Why New Programming Languages for Simulation?

GILBERT LOUIS BERNSTEIN
Stanford University
and
FREDRIK KJOLSTAD
Massachusetts Institute of Technology

Writing highly performant simulations requires a lot of human effort to optimize for an increasingly diverse set of hardware platforms, such as multi-core CPUs, GPUs, and distributed machines. Since these optimizations cut across both the design of geometric data structures and numerical linear algebra, code reusability and portability is frequently sacrificed for performance.

We believe the key to make simulation programmers more productive at developing portable and performant code is to introduce new linguistic abstractions, as in rendering and image processing. In this perspective, we distill the core ideas from our two languages, Ebb and Simit, that are published in this journal.

CCS Concepts: • **Computing methodologies → Simulation languages**

Additional Key Words and Phrases: Physical simulation languages, performance, compilation

ACM Reference Format:

Gilbert Louis Bernstein and Fredrik Kjolstad. 2016. Why new programming languages for simulation? ACM Trans. Graph. 35, 2, Article 20e (May 2016), 3 pages.
DOI: http://dx.doi.org/10.1145/2930661

Simulations are complicated, performance-critical applications that combine sophisticated computer science data structures with advanced mathematical computation. Best practice suggests using optimized linear algebra libraries, yet programmers of high-performance simulations invariably abandon this ideal in order to optimize computation around application data structures. Getting good performance requires a lot of human effort to manage data layout, vectorize, and parallelize code. And the situation is getting worse. High-performance systems in the near future will need to support concurrent execution on GPUs, multicore CPUs, a host of new architectures, and across distributed systems.

In response to similar trends, graphics researchers have proposed new programming languages to manage and abstract away from hardware complexity. Renderman [Upstill 1989], GPU Shading Languages, Cuda [Nvidia 2008], and more recently Halide [Ragan-Kelley et al. 2013] and Darkroom [Hegarty et al. 2014] are excellent examples. These languages promise considerable reductions in programmer effort, as reflected both by the amount of code that must be written and the degree to which programmers must optimize for specific hardware. We believe simulation is now primed to move to such programming languages. Although simulation is a complicated domain, we are starting to understand how to represent a large class of simulations with simple, general, and flexible high level programming language concepts built on solid foundations. We believe the ideas behind the relational algebra and modern databases have direct relevance to simulation, and we expect to see similar benefits to those the database community has accrued.

In these proceedings, two such languages are presented: Ebb [Bernstein et al. 2016] and Simit [Kjolstad et al. 2016]. These languages were developed independently by two separate groups centered at Stanford and MIT, and we believe they are first steps in a larger roadmap. We will let the articles speak for themselves on the strengths of their different operators and data structures. However, we encourage readers to note the common foundations we will lay out and join us in exploring a fascinating new research direction.

Linguistic Abstractions

Traditionally, stand-alone languages were developed for specific domains (e.g., MATLAB [2014] and Renderman). More recently, languages such as Halide and shading languages have been provided as libraries with language semantics and compiler support (sometimes called embedded languages). Ebb and Simit are also libraries with language semantics, which enable them to simultaneously increase performance and productivity while also being portable. The availability of the LLVM compiler infrastructure has dramatically reduced the cost of developing such libraries, and for users they offer a similar experience to using traditional libraries. We believe this design provides a much more direct path to adoption, and simulation packages can be built on top of such libraries to get the above benefits.

This is a broader trend in computer science. Ebb and Simit share ideas and structure with languages developed in other fields. Graph processing frameworks for big data (e.g., GraphLab [Low et al. 2010] and Ligra [Shun and Blelloch 2013]) use graph data models, and machine learning frameworks (e.g., Tensorflow [Abadi et al. 2016] and Torch [Collobert et al. 2016]) provide multidimensional arrays and mathematical operators. Perhaps most interestingly, these languages, as well as Simit and Ebb, draw on a rich history of

Authors' addresses: G. L. Bernstein, Gates Computer Science Building, 353 Serra Mall, Stanford, CA 94305; email: gilbert@gilbertbernstein.com; F. Kjolstad, MIT CSAIL, 32 Vassar St 32-G778, Cambridge, MA 02139; email: fred@csail.mit.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2016 ACM 0730-0301/2016/05-ART20e $15.00
DOI: http://dx.doi.org/10.1145/2930661

data-parallel programming exemplified by SQL and the relational algebra in databases [Codd 1970].

Ebb and Simit

Ebb and Simit are both inspired by the database literature, representing mesh data as interconnected sets, operated on by set-at-a-time operators (e.g., Ebb's kernels and Simit's assembly and global linear algebra). This captures the abundant data parallelism in simulation programs. Furthermore, the use of set relationships, interpreted as relational keys or edge sets, exposes locality in the simulation data that can be used to efficiently operate on it in parallel. While the two languages differ on the manifestation of these ideas, we are in strong agreement about the value of these abstractions.

Besides data abstractions, the two systems make a number of similar implementation choices. Both demonstrate performance portability with initial CPU and GPU implementations, encode irregular sparse matrices using the block compressed row storage format, and achieve competitive performance to hand-tuned code despite not yet having put much effort into performance engineering (e.g., no vectorization and no multithreading). Further, both systems are packaged as libraries with online compilers, allowing them to be mixed freely with other libraries and code (e.g., collision detection libraries). In addition, both languages provide support for calling out to external libraries written in C/C++ during execution (e.g., external solvers). As a result, both systems can be readily incorporated into existing simulation software, such as libraries used in game development, special effects, and engineering.

However, Ebb and Simit exploit different points in the design space of simulation languages. This results in simulations written in the two languages adopting different architectures and different divisions of responsibility between programmers and the compiler. On the one hand, Ebb is focused on the description of a wide variety of data structures out of primitives with high-performance guarantees. This leads to a user programmable geometric domain library layer, which does not exist in Simit. On the other hand, Simit is focused on the local/global distinction between local assembly kernels and high-performance global linear algebra. This leads to a global linear algebra language, which does not exist in Ebb.

These distinctions create different optimization opportunities and stories for Ebb and Simit. Simit's global linear algebra model immediately presents opportunities for a compiler to fuse, reorder and otherwise automate the optimization of linear algebra. While similar automatic transformations may be possible in Ebb, they will be challenging to discover and reason about in the kernel language. Ebb's data modeling primitives let programmers carefully control and optimize data representations inside domain libraries. While Simit may be able to expose similar controls to programmers, it will be challenging to ensure that such controls interact well with Simit's linear algebra representation and optimization.

Conclusion

The programming models presented in these articles are two early steps that explore some of what we believe are the most promising directions for future simulation application development. They are far from the last word on the subject. And with increasing complexity and diversity in parallel hardware, we expect the need for similar or complementary linguistic abstractions to only increase.

This vision of languages for writing simulations requires more tools, support for a broader range of computational patterns, optimizations and parallel machines. More research is needed:

(1) As more computation moves into low-cost data centers, the ability to cheaply port and scale simulations to distributed machines will become more important.
(2) Global operations like linear algebra expose new opportunities for code optimization.
(3) Abstracting data storage exposes new opportunities and choices for whole program layout optimization.
(4) New linguistic and computational strategies are needed to portably express, parallelize, and execute costly subcomputations like adaptive remeshing, collision detection, and solvers, or else these quickly become bottlenecks.
(5) Simulation languages open up new opportunities for transparent fault tolerance schemes based on job replication or recomputation.
(6) Simulation languages create the opportunity to diagnose and fix performance and correctness problems in new and powerful ways.
(7) High-performance simulation libraries/systems are frequently forced to trade off between good software engineering and performance; new languages for writing these systems present opportunities to revisit these trade-offs.

With these challenges addressed, we envision the adoption of new programming languages that enable programmers to create efficient, portable, and modular multiphysics simulations.

REFERENCES

Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, and others. 2016. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary DeVito, Matthew Fisher, Philip Levis, and Pat Hanrahan. 2016. Ebb: A DSL for physical simluation on CPUs and GPUs. *ACM Trans. Graph.* 35, 2, Article 21 (April 2016), 21:1–21:12. http://doi.acm.org/10.1145/2892632.

Edgar F. Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377–387.

Ronan Collobert, Clement Farabet, Koray Kavukcuoglu, Soumith Chintala, and others. 2016. torch: A Scientific Computing Framework for LUAJIT. Retrieved from http://torch.ch.

CUDA Nvidia. 2008. Programming guide. (2008).

James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.* 33, Article 144 (July 2014), 144:1–144:11. http://doi.acm.org/10.1145/2601097.2601174.

Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. 2016. Simit: A language for physical simulation. *ACM Trans. Graph.* 35, 2, Article 20 (April 2016), 20:1–20:21. http://doi.acm.org/10.1145/2866569.

Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A new parallel framework for machine learning. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*.

MATLAB. 2014. *Version 8.3.0 (R2014a)*. The MathWorks Inc., Natick, Massachusetts.

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. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.

Julian Shun and Guy E. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. ACM, New York, NY, 135–146. DOI:http://dx.doi.org/10.1145/2442516.2442530

Steve Upstill. 1989. *RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley Longman Publishing Co.

Received April 2016; accepted April 2016