OptiTrust: an Interactive Optimization Framework

Thomas Kœhler
Arthur Charguéraud
Begatim Bytyqi
Damien Rouhling
Inria
France

Yann Barsamian
École Européenne de Bruxelles
Belgium

Abstract

We present OptiTrust, an interactive framework for optimizing general-purpose C code via series of programmer-guided, source-to-source transformations. Optimization steps are described in transformation scripts, expressed as OCaml programs. At every step, the programmer may interactively visualize the effect of the transformation as the difference between two pieces of human-readable C code. OptiTrust has been previously employed to optimize numerical simulation code. In this work, we showcase how to use OptiTrust to optimize matrix multiplication. We compare against TVM, which also relies on programmer guidance, but which restricts the input language and lacks easily readable feedback.

1 Motivation

Achieving high-performance on modern, parallel and heterogeneous hardware is a challenging task that is critical in many domains, e.g., numerical simulation, image processing and machine learning. It remains common industrial practice to optimize code *by hand* in languages such as C/C++ [24, 31] or Fortran [32].

Manual code rewriting is highly time consuming, and must be repeated numerous times to reach high performance. Several optimization paths must be empirically explored because code performance is hard to assess without benchmarking. After this process, hand optimized code is generally hard to read, maintain, adapt to other hardware, and—worst of all—may contain bugs introduced in the process.

In order to reuse optimization efforts, *optimized libraries* such as BLAS or MKL for linear algebra were created. However, their use remains suboptimal as state-of-the-art general-purpose compilers (e.g. Clang, GCC, ICX) fail to apply relevant optimizations across library calls. This limitation has motivated the development of *specialized optimizing compilers*, able to apply global optimizations on high-level algorithms written in *Domain-Specific Languages* (DSLs).

Examples of specialized compilers include Halide [26] and TVM [14], which have proved successful in industry for image processing and machine learning, respectively. With slightly larger application domain are compilers for *array programming* languages [9, 17, 20, 29], exploiting the fact that multi-dimensional arrays are a key abstraction for many performance-demanding applications. Although the success of specialized optimizing compilers is undeniable, they cannot optimize programs that fall outside of their domain. Morevover, extending these specialized compilers has proved to be challenging [5].

OptiTrust is an optimization framework that aims to support general-purpose code, avoiding domain restriction. OptiTrust operates at the level of C code, by supporting source-to-source transformations. The programmer interactively develops a transformation script. At every transformation step, the programmer may visualize the corresponding *diff* between pieces of human-readable C code.

Prior work [12] has demonstrated the ability of OptiTrust to reproduce a state-of-the-art optimized implementation of a numerical Particle-In-Cell simulation. In the present work, we demonstrate the ability of OptiTrust to reproduce a reference implementation of matrix multiplication generated by the specialized compiler TVM.

2 The 5 Key Components of OptiTrust

1. An internal, simplified abstract syntax tree (AST) from which readable C code can be recovered throughout transformations. A similar idea of a simplified AST has been employed in the Cetus project [16].
2. A system for targeting program points, somewhat similar to XPath [15] for XML, but specialized for an AST. This system allows describing, in a concise and robust manner, one or several program points to transform.
3. A library of general-purpose transformations, including: core data layout transformations [30], instruction-level transformations [1], control flow transformations [33].
4. A scripting language, embedded in OCaml, for describing transformations. Transformation scripts are a classic technique for programmer-guided optimization frameworks [4, 6, 7, 13, 25, 27, 34–36]. Transformation scripts provide fine-grained control, and they allow chaining large numbers of transformation steps. Other approaches include use of *pragmas* [10, 22, 23], schedules [3, 14, 18, 26], or rewriting strategies [2, 8, 19, 21].
5. An interactive interface allowing to visualize code *diffs* associated with the transformation at a given line of the script, via a key shortcut in the code editor.

3 Optimizing Matrix-Multiply in OptiTrust

We investigate how OptiTrust compares with the specialized compiler TVM for producing an optimized implementation of matrix multiplication—a standard benchmark.

```
void mm(float* C, float* A, float* B, int m, int n, int p) {
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      float sum = 0.0f;
      for (int k = 0; k < p; k++) sum += A[i][k] * B[k][j];
      C[i][j] = sum; }}}
void mm1024(float* C, float* A, float* B) {
  mm(C, A, B, 1024, 1024, 1024); }
```

Listing 1. Unoptimized matrix multiplication. The function `mm` multiplies the matrices `A` and `B` and stores the result in `C`. The function `mm1024` specializes input sizes to 1024.

We start from the C code presented in Listing 1: a naive, unoptimized implementation of matrix multiplication. The aim is to produce similar code as the reference TVM *schedule* that was handwritten by an expert targeting Intel CPUs.[1]

We write an OptiTrust script in OCaml, as shown in Listing 2. It consists of 9 transformation steps, each one calling a function from the OptiTrust library, always providing a *target* describing where to apply the transformation as last argument. OptiTrust scripts are developed interactively: with the cursor on a given line of the script, we can press (e.g.) "F6" in the VSCode editor to visualize the *diff* associated with the transformation on that line. All intermediate versions of the code consist of human-readable, executable C code.

The final code is shown in Listing 3. It mirrors the structural optimizations described in the TVM reference schedule, including: improved data locality, parallelism, and specialization to 1024 sizes. A major difference is that we produce C code with OpenMP pragmas, whereas TVM targets directly LLVM IR. To check that the code produced using OptiTrust performs similarly to that produced by TVM, we benchmarked both codes on a 4-core Intel i7-8665U CPU with AVX2 support. Both codes have 90th percentile runtime of 9.4ms over 200 benchmark runs, corresponding to a speedup of 150× compared to the 90th percentile of the naive code.[2]

In summary, our case study shows that the OptiTrust general-purpose optimization framework can be used to interactively develop a code competitive with that produced by a state-of-the-art specialized compiler. Unlike specialized compilers, OptiTrust takes as input C code, and produces human-readable C code at every step.

4 Contents of the Talk

During the ARRAY workshop, we will present the OptiTrust matrix multiplication case study, and compare the script to the reference TVM schedule. We will give an overview of the transformations currently supported by OptiTrust, and describe ongoing work on leveraging Separation Logic annotations [11, 28] to validate transformation correctness

```
~~List.iter [("i", 32); ("j", 32); ("k", 4)] (fun (id, tile_size) ->
    Loop.tile (trm_int tile_size) ~index:("b" ^ id)
      ~bound:TileDivides [cFor id);
Loop.reorder_at ~order:["bi"; "bj"; "bk"; "i"; "k"; "j"]
  [cPlusEqVar "sum"];
Loop.hoist_expr ~dest:[tBefore; cFor "bi"] "pB"
  ~indep:["bi"; "i"] [cArrayRead "B"];
Function.inline ~delete:true [cFun "mm"];
Matrix.stack_copy ~var:"sum" ~copy_var:"s" ~copy_dims:1
  [cFor ~body:[cPlusEqVar "sum"] "k"];
Matrix.elim_mops [];
Loop.unroll [cFor ~body:[cPlusEqVar "s"] "k"];
Omp.simd [nbMulti; cFor ~body:[cPlusEqVar "s"] "j"];
Omp.parallel_for [nbMulti; cFunDef"mm1024"; dBody; cStrict; cFor""];
```

Listing 2. OptiTrust script for optimizing `mm1024`.

```
                                                  float s[32];
                                                  memcpy(s, &sum[i][0], sizeof(float[32]));
for (int k = 0; k < 4; k++) {                     for (int k = 0; k < 4; k++) {
  for (int j = 0; j < 32; j++) {                    for (int j = 0; j < 32; j++) {
    sum[i][j] += A[bi * 32 + i][bk * 4 + k]           s[j] += A[bi * 32 + i][bk * 4 + k]
                 * pB[bj][bk][k][j];                          * pB[bj][bk][k][j];
  }                                                 }
}                                                 }
                                                  memcpy(&sum[i][0], s, sizeof(float[32]));
```

Figure 1. Interactive *diff* for the `Matrix.stack_copy` step.

```
float* pB = (float*)malloc(sizeof(float[32][256][4][32]));
#pragma omp parallel for
for (int bj = 0; bj < 32; bj++) {
  for (int bk = 0; bk < 256; bk++) {
    for (int k = 0; k < 4; k++) {
      for (int j = 0; j < 32; j++) {
        pB[32768 * bj + 128 * bk + 32 * k + j] =
          B[1024 * (4 * bk + k) + 32 * bj + j]; }}}}
#pragma omp parallel for
for (int bi = 0; bi < 32; bi++) {
  for (int bj = 0; bj < 32; bj++) {
    float* sum = (float*)malloc(sizeof(float[32][32]));
    for (int i = 0; i < 32; i++) {
      for (int j = 0; j < 32; j++) {
        sum[32 * i + j] = 0.; }}
    for (int bk = 0; bk < 256; bk++) {
      for (int i = 0; i < 32; i++) {
        float s[32];
        memcpy(s, &sum[32 * i], sizeof(float[32]));
        #pragma omp simd
        for (int j = 0; j < 32; j++) {
          s[j] += A[1024 * (32 * bi + i) + 4 * bk + 0] *
                  pB[32768 * bj + 128 * bk + 32 * 0 + j]; }
        // [...] 4 other similar loops
        memcpy(&sum[32 * i], s, sizeof(float[32])); }}
    for (int i = 0; i < 32; i++) {
      for (int j = 0; j < 32; j++) {
        C[1024 * (32*bi + i) + 32*bj + j] = sum[32*i + j]; }}
// [...] free instructions
```

Listing 3. Optimized C code produced by the OptiTrust script for `mm1024`. This code has similar structure and achieves similar performance as the reference output of TVM.

beyond the limit of static analyses commonly used in general-purpose compilers.

We hope to engage with the ARRAY community by presenting an approach that tackles the limitations of specialized langages and compilers. Suggestions of challenging optimizations, representative benchmarks, and compilation techniques to investigate will be especially welcomed.

[1] https://tvm.apache.org/docs/how_to/optimize_operators/opt_gemm.html

[2] For the TVM code, the median is 2% faster than the 90th percentile; for the OptiTrust code, it is 8%. The manually optimized library Intel MKL delivers a 204× speedup compared to the 90th percentile of the naive code.

References

[1] J. Allen and K. Kennedy. 2002. *Optimizing Compilers for Modern Architectures.*

[2] O.S. Bagge, K.T. Kalleberg, M. Haveraaen, and E. Visser. 2003. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation.* 65–74.

[3] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *IEEE/ACM International Symp. on Code Generation and Optimization (CGO).* 193–205.

[4] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2016. Opening Polyhedral Compiler's Black Box. In *IEEE/ACM International Symp. on Code Generation and Optimization.*

[5] Paul Barham and Michael Isard. 2019. Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems.* 177–183.

[6] João Bispo and João MP Cardoso. 2020. Clava: C/C++ source-to-source compilation using LARA. *SoftwareX* 12 (2020), 100565.

[7] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI'08 ACM Conf. on Programming language design and implementation.*

[8] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Sci. Comput. Program.* 72, 1–2 (jun 2008), 52–70.

[9] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming.* 3–14.

[10] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP.*

[11] Arthur Charguéraud. 2020. Separation Logic for Sequential Programs (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 116 (aug 2020), 34 pages.

[12] Arthur Charguéraud, Begatim Bytyqi, Damien Rouhling, and Yann A Barsamian. 2022. OptiTrust: an Interactive Framework for Source-to-Source Transformations. (Sept. 2022). https://hal.inria.fr/hal-03773485 working paper.

[13] Chun Chen, Jacqueline Chame, and Mary W. Hall. 2008. *CHiLL: A Framework for Composing High-Level Loop Transformations.* Technical Report 08-897. University of Southern California.

[14] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI.*

[15] James Clark, Steve DeRose, et al. 1999. XML path language (XPath). https://www.w3.org/TR/1999/REC-xpath-19991116/

[16] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. 2009. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. *Computer* 42, 12 (2009), 36–42.

[17] Clemens Grelck and Sven-Bodo Scholz. 2006. SAC—a functional array language for efficient multi-threaded execution. *International Journal of Parallel Programming* 34 (2006), 383–427.

[18] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. 2020. Fireiron: a data-movement-aware scheduling language for GPUs. In *Proceedings of the ACM International Conf. on Parallel Architectures and Compilation Techniques.* 71–82.

[19] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29.

[20] Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation.* 556–571.

[21] Hélène Kirchner. 2015. Rewriting strategies and strategic rewrite programs. In *Logic, Rewriting, and Concurrency: Essays Dedicated to José Meseguer on the Occasion of His 65th Birthday.* 380–403.

[22] Michael Kruse and Hal Finkel. 2018. A Proposal for Loop-Transformation Pragmas. (2018). arXiv:1805.03374

[23] Olaf Krzikalla, Kim Feldhoff, Ralph Müller-Pfefferkorn, and Wolfgang E. Nagel. 2011. Scout: A Source-to-Source Transformator for SIMD-Optimizations. In *Euro-Par Workshops (2) (LNCS, Vol. 7156).*

[24] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. 2017. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing.* 1–6.

[25] Kedar S. Namjoshi and Nimit Singhania. 2016. Loopy: Programmable and Formally Verified Loop Transformations. In *Static Analysis - 23rd International Symposium, SAS (LNCS, Vol. 9837).*

[26] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Conf. on Programming Language Design and Implementation.* 12 pages.

[27] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. 2011. A Programming Language Interface to Describe Transformations and Code Generation. In *Languages and Compilers for Parallel Computing.*

[28] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the foundational verification of C code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conf. on Programming Language Design and Implementation.* 158–174.

[29] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. *ACM SIGPLAN Notices* 50, 9 (2015), 205–217.

[30] I-Jui Sung, John A. Stratton, and Wen-Mei W. Hwu. 2010. Data Layout Transformation Exploiting Memory-level Parallelism in Structured Grid Many-core Applications *(PACT '10).* 513–522.

[31] Ashkan Tousimojarad and Wim Vanderbauwhede. 2014. Comparison of three popular parallel programming models on the Intel Xeon Phi. In *Euro-Par 2014: Parallel Processing Workshops: Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II 20.* Springer, 314–325.

[32] Wim Vanderbauwhede and Gavin Davidson. 2018. Domain-specific acceleration and auto-parallelization of legacy scientific code in FORTRAN 77 using source-to-source compilation. *Computers & Fluids* 173 (2018), 1–5.

[33] M. Wolfe. 1995. *High performance compilers for parallel computing.*

[34] Qing Yi and Apan Qasem. 2008. Exploring the Optimization Space of Dense Linear Algebra Kernels. In *LCPC.*

[35] Qing Yi, Qian Wang, and Huimin Cui. 2014. Specializing Compiler Optimizations through Programmable Composition for Dense Matrix Computations. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, United Kingdom) *(MICRO-47).* USA, 596–608.

[36] Oleksandr Zinenko, Lorenzo Chelini, and Tobias Grosser. 2018. *Declarative Transformations in the Polyhedral Model.* Research Report RR-9243.