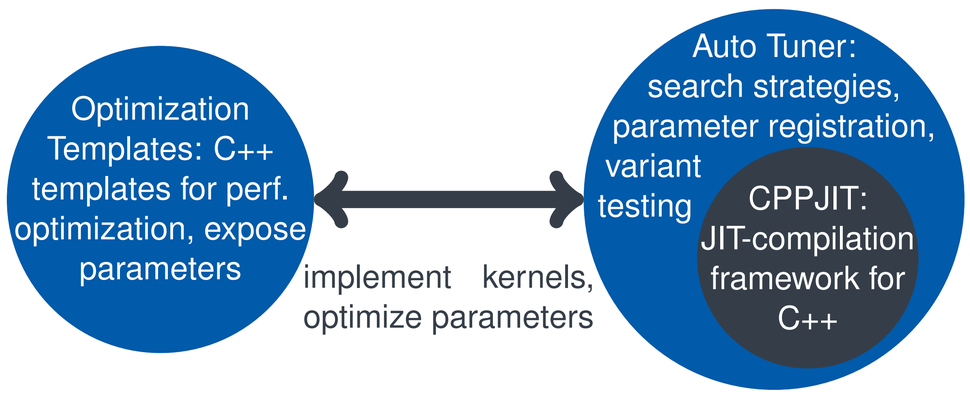
- Optimal node-level performance requires extensive optimizations
- Auto tuning: Formulate algorithm, so that it can automatically adjust to the underlying hardware platform in order to maximize performance
- We present the new auto tuning framework AutoTuneTMP, which combines JIT compilation with C++ metaprogramming for easier auto tuning

Related Work

Prior approaches focus on domain-specific languages, introduce languages or use compile-time templates:

- ATLAS [1], SPIRAL [2] and FFTW [3] provide excellent performance in their domains
- Active Harmony [4] is a framework for fast, distributed variant search, exposing parameters is left to the user
- OCCA [5] introduces a new portable language with JIT support
- SkePU [6] uses C++ templates to implement kernels. It performs auto tuning without JIT compilation
- Apollo focuses on the fast selection of kernel variants through machine learning[7]. It does not make use of JIT compilation.

AutoTuneTMP



CPPJIT

Compilation and invocation of JIT-compiled kernel

```
CPPJIT_DECLARE_DEFINE_KERNEL(int(int, int), sum)
int main(void) {
  // set source, compile and load kernel
  cppjit::sum.compile("examples/kernel_sum");
  // now use it like any other function
  int r = cppjit::sum(2, 3);
  return 0;
}

/* compute kernel in separate file "sum.cpp" */
extern "C" int sum(int a, int b) { return a + b; }
```

- Prior template approaches only support a limited search space, as variants have to be instantiated at compile time
- JIT compilation enables vast parameter spaces
- Backend is standard compiler (gcc . . .), kernel is loaded at runtime
- Near-seamless integration into application, full C++ language can be used in kernel, application and compute kernel share memory, . . .

Optimization Templates

Tiling of matrix *M* into 16x16 blocks, tile sizes are tunable

```
std::vector<memory_layout::tiling_info_dim> conf(2);
conf[0].tile_size_dir = 16; conf[0].stride = stride_x;
conf[1].tile_size_dir = 16; conf[1].stride = stride_y;
auto M_tiled = memory_layout::make_tiled<2>(M, conf);
```

- We provide generic HPC data structures: currently struct-of-array layout abstraction and d-dimensional array tiling
- Furthermore, we provide loop templates for full or partial loop unrolling, and loop nest templates for semi-automatic blocking

- Many templates have parameters that require tuning for optimal performance: unrolling factor, block sizes for blocked loop nest, . . .
- User-supplied parameters can be registered and tuned, too

Auto Tuner

Register parameters and test function, then tune kernel

```
AUTOTUNE_DECLARE_DEFINE_KERNEL(int(const int), add_one)
int main(void) {
  // register parameters
  autotune::add_one.add_parameter("UNROLL_LOOP", {"0", "1"});
  // define test function for kernel invocations
  auto test = [](const int &r) {/* ... */};
  // tune with line search
  autotune::tuners::line_search tuner(autotune::add_one, test);
  // input for variant evaluation
  const size_t a = 5;
  auto optimal_indices = tuner.tune(a);
  /* use kernel with optimized parameters */
```

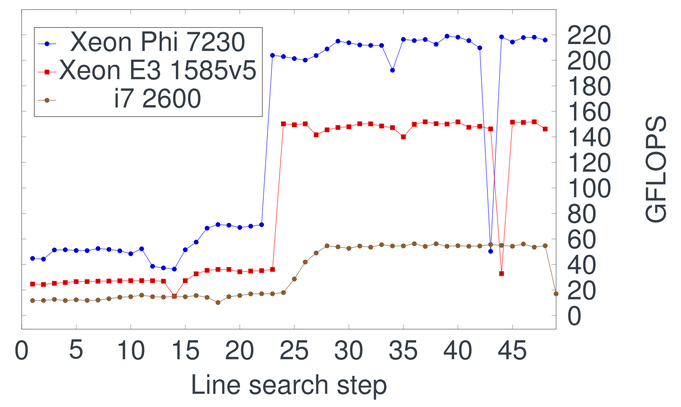
- Auto tuner component extends CPPJIT
- Permits registration of parameters and provides search strategies
- Currently implemented search strategies: bruteforce, line search, . . .
- Use parameter values by including parameter header in kernel

First Results for an Auto-Tuned DGEMM

- DGEMM shows strong benefits from tiling of the matrix into small submatrices that are processed by threads individually
- Parallelized and vectorized (with Vc [8]) application, tiling is implemented through optimization templates
- Goal: Tune L1 and L2 block sizes, based on given value range
- Kernel with 6 parameters and small value set each:

L2X	L2Y	L2K	L1X	L1Y	L1K
15, 35, 70, 140, 175	16, 32, 64, 128, 256	16, 32, 64, 128, 256	5, 10, 35, 70	8, 16, 32, 64	1, 32, 64, 128

- 8000 parameter combinations/kernel variants to tests
- Platforms: Xeon Phi 7230 (64C, 1.3GHz, 2662GF), Xeon E3-1585v5 (4C, 3.5GHz, 224GF), i7 2600 (4C, 3.4GHz, 109GF)
- Development of performance with line search strategy (48 steps):



- Algorithm reaches 68% of peak performance on Xeon E3, 61% on i7
- Xeon Phi requires further optimizations for optimal performance
- Best parameters for both devices:

	L2X	L2Y	L2K	L1X	L1Y	L1K
E3-1585v5	140	256	32	10	8	32
Phi 7230	70	64	32	5	8	32

Next Steps

- Implementation of further loop transformation templates that control parallelization, via light-weight threading framework HPX [9]
- Experiments with Active Harmony as optimization backend
- Real world application: FMM kernels of large astrophysics code
- Real world application: sparse grids data mining kernels

GitHub: https://github.com/DavidPfander-UniStuttgart/AutoTuneTMP

References

[1] R. Clint Whaley and Jack J. Dongarra. “Automatically Tuned Linear Algebra Software”. 1998.
[2] Markus Püschel et al. “Spiral: A Generator for Platform-Adapted Libraries of Signal Processing Alogorithms”. 2004.
[3] M. Frigo and S. G. Johnson. “FFTW: an adaptive software architecture for the FFT”. 1998.
[4] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. “Active Harmony: Towards Automated Performance Tuning”. 2002.
[5] David S. Medina, Amik St.-Cyr, and Timothy Warburton. “OCCA: A unified approach to multi-threading languages”. 2014.
[6] Johan Enmyren and Christoph W. Kessler. “SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems”. 2010.
[7] D. Beckingsale et al. “Apollo: Reusable Models for Fast, Dynamic Tuning of Input-Dependent Code”. 2017.
[8] Matthias Kretz and Volker Lindenstruth. “Vc: A C++ library for explicit vectorization”. 2012.
[9] Hartmut Kaiser et al. “HPX: A Task Based Programming Model in a Global Address Space”. 2014.