Transformational Programming for time- and frequency-domain EM simulation

Andreas Klöckner[1]

Courant Institute of Mathematical Sciences, 251 Mercer St, New York, NYU kloeckner@cims.nyu.edu

Summary. Detailed, high-fidelity electromagnetic simulations entail a significant computational cost, and this cost may be managed by efficient use of modern computational resources. Modern many-core architectures pose a challenge by being both more diverse and more complicated than conventional computers. This contribution presents strategies and software packages based on run-time code generation that help deal with this emerging complexity. We demonstrate their use and effectiveness by applications to discontinuous Galerkin time-domain and integral-equation-based frequency-domain EM simulations.

1 Introduction

Graphics processing units (GPUs) and many-core machines have enjoyed tremendous impact in recent years, because their use has brought about significant cost reductions for a number of numerical methods. Yet, gains from the use of many-core machines have not been uniformly distributed across methods. Further, adoption of these machines, despite their advantages, has been far from universal. These two facts hint at underlying issues that must be resolved before the promise of these recent hardware advances is realized.

The first issue is that the choice of computational method has thus far often been made in complete ignorance of machine concerns. Examples of this are methods that may satisfy some theoretical optimality criterion, but which are outrun in practice by methods that make some concessions to the hardware and are slightly suboptimal in theory. The consequence of this is that computational methods and their implementation have merged into one joint design space that cannot easily be split into separate concerns.

The second issue is that this new hardware requires specialist knowledge to program. Not only must the programmer be aware of the sometimes intricate semantics of parallel programming models–she must also understand the performance implications of each of the (often many) semantically equivalent ways of expressing a single computation. And even if the programmer possesses some intuition on hardware response to different coding techniques, any given computational task may still require trying numerous approaches to achieve good machine utilization. Worse, this procedure has some likelihood of needing to be repeated when new generations of the same hardware, or *especially* when a different vendor's hardware is to be used. All this translates to extra cost, leading many potential users to forgo the potential execution time gains of many-core implementation.

2 Transformational programming

Unlocking the benefit of GPUs for a majority of users is a thorny problem, to which many solutions have been proposed–too many to even begin to provide a concise overview in this setting. In 2009, we pioneered one very simple starting strategy that has enjoyed a measure of success in the marketplace, in the form of our packages *PyCUDA* and *PyOpenCL*, whose use will be briefly discussed. A cornerstone of this strategy was to enable run-time code generation ('RTCG') [1]. RTCG allows the user to apply more intelligence than customarily supplied by compilers to process and reason about the source code that carries out a desired computation. The basic flow of information in this setting is illustrated in Fig. 1. In other words, our tool provided, in some sense, the smallest possible stepping stone for the creation of additional tools.

Our current work reapplies this recipe of *creating the smallest possible tool* at the next higher level of abstraction, within the field of code generation. We start from the assumption that a computational task is given as a mathematical statement in index-based form, such as

```
c[i,j] = sum(k, a[i,k]*b[k,j]).
```

Further, a set of bounds on the loop variables (`i`, `j`, and `k` in this case) is given as an intersection of affine constraints, in the notation of the *isl* integer set library [4]:

```
[n] -> {[i,j,k]: 0<=i,j,k<n},
```

where we note that `n`, the matrix size, enters as a run-time-variable parameter. Starting from this mathematical statement of the desired operation (along with declarations specifying data storage formats and types), the user may then issue *transformations* that make the generated code more suitable for a certain piece of target hardware by better respecting granularities such as machine vector widths and appropriate sizing of prefetch buffers. Importantly, each

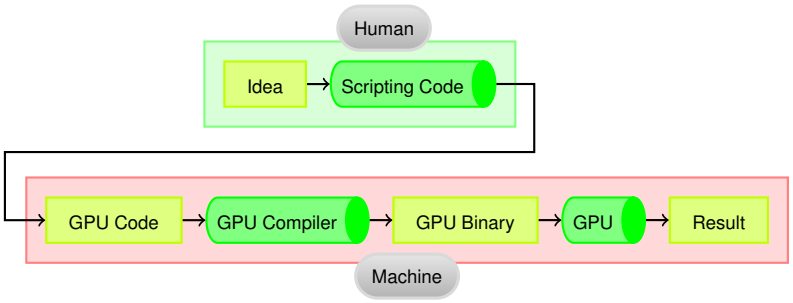

Fig. 1. Operating principle of GPU run-time code generation.

such transformation is guaranteed to preserve the operational semantics meaning of the original, untransformed kernel–but, given the right transformations, may execute far faster on a given piece of hardware. Available transformations include strip-mining, loop unrolling, parallelization, prefetching, cache management, and many more.

Like our previous tools, this code generator, called '*loo.py*', cannot solve *all* problems encountered in making GPU programming accessible. Nonetheless, we claim that what it provides is useful:

- *Increase the trial rate of an expert programmer.* Manually carrying out the transformations allowed by the tool is a tedious, error-prone task. Finding and correcting these errors takes time that can be put to better use.
- *Provide a stepping stone on which more tools can be built.* Loo.py is deterministic and does not attempt to guess or be intelligent on the user's behalf. Tools with such intelligence can be built *on top* of loo.py with relative ease.
- *Facilitate performance portability.* Since loo.py clearly distinguishes the description of the desired computation from the transformations achieving hardware specialization, this latter part may be changed or adapted to new hardware *without* having to revisit the basic computational goal.
- *Channel thought through language design.* The tool enforces a clear separation between mathematical and implementation concerns, even if both influence each other in a conceptually more abstract design space.

In proposing this tool, we have built upon experience gained from earlier work [2] on the type of transformations necessary in GPU programming. In the next section, we discuss how loo.py conceptually and factually supersedes this research.

While loo.py bears some similarity to prior efforts in transformational programming (e.g. CUDA-CHiLL [3]), it is novel because, first, it is *not* a source-to-source translator, but instead views transformations as first-class objects in its language, and second, it integrates into an existing ecosystem of GPU scripting tools centered on PyOpenCL.

3 Evaluation and Conclusions

We evaluate our tool by applying it to time-domain EM simulations using discontinuous Galerkin methods as well as to singular quadrature tasks originating from electromagnetic problems in the frequency-domain, demonstrating the effectiveness of the language exposed along with its applicability to real-world tasks in electromagnetic simulation. We further show performance results supporting the notion that high-performance codes on a broad variety of hardware can be reached by the provided transformations.

In providing loo.py, we hope to build a bridge between computer science innovation in tool building, and application scientist needs. We hope that the tool may provide a basis for innovation and discussion in methods for producing both prototype- and production-grade EM solvers with the least possible effort.

Acknowledgement. The author's research was partially funded by AFOSR under contract number FA9550-07-1-0422, through the AFOSR/NSSEFF Program Award FA9550-10-1-0180 and also under contract DEFG0288ER25053 by the Department of Energy. Loo.py is joint work with Tim Warburton. Time-domain EM is joint work with Jan Hesthaven and Tim Warburton. Frequency-domain EM is joint work with Leslie Greengard and his group.

References

1. A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.
2. A. Klöckner, T. Warburton, J. Bridge, and J.S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *J. Comp. Phys.*, 228:7863–7882, 2009.
3. G. Rudy, M. Khan, M. Hall, C. Chen, and J. Chame. A programming language interface to describe transformations and code generation. *Languages and Compilers for Parallel Computing*, pages 136—-150, 2011.
4. S. Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software – ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer Berlin / Heidelberg, 2010.