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

Extended Abstract

David Pfander
University of Stuttgart
Stuttgart, Germany
David.Pfander@ipvs.uni-stuttgart.de

Dirk Pflüger
University of Stuttgart
Stuttgart, Germany
Dirk.Pflueger@ipvs.uni-stuttgart.de

ABSTRACT

Maximizing the performance on modern hardware platforms has become more and more difficult, due to different levels of parallelism and complicated memory hierarchies. Auto tuning can help developers to address these challenges by writing code that automatically adjusts to the underlying hardware platform. AutoTuneTMP is a new C++-based auto tuning framework that uses just-in-time compilation to enable runtime-instantiable C++ templates. We use C++ template metaprogramming to provide data structures and algorithms that can be used to develop tunable compute kernels. These compute kernels can be tuned with different optimization strategies. We demonstrate for a first prototype the applicability and usefulness of our approach by tuning 6 parameters of a DGEMM implementation, achieving 68% peak performance on an Intel Skylake processor.

CCS CONCEPTS

• **Software and its engineering → Software performance**; *Just-in-time compilers*;

KEYWORDS

auto tuning, template metaprogramming, node-level performance

ACM Reference Format:
David Pfander and Dirk Pflüger. 2017. AutoTuneTMP: Auto Tuning in C++ With Runtime Template Metaprogramming. In *Proceedings of SuperComputing 2017, Denver, Colorado USA, November 2017 (SC'17),* 3 pages.

1 INTRODUCTION

In the last years, maximizing the performance on a modern hardware platform has become more and more difficult. The number of cores has increased and the vector units have become wider. At the same time, the gap between the computational throughput of the cores and the bandwidth of the memory has widened. All of these aspects are highly relevant to fully utilize the computational resources of modern hardware and have to be considered by HPC programmers. Auto tuning helps to address these issues.

SC'17, November 2017, Denver, Colorado USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.

Algorithms with auto tuning capabilities adjust themselves to the underlying hardware platform to maximize performance. Usually, this is achieved by tuning performance-critical parameters that are exposed by the algorithm to be tuned.

In this work, we propose a new auto tuning framework called AutoTuneTMP that consists of three components. A just-in-time (JIT) compilation framework, an auto tuner that searches for optimal parameter combinations, and a collection of C++ templates that help to expose performance-critical parameters. Our approach differs from other approaches in two important aspects. Because we seamlessly integrate with C++, the full C++ language can be used to develop complex, scientific codes. Furthermore, our approach overcomes the limitations of C++ templates through our JIT compilation component, which makes it possible to instantiate templates not only during compilation, but also at runtime. C++ runtime-templates enable us to provide generic, parameterized data structures and algorithms for optimization.

2 RELATED WORK

The framework Active Harmony enables an efficient search of the parameter space through numerous search strategies[2].

There are several auto tuners that deliver near-optimal performance in specific domains. The ATLAS project provides highly-efficient auto-tuned linear algebra routines[8]. Similarly, projects like Spiral and FFTW enable the automated generation of high-performance DSP algorithms[4, 7].

The frameworks SkePU and Apollo are most similar to our approach. SkePU provides a collection of C++ templates that support the programming of efficient algorithms[3]. Apollo focuses on the fast selection of kernel variants at runtime through machine learning[1]. Neither framework uses JIT compilation.

OCCA introduces a new language that enables portability through JIT-compilation[6]. It, however, does not support auto tuning.

3 AUTOTUNETMP

In this section, we present AutoTuneTMP in more detail. We introduce our JIT compilation component, our collection of optimization templates and our auto tuner.

3.1 CPPJIT

Templates are well-suited for the implementation of auto tuning frameworks. They make it possible to write parameterized data structures and algorithms, which can be used to program tunable algorithms. However, because templates are instantiated at compile-time, the compilation of numerous variants of a compute 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; }
```

Figure 1: Declaration, definition and use of the compute kernel sum.

leads to very long compile times and large binary files. Because of this, classical template-based approaches only work for kernels with small parameter spaces. By combining templates with a JIT compilation component, runtime-instantiable templates are obtained. Because the templates are instantiated in JIT-compiled code, they can depend on variable values of the surrounding application. With this approach, template-based auto tuning can be applied to kernels with vast parameter spaces.

In Fig. 1, we show how CPPJIT integrates with C++ and how a JIT-compiled kernel can be used. The kernel is declared and defined with macros that take signature and name of the compute kernel as arguments. In this example, the compute kernel `cppjit::sum` is generated. To JIT-compile the kernel, the member function `compile` is called. A standard C++ compiler like gcc can be specified for JIT compilation. After the kernel is compiled, it is loaded into the running application. The kernel has to be declared with C-linkage, because loading the kernel is implemented with POSIX C-calls[1]. In the kernel, arbitrary C++ constructs can be used and the kernel shares its address space with the calling application, no memory transfers are required.

3.2 Optimization Templates

We provide a collection of parameterized templates that address some commonly encountered optimization challenges. This collection of templates is continuously expanded to improve the applicability of our framework. Currently, we have an abstraction for struct-of-array (SoA) data structures that are useful for writing vectorized kernels. Furthermore, we have a template for tiling d-dimensional arrays, which can be used to improve the spatial and temporal locality of an algorithm. The tiling template can be combined with a template for the configurable iteration of tiles, this is also called blocking. We furthermore provide a template for loop unrolling, which can be used to avoid the overhead of short loops with few instructions in their loop body.

Some of these templates provide parameters that can be tuned. The size of the tiles of the tiling template is configurable with several parameters, as is the tiled traversal. The loop unrolling template enables loops to be fully unrolled, partially unrolled or not unrolled at all.

[1]This does not limit the use of C++ in the kernel

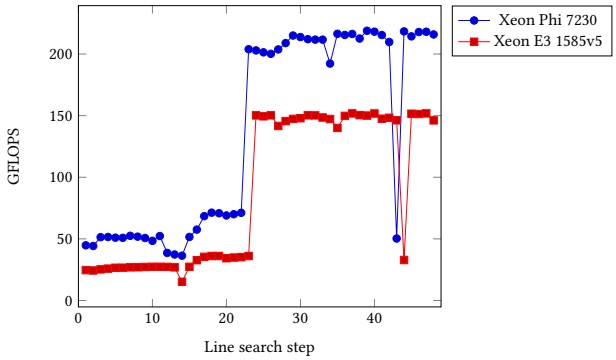

Figure 2: Performance of the DGEMM kernel during the optimization process.

3.3 Auto Tuner

The auto tuner component of AutoTuneTMP extends CPPJIT and integrates it with the optimization templates. It provides an API for registering the parameters of a kernel together with their value ranges. Furthermore, this component provides search strategies to find good or even optimal parameter combinations in the parameter space. Currently, our auto tuner implements brute-force, line search, Monte Carlo and simulated annealing as search strategies.

4 FIRST RESULTS FOR AN AUTO-TUNED DGEMM

To validate our approach, we implemented a tunable, parallelized and vectorized DGEMM that uses two of our optimization templates. The tiling template is used to split the possibly large matrix into smaller tiles that are processed by threads individually. We then use our tiled iteration template to implement two levels of blocking for the L1 and L2 caches. In this tuning scenario, we need to find values for 6 parameters that are exposed by the blocking approach.

We ran our experiments on two hardware platforms. An Intel Xeon E3-1585v4 processor with four cores at a clock rate of 3.5GHz and a double precision peak performance of ≈224GFlops. And an Intel Xeon Phi 7230 processor with 64 cores that are clocked at 1.3GHz. Its peak performance is ≈ 2662 double precision GFlops.

Fig. 2 shows the performance of the DGEMM kernel when tuned with 49 steps of a line search algorithm and a matrix size of 4096×4096. Within a few steps, a reasonable parameter combination is detected, significantly improving the performance. On the Xeon E3 platform, 150GFlops are achieved after tuning. This corresponds to 68% of the peak performance of the processor.

5 NEXT STEPS

As a next step, we are adding additional optimization templates, mainly targeting parallelization with light-weight threads. For this, we will make use the parallelization framework HPX[5]. Furthermore, we plan to integrate AutoTuneTMP with Active Harmony, because it offers a wide range of search strategies. And finally, we are working on a large astrophysics application, for which additional parameterized templates have already been developed.

REFERENCES

[1] D. Beckingsale, O. Pearce, I. Laguna, and T. Gamblin. 2017. Apollo: Reusable Models for Fast, Dynamic Tuning of Input-Dependent Code. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 307–316. https://doi.org/10.1109/IPDPS.2017.38

[2] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. 2002. Active Harmony: Towards Automated Performance Tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1–11. http://dl.acm.org/citation.cfm?id=762761.762771

[3] Johan Enmyren and Christoph W. Kessler. 2010. SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications (HLPP '10)*. ACM, New York, NY, USA, 5–14. https://doi.org/10.1145/1863482.1863487

[4] M. Frigo and S. G. Johnson. 1998. FFTW: an adaptive software architecture for the FFT. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, Vol. 3. 1381–1384 vol.3. https://doi.org/10.1109/ICASSP.1998.681704

[5] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS '14)*. ACM, New York, NY, USA, Article 6, 11 pages. https://doi.org/10.1145/2676870.2676883

[6] David S. Medina, Amik St.-Cyr, and Timothy Warburton. 2014. OCCA: A unified approach to multi-threading languages. *CoRR* abs/1403.0968 (2014). http://arxiv.org/abs/1403.0968

[7] Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W. Johnson. 2004. Spiral: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms. *Int. J. High Perform. Comput. Appl.* 18, 1 (Feb. 2004), 21–45. https://doi.org/10.1177/1094342004041291

[8] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC '98)*. IEEE Computer Society, Washington, DC, USA, 1–27. http://dl.acm.org/citation.cfm?id=509058.509096