ISSN 2186-7437

NII Shonan Meeting Report

No. 2012-4

Bridging the theory of staged programming languages and the practice of high-performance computing

Oleg Kiselyov
Chung-chieh Shan
Yukiyoshi Kameyama

May 19–22, 2012

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan

Bridging the theory of staged programming languages and the practice of high-performance computing

Organizers:
Oleg Kiselyov (Monterey, CA, USA)
Chung-chieh Shan (Cornell University, USA)
Yukiyoshi Kameyama (University of Tsukuba, Japan)

May 19–22, 2012

We organized a discussion-heavy NII Shonan Meeting to bridge the theory of programming languages (PL) with the practice of high-performance computing (HPC). The topic of the discussion was code generation, or *staging*, a form of meta-programming. Both the PL and HPC communities have come to realize the importance of code generation: whereas PL theorists widely regard staging as the leading approach to making modular software and expressive languages run fast, HPC practitioners widely regard staging as the leading approach to making high-performance software reusable and maintainable. This confluence of theory and practice gave us a rare chance to bring together PL researchers and the potential consumers of their work.

A particular area of current interest shared by PL and HPC researchers is how to use *domain-specific languages* (DSL) to capture and automate patterns and techniques of code generation, transformation, and optimization that recur in an application domain. For example, HPC has created and benefited from expressive DSLs such as OpenMP directives, SPIRAL's signal processing language, and specialized languages for stencil computations and domain decomposition. Moreover, staging helps to build efficient and expressive DSLs because it assures that the generated code is correct in the form of precise static guarantees.

Alas, the communication between PL researchers working on staging and HPC practitioners could be better. On one hand, HPC practitioners often do not know what PL research offers. On the other hand, PL researchers often do not know how much HPC practitioners who write code generators value this or that theoretical advance or pragmatic benefit—in other words, how the HPC wish list is ranked by importance. We thus aimed to solicit and discuss real-world applications of assured code generation in HPC that would drive PL research in meta-programming. Specifically, we tried to determine

- how viable assured (MetaOCaml-like) meta-programming is for real-world applications;
- how strong the demand is for static assurances on the generated code: well-formedness, well-typedness, numeric stability, absence of buffer overflows

or dimension mismatch errors, etc.;

- how important portability is, whether to move to a different target language or a different hardware platform;
- which difficulties are "just" engineering (e.g., maintaining a viable, mature meta-programming system), which difficulties are of finding a good syntax, and which difficulties are foundational (e.g., code generation with binders and effects).

In short, we asked how program generation can or should help HPC.

The participants consisted of three groups of people: PL theorists, HPC researchers, and PL-HPC intermediaries (that is, people who are working with HPC professionals, translating insights from PL theory to HPC practice). To promote mutual understanding and benefit, we scheduled lots of time for discussion, and we favored presentations that elicit questions rather than merely provide answers.

The final list of participants besides organizers at this NII Shonan Meeting is as follows:

1. Ashish Agarwal, New York University (USA)
2. Baris Aktemur, Ozyegin University (Turkey)
3. Kenichi Asai, Ochanomizu University (Japan)
4. Shigeru Chiba, Tokyo Institute of Technology (Japan)
5. Albert Cohen, INRIA & ENS (France)
6. Zhenjiang Hu, National Institute of Informatics (Japan)
7. Atsushi Igarashi, Kyoto University (Japan)
8. Andrei Klimov, Keldysh Institute of Applied Mathematics, Russian Academy of Sciences (Russia)
9. Ken Friis Larsen, University of Copenhagen (Denmark)
10. Frédéric Loulergue, University of Orléans (LIFO) (France)
11. Geoffrey Mainland, Microsoft Research Cambridge (UK)
12. Takayuki Muranushi, Kyoto University (Japan)
13. Tiark Rompf, EPFL (Switzerland)
14. Alexander Schliep, Rutgers University (USA)
15. Jeremy Siek, University of Colorado at Boulder (USA)
16. Matthew Sottile, Galois, Inc. (USA)
17. Reiji Suda, University of Tokyo (Japan)
18. Eijiro Sumii, Tohoku University (Japan)
19. Walid Taha, Halmstad University (Sweden)

20. Daisuke Takahashi, University of Tsukuba (Japan)

21. Peter Thiemann, Universitaet Freiburg (Germany)

In addition to formal individual talks and informal discussion groups, each participant also gave a 5-minute self-introduction.

Tangible Outcomes

Introducing staging tools

Besides the venerable MetaOCaml language, the participants were also interested in using Lightweight Modular Staging in Scala. Accordingly, in addition to Walid Taha's introductory talk on staging, Tiark Rompf gave a tutorial of installing and using the Delite framework in Scala.

Introducing staging needs

Many participants expressed a desire for *portable performance*. Existing concepts and techniques for improving performance, such as the byte/flop ratio and array padding, were introduced to the participants throughout the meeting, especially during the introductory talks by HPC practitioners Reiji Suda and Daisuke Takahashi. Our discussions then suggested that the degree to which HPC techniques can be made portable by staging today varies from trivially easy to interestingly difficult.

Therefore, we decided to collect examples of what staging *should* express, i.e.,

- a high-level specification that a high-performance programmer *should* be able to write easily and
- a low-level efficient implementation that is known today but typically written by hand tediously.

In response to each challenge statement, staging researchers can then show how to implement the same optimizations and derive a similar efficient implementation automatically from a similar simple specification. Because the intention of the challenge and response is to illustrate *portable* performance within reach, the solutions should not be evaluated solely by their raw performance or by their literal identity with the challenge statement, but also by their elegance (uniformity, modularity, clarity, extensibility, etc.). The benefits of such an interaction are bidirectional:

- HPC people would obtain the efficient implementation without writing tedious code by hand.
- Staging people would obtain real-world applications to enhance the theory and technology of staging.

Kenichi Asai initiated this collection during the meeting. We decided to name the collection "Shonan Challenge". During the meeting, the following challenge examples were submitted by various participants:

1. Overlap tiling
2. FFTN
3. Sparse matrix format
4. Complex number representation
5. A staged matrix algebra library
6. A dynamic programming library
7. Integer intervals query
8. Image processing pipeline
9. A staged Hidden Markov Model library
10. OpenCV computer vision library
11. Also a list of notable HPC test code suites

Ongoing group work

As the list above indicates, during the meeting we collected a bunch of HPC examples for the Shonan Challenge. We next need to

- organize (classify, specify) the examples into a form that can be distributed and understood as a public Web page,
- respond to the challenge statements with staging solutions, and
- list new challenge statements.

We are coordinating this work by sharing a mailing list and code repository. We hope it will evolve into a report and presentation, whose abstract might read as follows (to misquote Aydemir et al.'s TPHOLs 2005 paper on the POPLmark challenge):

> How close are we to a world where
>
> - every paper on high-performance computing is accompanied by an electronic appendix with machine program generators?
> - natural-science grad students no longer need to translate their high-level formulas into Fortran?
>
> We propose an initial set of benchmarks for measuring progress in this area. These benchmarks embody many aspects of HPC that are challenging to formalize: .... We hope that these benchmarks will help clarify the current state of the art, provide a basis for comparing competing technologies, and motivate further research.

Taking a step back from the Shonan Challenge, we also generated a list of what to do to strengthen the technical and social connection between HPC and staging.

1. solve HPC problems
    - respond to and improve Shonan Challenge list
    - Matthew Sottile's list of notable HPC test code suites
    - Reiji Suda's infinite stream of HPC problems
    - from kernels to larger libraries and (one-off) applications
    - raising abstraction level (avoid $O(nm)$) across computation models (parallel processing, FPGA, GPU)
    - share patterns (intermediate representations, genetic algorithms) across domains, at this kind of event
    - stencil programs, applied math
    - data layout optimization
2. specialize code in Fortran, restricted Python, whatever people use
    - consider existing infrastructure: Tempo, Glück's Fortran specializer, PGG (easy to retarget for another output language), ROSE, Scala LMS, MetaOCaml
    - pursue and control partial evaluation
3. bring/keep MetaOCaml up to date
    - native code compiler for BER MetaOCaml
    - interest from people and support from OCaml team
    - consortium funding for a research programmer
4. develop staging technology and theory
    - need rewriting around staging
    - static safety and run are desirable but not absolutely necessary
5. advertise to two different audiences
    - users (e.g., at the International Conference on Computational Science), grad students, scientists
    - experts (e.g., at the SC Conference)
6. advertise to different venues
    - domain venues (biology, astrophysics, vision, linguistics, ...) will pay attention to impact
    - tutorials
        - ICFP (but that's not where customers are)
        - The SC Conference! The International Conference on Computational Science! They are two different audiences
        - HIPEAC (European Network of Excellence on High Performance and Embedded Architecture and Compilation)—see past tutorials

- Exhibit or "birds of a feather" session at the SC Conference or at the International Conference on Computational Science
- SIAM mini-symposia
- Web bibliography of staging applications and research
- annual StagedHPC
- state-of-the-art document
- ecosystem of (domain-specific) offerings, branded "staging", starting with gateway drug such as operator overloading

Overview of Talks

Staging 15 years later

Walid Taha, Halmstad University, Sweden

Multi-stage Programming with Explicit annotations, or staging for short, is an approach to giving the programmer control over evaluation order. Staging was introduced in 1997 in a paper by Tim Sheard, my PhD advisor, and myself. Significant activity by numerous researches followed. Many view staging as a way of improving program performance. While technically very true, there is an important reason why this can be a misleading characterization. To address this problem, this talk briefly reviews staging, describes some important lessons learned about staging, and suggests some promising directions for future work.

Performance Tuning for High Performance Computing Applications

Daisuke Takahashi, University of Tsukuba, Japan

In this talk, basic performance tuning techniques for high performance computing (HPC) applications will be presented. Many HPC applications work well when the data sets fit into the cache. However, when the problem size exceeds the cache size, the performance of these applications decreases dramatically. The key issue in the implementation of HPC applications is minimizing the number of cache misses. Vectorization and parallelization are also important to achieve high performance on multi-core processors that have short vector SIMD instructions. Some performance results will be also presented.

BINQ: A Domain Specific Language for Genomic Computations

Ashish Agarwal, NYU, USA

Unabated advances in genomics technologies continue to increase the volume and variety of data that Biologists work with on a day-to-day basis. This results in two distinct software challenges: (i) Biologists require a platform that manages the deluge of data and methods they regularly employ, and (ii) this platform must parallelize code over multiple cores and nodes. I propose BINQ, a

purely functional domain-specific-language (DSL) that allows expressing a wide range of computations commonly needed in genomics and systems biology. The type system aims to support the wide range of data and analyses required in a compact elegant language, and also to encode properties required for efficient implementation, such as whether a list is sorted. I will describe the challenges in implementing this language and explain where the parallelism opportunities lie. Finally, I will review our OCaml code base, which operates NYU's Genomics Core Facility, and serves as motivation for the design of BINQ.

Paraiso: A code generation and automated tuning framework for explicit solvers of partial differential equations

Takayuki Muranushi, Kyoto University, Japan

As a simulation astrophysicist, my colleagues and I wish for something that (1) generates optimized codes on massively parallel computers such as the K computer, GPU-based supercomputers etc. (2) can describe practical (= complicated) algorithms and (3) is easy to program. None were available. Advances in programming languages made it possible even for physicists to design their own DSL. My hope is collaboration. Parallel and abstract languages act as interfaces, letting one group decompose their problems to such interface languages and other groups work on translating to real machines, and both groups work together in developing the interface.

As an example of such abstract interface I describe the Orthotope Machine and Paraiso, a DSL framework for generating parallel machine stencil computation, embedded in Haskell. Paraiso lets one write complicated hyperbolic partial differential equations (PDE) algorithms in succinct ways, by using mathematical language like type level tensors and algebraic type classes. Higher-order operations are also possible. Paraiso turns PDE algorithms into a program on the Orthotope Machine, a parallel virtual machine with very limited instruction set. These programs are then translated into multicore or GPU implementations. Paraiso automatically searches for faster implementation by benchmarking and evolutionary computation.

Reliable Generation of High-Performance Matrix Algebra

Jeremy Siek, University of Colorado at Boulder, USA

Scientific programmers often turn to vendor-tuned Basic Linear Algebra Subprograms (BLAS) to obtain portable high performance. However, many numerical algorithms require several BLAS calls in sequence, and those successive calls result in suboptimal performance. The entire sequence needs to be optimized in concert. Instead of vendor-tuned BLAS, a programmer could start with source code in Fortran or C (e.g., based on the Netlib BLAS) and use a state-of-the-art optimizing compiler. However, our experiments show that optimizing compilers often attain only one-quarter the performance of hand-optimized code. In this talk I present a domain-specific compiler for matrix algebra, the Build to Order BLAS (BTO), that reliably achieves high performance using a scalable search algorithm for choosing the best combination of loop fusion, array contraction,

and multithreading for data parallelism. The BTO compiler generates code that is between 16% slower and 39% faster than hand-optimized code.

A Calculational Framework for Parallel Programming with MapReduce

Zhenjiang Hu, NII, Japan

MapReduce, being inspired by the map and reduce primitives available in many functional languages, is the de facto standard for large scale data-intensive parallel programming. Although it has succeeded in popularizing the use of the two primitives for hiding the details of parallel computation, little effort has been made to emphasize the programming methodology behind. In this talk, I'd show that MapReduce can be equipped with a programming theory in calculational form. By integrating the generate-and-test programing paradigm and semirings for aggregation of results, we propose a novel parallel programming framework for MapReduce. The framework consists of two important calculation theorems: the shortcut fusion theorem of semiring homomorphisms bridges the gap between specifications and efficient implementations, and the filter-embedding theorem helps to develop parallel programs in a systematic and incremental way.

This is a joint work with Kento Emoto and Sebastian Fischer.

Combining Staged Programming and Empirical Optimization

Baris Aktemur, Ozyegin University, Turkey

A major motivation behind staged programming is to speed up programs by specializing them according to runtime inputs. Applications of staged programming we see in (theoretical) papers are mostly small examples that do not scale to large data used in HPC. To remedy this problem, staged programming must be performed with awareness of how to process large data efficiently. For instance, the generated program must utilize the cache effectively. However, given the complexity of modern computer architectures, this is not a trivial task. Empirical optimization (aka auto-tuning) techniques have been successfully used in HPC to adapt programs to target machines by selecting the best version as observed in install-time experiments. I will discuss our on-going work on combining empirical optimization with staged programming to select the best generator out of several candidates. This is a joint work with Prof. Sam Kamin and his research group.

Programming language features required in high performance computing, parallel computing, and automatic tuning

Reiji Suda, University of Tokyo, Japan

The author is working on high performance computing, parallel computing, numerical algorithms, and automatic tuning (autotuning). Recently the requirements on such research are increasing because of several factors, for example,

now every processor has multiple cores. One of the difficulties in high performance computing is that the optimal choice depends on HW, SW, data and environmental conditions. To attain high performance on many conditions, software must have adaptability in it, called autotuning. For autotuning, we need to generate several variants of programs (and to choose an appropriate variant for each condition). Such variants can be generated by code transformation or code generation. Needs of explicit language support are high in variations of data structure, algorithms and parallel processing. Language support for debugging and testing of such software is also expected.

ForOpenCL: Transformations Exploiting Array Syntax in Fortran for Accelerator Programming

Matthew Sottile, Galois Inc, USA

This talk describes a small experimental extension to Fortran 2008 that was implemented via source-to-source transformations to hide the low-level OpenCL implementation of stencil-based algorithms written in Fortran. The basis of this work is the use of three notable features that were introduced in the 1990 Fortran standard: pure functions; elemental functions; and data parallel array syntax. This talk gives a brief overview of the concept and prototype that we developed based on the ROSE compiler infrastructure. The basic building block in ForOpenCL is the concept of a halo that defines for a computation on an element of an array the extent of the array around it that is required to be accessible. This pattern arises frequently in stencil-based programs. We are currently working on a deeper extension to Fortran itself, that we call "Locally Oriented Programming", where type annotations are used to express information related to halos. This halo information can be used by the compiler to determine efficient memory allocation and transfer patterns in hybrid systems and message passing operations in MPI-based programs.

MetaHaskell: Type Safe Heterogeneous Metaprogramming

Geoffrey Mainland, Microsoft Research Cambridge, UK

Languages with support for metaprogramming, like MetaOCaml, offer a principled approach to code generation by guaranteeing that well-typed metaprograms produce well-typed programs. However, many problem domains where metaprogramming can fruitfully be applied require generating code in languages like C, CUDA, or assembly. Rather than resorting to add-hoc code generation techniques, these applications should be directly supported by explicitly heterogeneous metaprogramming languages.

In this talk I will present MetaHaskell, an extension of Haskell 98 that provides modular syntactic and type system support for type safe metaprogramming with multiple object languages. Adding a new object language to MetaHaskell requires only minor modifications to the host language to support type-level quantification over object language types and propagation of type equality constraints. We demonstrate the flexibility of our approach through three object languages: a core ML language, a linear variant of the core ML language, and a subset of C. All three languages support metaprogramming

with open terms and guarantee that well-typed MetaHaskell programs will only produce closed object terms that are well-typed.

High Performance Embedded DSLs with Delite

Tiark Rompf, EPFL, Switzerland

Delite is a framework for building high performance DSLs embedded in Scala. This talk will describe what it takes to build a Delite DSL and describe several real DSLs already being developed with Delite (including machine learning, graph analysis, and scientific computing). Delite handles parallel optimization and code generation, allowing DSL authors to focus on identifying the right abstractions and implementing domain-specific optimizations. We show that DSLs implemented with Delite achieve good performance compared to standard alternatives. Finally, we look ahead to what's coming next with high performance DSLs and Delite.

Towards application of supercompilation and metacomputation to high performance computing

Andrei V. Klimov, Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, Russia

I work in a team which has experience in two areas: 1) construction of program specializers based on the methods of supercompilation and partial evaluation (which we collectively refer to as metacomputation) and 2) construction of supercomputers and development of software for them. Based on this work, we see large potential impact of metacomputation on HPC, most notable being as follows.

- Our experience shows that powerful program specializers are capable of converting well-structured human-oriented, but badly parallelizable programs into "flat" code consisting of nested loops, for which existing parallelization methods apply well.
- It is common opinion that development of new models of computation and renewing of some old ones (like dataflow) is required, which poses the problem of transforming legacy code to new forms (e.g., Fortran code to dataflow, data parallel, SIMD, pipelined code). There is no silver bullet, but such deep program analysis and transformation methods are actually emerging.
- Another widespread viewpoint is that program verification becomes much more important in the parallel era than before. Between two approaches to program verification: based on logic methods and based on deep program transformation like supercompilation, we chose the later as more promising.

Computational Effects across Generated Binders: Problems and solutions

Oleg Kiselyov

Code generation is the leading approach to making high-performance software reusable. Using a set of realistic examples, we demonstrate that side effects are indispensable in composable code generators, especially side effects that move open code past generated binders. We challenge the audience to implement these examples in their favorite code-generation framework.

We implemented the examples ourselves using a prototype library of code-generating combinators in Haskell. This library statically assures not only that all generated code is well-formed and well-typed but also that all generated variables are bound lexically as expected. Such assurances are crucial for code generators to be written by domain experts rather than compiler writers, because the most profitable optimizations are domain-specific ones.