NPS-NRL-Rice-UIUC Collaboration on Navy Atmosphere-Ocean Coupled Models on Many-Core Computer Architectures Annual Report

Lucas C. Wilcox
Department of Applied Mathematics
Naval Postgraduate School
833 Dyer Road; Monterey, CA 93943 USA
phone: (831) 656-3249 fax: (831) 656-2355 e-mail: lwilcox@nps.edu

Francis X. Giraldo
Department of Applied Mathematics
Naval Postgraduate School
833 Dyer Road; Monterey, CA 93943 USA
phone: (831) 656-2293 fax: (831) 656-2355 e-mail: fxgirald@nps.edu

Timothy Campbell
Naval Research Laboratory
Oceanography Division, Code 7322; Stennis Space Center, MS 39529 USA
phone: (228) 688-5284 e-mail: tim.campbell@nrlssc.navy.mil

Andreas Klöckner
Department of Computer Science
University of Illinois at Urbana-Champaign
201 North Goodwin Avenue; Urbana, IL 61801 USA
phone: (217) 244-6401 e-mail: andreask@illinois.edu

Timothy Warburton
Department of Computational and Applied Mathematics
Rice University
6100 Main Street, MS 134; Houston, TX 77005 USA
phone: (713) 348-5666 e-mail: tim.warburton@rice.edu

Timothy Whitcomb
Naval Research Laboratory
7 Grace Hopper Ave, MS2; Monterey, CA 93943 USA
phone: (831) 656-4812 fax: (831) 656-4769 email: tim.whitcomb@nrlmry.navy.mil

Award Number: N0001414WX20292; N0001414WX20221; N000141410117; N000141310873

LONG-TERM GOALS

The ever increasing pace of improvement in the state-of-the-art high performance computing technology promises enhanced capabilities for the next-generation atmospheric models. In this project we primarily consider incorporating state of the art numerical methods and algorithms to enable the Nonhydrostatic Unified Model of the Atmosphere (http://faculty.nps.edu/fxgirald/projects/NUMA), also known as NUMA, to fully exploit the current and future generations of parallel many-core computers. This includes sharing the tools developed for NUMA (through open-source) with the U.S. community that can synergistically move the knowledge of accelerator-based computing to many of the climate, weather, and ocean laboratories around the country.

OBJECTIVES

The objective of this project is fourfold. The *first* objective is to identify the bottlenecks of the NUMA and then circumvent these bottlenecks through the use of: (1) analytical tools to identify the most computationally intensive parts of both the dynamics and physics; (2) intelligent and performance portable use of heterogeneous accelerator-based many-core machines, such as General Purpose Graphics Processing Units (GPGPU or GPU, for short) or Intel's Many Integrated Core (MIC), for the dynamics; and (3) intelligent use of accelerators for the physics. The *second* objective is to share and extend the tools we use to develop code that is portable across threading APIs (achieved by using OCCA2 https://github.com/tcew/OCCA2) and has portable performance (by using Loop.py https://github.com/inducer/loopy). The features we are adding to OCCA2 and Loo.py under this project are targeted first to help our many-core acceleration of NUMA effort but should be generally applicable to many scientific computing codes from the general climate, weather, and ocean laboratories around the country. The *third* objective is to implement Earth System Modeling Framework (ESMF) interfaces for the accelerator-based NUMA mini-apps allowing the study of coupling many-core based components. We will investigate whether the ESMF data structures can be used to streamline the coupling of models in light of these new computer architectures which require memory access that has to be carefully orchestrated to maximize both cache hits and bus occupancy for out of cache requests. The *fourth* objective is to implement NEPTUNE (Navy Environmental Prediction sysTem Using the NUMA corE) as an ESMF component allowing NEPTUNE to be used as an atmospheric component in a coupled earth system application. A specific outcome of this objective will be a demonstration of a coupled air-ocean-wave-ice system involving NEPTUNE (with NUMA as its dynamical core), HYCOM, Wavewatch III, and CICE within the Navy ESPC. The understanding gained through this investigation will have a direct impact on the Navy ESPC that is currently under development.

APPROACH

Following the lead of various DoE labs (Swaminarayan 2011), we are adapting NUMA to accelerator-based many-core machines in a step-by-step process to achieve our first objective. At each step we are developing *mini-apps* which are self-contained programs that capture the essential performance characteristics of different algorithms in NUMA. This plan to partition the development of the heterogeneous architecture version of NUMA into small chunks of work that

can be handled somewhat independently will allow us to produce (at every stage of the work pipeline) a result that is beneficial not only to the NUMA developers and user groups but also to the larger climate, weather, and ocean community. The many-core mini-apps that will be develop will include:

Dynamics

Explicit-in-time CG a continuous Galerkin discretization of the compressible Euler mini-app with explicit time integration;

Explicit-in-time DG a discontinuous Galerkin discretization of the compressible Euler mini-app with explicit time integration;

Vertically Semi-Implicit CG a continuous Galerkin discretization of the compressible Euler mini-app with vertically implicit semi-implicit time integration;

Vertically Semi-Implicit DG a discontinuous Galerkin discretization of the compressible Euler mini-app with vertically implicit semi-implicit time integration;

Physics

Moisture a parameterized moisture mini-app; and

Radiation we intend to use the GPU implementation of radiation physics from the ESPC-RRTMGP group.

Once the performance of a mini-app is accepted it will be considered for adoption into NUMA. Placing the kernels back into NUMA will be lead by Giraldo and his postdoctoral researcher Abdi. We will also make these mini-apps available to the community to be imported into other codes if desired. Wilcox is working closely with Warburton and his team to lead the effort to develop the mini-apps including hand rolled computational kernels optimized for GPU accelerators. These kernels are written "hand-written" in OCCA2, a library Warburton's group is developing that allows a single kernel to be compile using many different threading frameworks, such as CUDA, OpenCL, OpenMP, and Pthreads. We are initially developing hand-written kernels to provide a performance target for the Loo.py generated kernels. Parallel communication between computational nodes will use the MPI standard to enable the mini-apps to run on large scale clusters. Using these community standards for parallel programing will allow our mini-apps to be portable to many platforms, however the performance may not be portable across devices. For performance portability, we, lead by Klöckner, are using Loo.py to generate OCCA2 kernels which can be automatically tuned for current many-core devices along with future ones.

The second objective is to expand (as needed by the mini-apps) the OCCA2 and Loo.py efforts, lead by Warburton and Klöckner. These will be extended naturally in tandem with the requirements that emerge during the development of the mini-apps. We will take a pragmatic approach where features will only be added as they are needed by the mini-apps or will aid in the transition of the mini-app technology back into NUMA. For the sharing part of the objective, both OCCA2 and Loo.py are open source and can be downloaded from https://github.com/tcew/OCCA2 and https://github.com/inducer/loopy, respectively. They are operational and are ready to be evaluated for use in other projects. As it warrants, we will give presentations and demonstrations of the tools to help increase their adaption.

The third objective, lead by Campbell and Wilcox, is to implement Earth System Modeling Framework (ESMF) interfaces for the accelerator-based computational kernels of NUMA allowing the study of coupling many-core based components. Once we have working mini-apps implementing a significant set of physics, we will coordinate with the "An Integration and Evaluation Framework for within the proposed ESPC Coupling Testbed" if it is deemed useful to have the mini-app to help test coupling of many-core components.

The forth goal, lead by Campbell, is to implement NEPTUNE as an ESMF component allowing NEPTUNE. This will be done in collaboration with the developers of NEPTUNE at NRL Monterey. Once NEPTUNE has been made a componet, we will move to running the a coupled air-ocean-wave-ice system involving NEPTUNE (with NUMA as its dynamical core), HYCOM, Wavewatch III, and CICE within the Navy ESPC.

WORK COMPLETED

In the course of this project (the first three years plus the two year option), we plan to carry out the following work items:

1. identify current bottlenecks in the NUMA modeling system;
2. port the explicit time-integration portion of the dynamics onto many-core devices;
3. port the moisture schemes onto many-core devices;
4. port the implicit-in-the-vertical dynamics onto many-core devices;
5. port long-wave radiation and other costly physics onto many-core devices;
6. transition many-core kernels into NUMA;
7. adapt the Loo.py code generator for the needs of the project;
8. develop a source-to-source translation capability built on Loo.py to facilitate the NUMA transition;
9. implement ESMF interfaces for many-core components;
10. implement NEPTUNE as an ESMF component;
11. assess performance against current modeling suite;
12. foster the formation of a community working group on using accelerators for atmosphere-ocean modeling.

The management plan for these work items is shown in table 1. Below is a summary on the progress we have obtained on these work items.

Identifying bottlenecks of NUMA In order to facilitate the construction of compute-kernels for use on many-core devices, some of the original data structures of NUMA had to be modified. Since this is a significant change to the code base, we decided to do this before doing a performance analysis to determine the bottlenecks of NUMA. For example, in the CG code the state vector was originally stored as $q(1:N_{var}, 1:N_{poin})$ where N_{var} are the number of variables per gridpoint (e.g.,

Activity Months:	0–12	13–24	25–36	37–48	49–60
Identifying bottlenecks	▷				
Explicit dynamics	▷	•			
Moisture	▷	•			
Vertically-implicit dynamics		▷	•	•	
More physics processes				•	•
Transition many-core kernels into NUMA		▷	•	•	•
Adapt Loo.py	▷	•	•	•	
Develop source translation tool		•	•	•	•
Implement ESMF in mini-apps			•	•	•
Implement ESMF in NEPTUNE	▷	•			
Assess Performance	▷	•	•	•	•
Create Working Group	•				
Disseminate work (Publications)		▷	•	•	•

Table 1: Time-line of proposed activities for different months of the project. The triangle symbol indicates activities that have started.

5 for dry dynamics and 8 including water vapor, condensation, and rain) and N_{poin} are the number of gridpoints. This we denote as a *Global Grid Point* (GGP) storage. On the other hand, in the DG code the state vector is typically stored as $q(1:N_{var}, 1:N_{pts}, 1:N_{elem})$ where N_{pts} are the number of points inside each element and N_{elem} are the total number of elements. We denote the DG storage as *Local Element-Wise* (LEW) storage. Rather than carrying two explicitly different ways of storing the data, we decided on the GGP storage with the inclusion of special pointers that allows us to effectively use either storage. GGP storage will allow us to perform computations efficiently on the compute-devices and, perhaps equally importantly, will allow the GPU version of NUMA to be based on the same data structures of the NUMA version inside of the Naval Research Laboratory's NEPTUNE system. Another advantage of using only one datastructure for the state vector is that we are now able to include both the CG and DG versions of NUMA within the same code, including the original NUMA code which uses the GGP storage (the NUMA version inside NEPTUNE). To make sure that the fidelity of NUMA was maintained we needed to confirm that both the GGP‘ and LEW storage versions of NUMA gave the same results. The main challenge in the construction of a parallel CG code is in the construction of the *direct stiffness summation* (DSS) operator which enforces the C^0 condition of the solution at all element boundaries. A new DSS operator had to be written and this new DSS operator had to be tested to ensure fidelity with the previous results. At this stage, we have a CG version of NUMA that can use either GGP or LEW storage. The next step is to modify NUMA to also handle DG but this will be relatively straightforward since DG uses the LEW storage that has now been added to NUMA. Another advantage of adding the LEW storage (for both CG and DG) is that NUMA is now capable of handling non-conforming grids which is essential for use in efficient adaptive mesh refinement (AMR). The addition of AMR in three-dimensions is work in progress.

Various other optimization strategies for NUMA have been implemented including efficient construction of the initial grids. In addition, the current domain decomposition strategy that uses the METIS software is being reconsidered. We began exploring other techniques for constructing the graph-partitioning via the use of space-filling curves. If an octree approach to grid generation

is used, the graph-partitioning approach is straightforward. However, we seek a general graph-partitioning based on space-filling curves for general unstructured grids since it may be necessary to construct initial grids off-line (via some other software) and then read into NUMA.

Explicit Dynamics We have begun the development of the explicit dynamics mini-app. To start, we implemented non-linear Euler flow solvers including two-additional tracer fields for moisture in two discretization variants: continuous and discontinuous Galerkin spatial discretizations both using explicit Runge–Kutta time integration. All of the computationally intensive parts of this baseline solver have been implemented with "hand-written" OCCA2 kernels. We verified both the continuous Galerkin and discontinuous Galerkin versions of the solver against a 3D extruded version of the isentropic vortex benchmark given in Zhou and Wei (2003) (results of this verification can be found in figure 1). So far we are satisfied that we have a correct implementation of the methods used in NUMA to discretize the compressible Euler equations.

Once the code was verified, we profiled the program and started to address the bottlenecks of the code (current performance profiles of the code can be found in the RESULTS section of this report). For these "hand-written" computational kernels, this was a tedious process of trying different strategies for memory layout and computation organization. Here we give a couple examples of changes we made that improved the performance of the explicit dynamics mini-app. We ended up splitting the volume integration evaluation from one kernel into two kernels (one for the horizontal and one for the vertical derivatives of the reference element). This change in computational organization reduced the register pressure of the kernels, gave the volume integration a $6\times$ speed improvement. We also changed the memory layout of the fields processed in the computational kernels from one field after the other to an interleaved version so all the fields for one grid point are grouped together. This allowed us to use vectorized loads of data in the DSS operation and the numerical flux providing a more regular memory access. This change drastically reduced the time spent in the DSS operation.

We next started implementing stabilization required for practical weather application. This required the implementation of a variable viscosity diffusion operator. We started by implementing a continuous Galerkin discretization of the variable viscosity diffusion operator. To isolate the diffusion operator, this was verified by solving a forced heat equation where the solution was generated by the method of manufactured solutions (results of this verification are given in figure 2). Again once the code was verified, we profiled the mini-app and looked for kernels that can be improved. The memory layout change described above also helped to simplify the diffusion kernels, which where overly complicated due to the nonuniform memory access of some of the derivatives. We are currently in the process of developing the discontinuous Galerkin version of the diffusion kernels.

The remaining items we would like to add to the explicit mini-app model are: (1) the use of perturbed variables; (2) different boundary conditions; and (3) a moisture model as detailed below. Along with this we are going to use the MPI (Message Passing Interface) API to parallelize the mini-app across multiple many-core processors.

Now that we have the basic structure for the computation in the explicit mini-app developed, we can start generating computational kernels with Loo.py and benchmark them against our "hand written" kernels. This will be done with the goal being to create performance portable kernels that

can be semi-automatically tuned to new architectures.

Vertically-Implicit Dynamics From March 14 to April 1, Rajesh Gandham (a graduate student in Warburton's research group at Rice University) visited NPS. Part of his visit focused on where his expertise can be used in this project. He has developed an algebraic multigrid solver (that runs on the CPU and GPU using OCCA) that we are considering using for the linear solves when doing implicit time-stepping. Specific he tested his algebraic multigrid (AMG) solver as a preconditioner on a reference linear system from NUMA's Schur complement form of the CG discretization with implicit time-stepping. The number of iterations needed were lower than currently required in NUMA, now the question becomes is it more efficient? Once the implicit time-stepping mini-app is implemented for many-core we can systematically address this question.

Moisture In our explicit dynamics mini-app we have designed the kernels for 2–3 moisture variables to include warm moist processes in the model. We are now determining which moisture parameterization to use. We are also considering using the warm air moist processes described in Satoh (2003). To verify the implementation we are going to start with the moist bubble testcase from Duarte et al. (2013).

Transition many-core kernels into NUMA A two-pronged approach for *GPUfying* NUMA is currently being undertaken. In one approach, so-called mini-apps are being developed. In another approach, standard GPUfying approaches have been implemented within a two-dimensional version of NUMA. The main reason for this second approach is to train our postdoctoral fellow (who arrived 4 months ago) in the implementation of NUMA on accelerators. It also allows us to discover issues that will come up when transitioning the many-core kernels into NUMA. In this second approach, subroutines that take up the majority of the computation time have been identified via profiling of the code. The results of profiling NUMA2D are shown in table 2 where we see that 70% of the computation time is taken up by the evaluation of the volume integrals. Although the profiling shown is for the CG version of the code, one advantage of

Table 2: NUMA2D: Profiling results

% time	subroutine name
69.17	Volume Integrals
13.21	Boundary Conditions
9.49	DSS Operator

a unified CG/DG approach is that this exact same Volume Integral routine is used in either version of the code and so optimizing this routine will impact both the CG and DG results.

This simple profiling experiment allows us to prioritize the routines that need to be GPUfied in order to speed up the code if only to gauge the types of speedups expected (however, to reap the full benefits of accelerators requires porting the entire code onto the compute-device and this is the role of the first approach mentioned above that involves the construction of mini-apps). As a basic test, we have included CUDA and OpenACC versions of the Volume Integral routine. In addition, a work in progress to GPUfy NUMA uses OCCA2 kernels written in C++, that can be

compiled in to either OpenMP, OpenCL or CUDA at runtime. The major advantage of this approach is that NUMA can be run on non-homogeneous computing hardware with optimized code. Performance portability is difficult to achieve on future supercomputers that use various type of accelerators (GPUs, Xeon-Phi, and SIMD etc). All of these experiences will help us inform the development of the Loop.py based source-to-source translation tool that will ease the move of NUMA3D to accelerator based computing.

The results obtained on the GPU and CPU are the same except for minor differences due to truncation errors. However no speed up could be obtained on the GPU because time-integration for the explicit solver was not moved to the compute-devices. Thus variables have to be copied back and forth between the CPU and GPU at each time step. This seems to have impacted performance significantly. This is to be expected since to get good performance requires moving all of the compute-intensive operations to the compute-device without moving data to and from the host. If this is done, then performance is lost which is what this educational exercise confirmed. The next step will be to move all of the operations to the compute-devices. However, this will be done via the introduction of the mini-apps mentioned previously.

Adapt Loo.py Efforts surrounding inLoo.py in this project took a three-pronged approach. First of all, the tool itself was tested and matured further, incorporating new features and improving the usability of existing ones. This general tool maturity will benefit *all* aspects of Loo.py usage as the project progresses. Second, a syntax was investigated that would, in its final state, allow nearly unmodified Fortran kernels to be ingested and transformed by Loo.py. Third, Loo.py's current capabilities were evaluated using hand-written accelerator codes that were created as part of other subprojects. In each case, a determination was made whether the optimization strategy used by the hand-written code was accessible to Loo.py, and if that was not the case, an extension to transformation vocabulary was designed that will be brought to bear on this class of compute kernels.

In addition to the above, Loo.py and its documentation were publicly released and presented at an international conference.

Implement ESMF in NEPTUNE On February 19 and 20 Campbell visited NPS to kickoff the NUMA as an ESMF Component part of the project. We discussed the requirements and code implications of making NUMA an ESMF component conforming to the NUOPC layer. Campbell also visited NRL Monterey during this trip where they are working on the NEPTUNE system which is a next-generation unified global-regional prediction system using NUMA as the dynamical core. Initially we had proposed to make NUMA a component but with the development of NEPTUNE, we are shifting our focus to making NEPTUNE a component. This should be more beneficial to the overall ESPC Air-Ocean-Land-Ice Global Coupled Prediction effort.

Assess Performance We are continuously assessing performance of all of the different strategies for running code on many-core processors. The main assessment this year was of the NUMA code itself described above and of the explicit dynamics mini-app detailed in the RESULTS section below.

Create Working Group Although we have not setup a formal working group, Wilcox had discussions with the Optimized Infrastructure for ESPC group related to the details about many-core programming and implementations (Giraldo listened in). We discussed ways of determining resources with various parallel programming APIs. We will continue discussions so that the many-core version of NUMA will be able to be called from within ESPC. We stand ready to help other groups with any discussions that might help with their transition to many-core computing.

Communication with the community To exchange ideas with the broader community of researchers interested in developing high-order accurate simulations, including those for climate, weather, and ocean we organized two minisymposium this year. For the first, Warburton and Wilcox held a minisymposium at the International Conference on Spectral and High Order Methods 2014 (ICOSAHOM http://www.icosahom2014.org/) to bring together senior and junior international researchers from the national labs, academia, and industry who are actively engaged in the development of high performance algorithms for high-order PDE discretizations on many-core architectures. For the second, Giraldo, Navarra, and Tribbia held a minisymposium at ICOSAHOM on "Local High-Order Methods for Climate, Weather, and Ocean" which brought together the major modeling efforts from around the world that are focused on working on local high-order methods such as spectral element and discontinuous Galerkin methods.

RESULTS

Explicit Dynamics We have developed a new many-core time-explicit compressible Euler equations solver mini-app during the last year. This new capability, uses a continuous or discontinuous spatial discretization and an explicit Runge–Kutta temporal discretization mimicking what is used in NUMA. This solver currently uses OCCA2 kernels which allows it to run on various threading back ends, including OpenMP, Pthreads, OpenCL and CUDA. In figure 1 we present verification results of the solver for an isentropic vortex benchmark from Zhou and Wei (2003). We are in the process of adding variable viscosity for stabilization. We have the continuous Galerkin diffusion operator implemented and present verification of the implementation, given in figure 2, by solving the heat equation for a manufactured solution.

In addition to verification tests we have also measured the performance of our explicit dynamics mini-app and present the results for running: (1) with a CUDA back end on an Nvidia GeForce GTX Titan Black (figure 3); (2) with an OpenCL back end on an Nvidia GeForce GTX Titan Black (figure 4); (3) with an OpenCL back end on an AMD 7970 Ghz Edition (figure 5); and (4) with the Intel OpenCL back end on a pair of Intel(R) Xeon(R) CPU E5-2650 v2 processors (figure 4). We will use this as a baseline of performance for our computational kernels. In these plots: "horizontal volume" and "vertical volume" are kernels that compute the volume derivatives on each element, which are shared by CG and DG; "surface" is a kernel that computes the numerical fluxes and boundary conditions in DG; "project" is a kernel that computes the DSS for CG; "boundary" is a kernel that imposes the boundary conditions through a penalty in DG; and "update" is a kernel that performs the explicit RK update for both CG and DG. We have similar performance numbers for the variable viscosity kernels. Most of these "hand written" kernels are towards the outer limit of what is feasible and maintainable. Our goal is to explore a larger set of

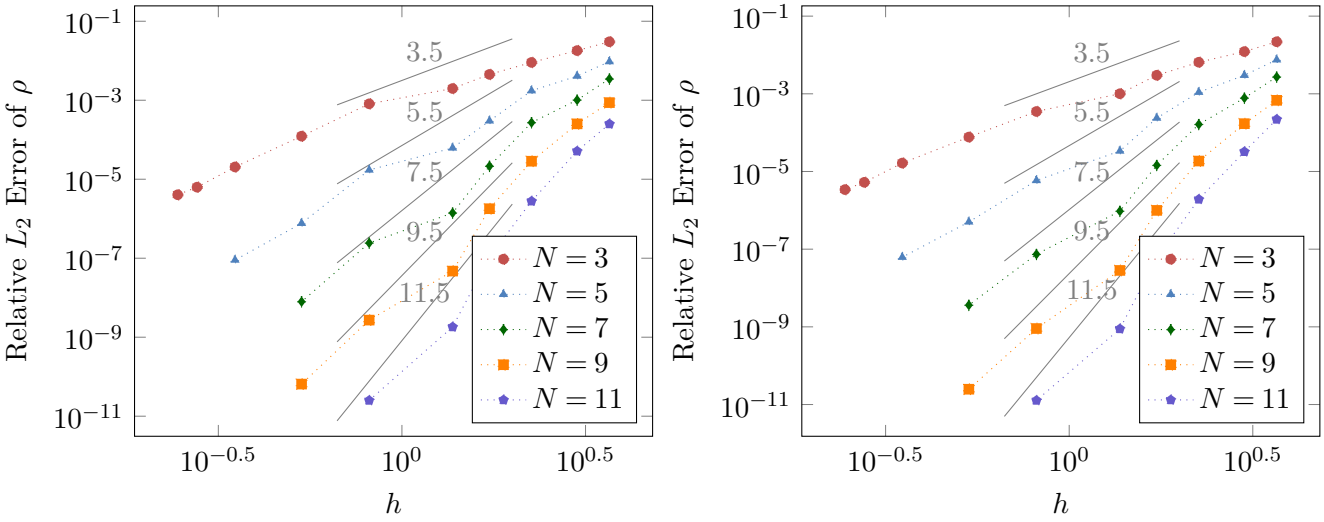

Figure 1: (Left) Continuous Galerkin; (Right) Discontinuous Galerkin: Error plot for a compressible Euler equations isentropic vortex verification test case running in double precision on an AMD 7970.

kernels through composite transforms provided by Loo.py.

Loo.py Loo.py is a code generation component that will be used for automated, multi-target code generation in the later stages of the projects, as results from the 'hand-written' effort are used to guide and focus the development of the code generator. This will eventually lead to a tool that can accept Fortran loops, convert them to Loo.py's internal data model and carry out a rich set of transformations on them, and finally generate code in the form of OpenCL (currently implemented) or OCCA2 (to be implemented) kernel code. Building upon and integrating with OCCA2's Fortran runtime features, this capability will allow seamless integration of Loo.py-transformed kernels into the mini-apps and NUMA.

The core contributions in the approach behind Loo.py are the following: (1) A novel, partially ordered programming language and corresponding internal representation of array-based programs based loosely on the polyhedral model was described. (2) An extensive library of transformations was presented to act upon the internal representation that is able to capture many commonly used tuning strategies. (3) A novel way of assembling heterogeneous computational software is presented. The approach uses a dynamic language for high-level control while interfacing with a run-time code generator for high-performance execution. It builds and improves upon the model of run-time code generation from a scripting language proposed in (A. Klöckner et al. 2012). (4) The data model exposes enough information for a strong run-time interface that provides safe, efficient transitions between host and embedded language, optionally enabling type-generic programming. (5) The ideas above combine to yield good user program maintainability by enforcing strong separation of concerns between computation semantics and performance optimization, easily capturing program variants and allowing optimization reuse.

Klöckner has released Loo.py as a software tool during this reporting cycle, with a web homepage at http://andreask.cs.illinois.edu/software/loopy, a source code repository at

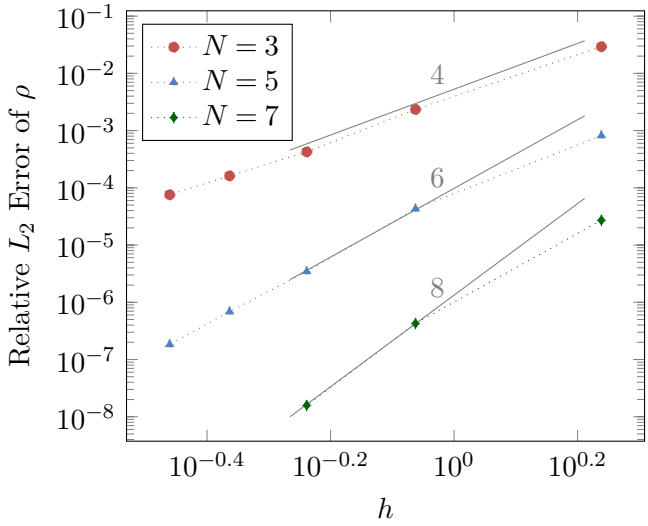

Figure 2: Continuous Galerkin: Error plot of the explicit dynamics mini-app for a heat equation verification benchmark test case in double precision running on an AMD 7970.

https://github.com/inducer/loopy, and a growing set of documentation at http://documen.tician.de/loopy. The release of Loo.py coincided with the presentation of the paper (A. Klöckner 2014), which was accepted to ARRAY'14, the inaugural workshop on Libraries, Languages, and Compilers for Array Programming co-located with the Association of Computing Machinery's Conference on Programming Language Design and Implementation ("PLDI") 2014 in Edinburgh.

This inaugural release along with its first presentation as a software tool (rather than as the 'just' the subject of research) catalyzed a number of important developments in loo.py:

- Syntax and usability improvements. A number of small inefficiencies and 'grown' inelegances came to light as the narrative for the paper was developed. These aspects have been addressed, resulting in a cleaner, neater input language for Loo.py.
- To encourage adoption, Loo.py was made as self-contained as possible. This effort included the elimination of any dependencies of Loo.py that needed a separate compilation step outside of what the Python package manger can handle in an unassisted manner. This results in a faster, more streamlined installation process for members of the project team desiring to use Loo.py.

Along with this process of moving towards release-readiness, a number of further improvements were made:

- Loo.py as it is today is fairly fast at generating and transforming code, making its use in a just-in-time code generation capacity a credible possibility. To enable it to hit even tighter performance requirements, a *caching framework* was put into place that allows allows reuse of previously computed transformation and generation results.
- One of the Loo.py's key benefits over more traditional optimizing compilers is that, since it acts under the direction of the user, it is able to perform transformations that are either

impossible or tremendously hard to carry out for a traditional optimizing compiler. A compiler is required to preserve the semantics of user code, and as such is barred from making non-equivalent changes. Freed from this constraint, one impactful transform that Loo.pyis able to apply is to change the way data is laid out in memory. While often tremendously beneficial for performance, this is not a practical capability until code can also be generated to copy data from the original to the new layout in a semi-automatic fashion. The ability to perform this function was implemented.

To prepare for Loo.py's use in NUMA and gNUMA, a preliminary Fortran frontend was developed that makes it possible to use conventional Fortran kernels as input to Loo.py. An example of this syntax (along with further syntax for applying transformations) is shown below:

```
subroutine fill(out, a, n)
  implicit none

  real*8 a, out(n)
  integer n

  do i = 1, n
    out(i) = a
  end do
  do i = 1, n
    out(i) = out(i) * 2
  end do
end

!$loopy begin transform
!
! fill = lp.split_iname(fill, "i", 128,
! outer_tag="g.0", inner_tag="l.0")
! fill = lp.split_iname(fill, "i_1", 128,
! outer_tag="g.0", inner_tag="l.0")
!$loopy end transform
```

While, by design, Fortran is able to express more constructs than Loo.py, whose input language is not fully general-purpose. In particular, Loo.pyimposes more stringent ordering requirements than Fortran, which is based a conventional sequentially-ordered execution. One consequence is that using a Fortran kernel such as the example above as input to Loo.pyentails a promise by the user that it is safe to do so. The process of making this decision is aided by Loo.pythrough the rejection of input code that would not be safe to convert—and when in doubt, err on the side of caution. Building on this effort, PI Köeckner and PI Wilcox have identified a number of transformations that allow the user to reason about and, optimally, undo optimizations that were previously applied to Fortran code by hand. An experimental version of this Fortran frontend is available at https://github.com/inducer/floopy.

Presenting performance results for a code generator like Loo.py is not very meaningful in general, as the obtained performance hinges on the sequence of transformations specified by the user to a far larger degree than it might for an optimizing compiler. After all, Loo.py is not a compiler, but a code generator. Nonetheless, the argument that good performance is achievable using Loo.py's transformations merits being supported. table 3 summarizes performance results for a variety of workloads across CPUs and GPUs. These performance numbers were obtained by running Loo.py's

		Intel	AMD	Nvidia
`saxpy`	[GBytes/s]	18.6	231.0	232.1
`sgemm`	[GFlops/s]	12.3	492.3	369.4
3D Coulomb pot.	[M Pairs/s]	231	10949	9985
Simplex dG FEM volume	[GFlops/s]	77.4	1251	351
Simplex dG FEM surface	[GFlops/s]	25.9	527	214

Table 3: Performance results for a number of simple performance tests on Loo.py. 'Intel' tests were run on an Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz using the 64-bit Linux Intel OpenCL SDK, build 76921. 'AMD' tests were run on an AMD Radeon HD 7990 using AMD's `fglrx` driver version 14.1beta1 and ICD version 14.3beta1. 'Nvidia' tests were run on an Nvidia GeForce Titan using Nvidia's 64-bit Linux driver and ICD version 331.49. It should be noted that the discontinuous Galerkin FEM kernels shown in this table operate on tetrahedra and have very different operation counts and performance profiles from the ones shown in the remainder of this report. As a result, direct comparison of performance numbers does not yield meaningful results and should be avoided.

test cases against the list of devices specified in the caption of table 3. Since Loo.py's tests are, for now, more focused on correctness than performance, these results should be viewed as a lower bound, in the sense that better performance should be available with rather limited tuning effort.

A careful exploration of how Loo.py's transformation language enables access to performance across a variety of common numerical operations is the subject of a forthcoming article (A. Klöckner and Warburton 2014). While Loo.py is a useful system *today*, a number of extensions are likely to broaden its appeal and increase its usefulness, especially in the context of large, existing Fortran code bases such as the ones being investigated in this project. A core part of the investigation will focus on transformations that help take in, reason about, and transform-to-optimize idiomatic Fortran code.

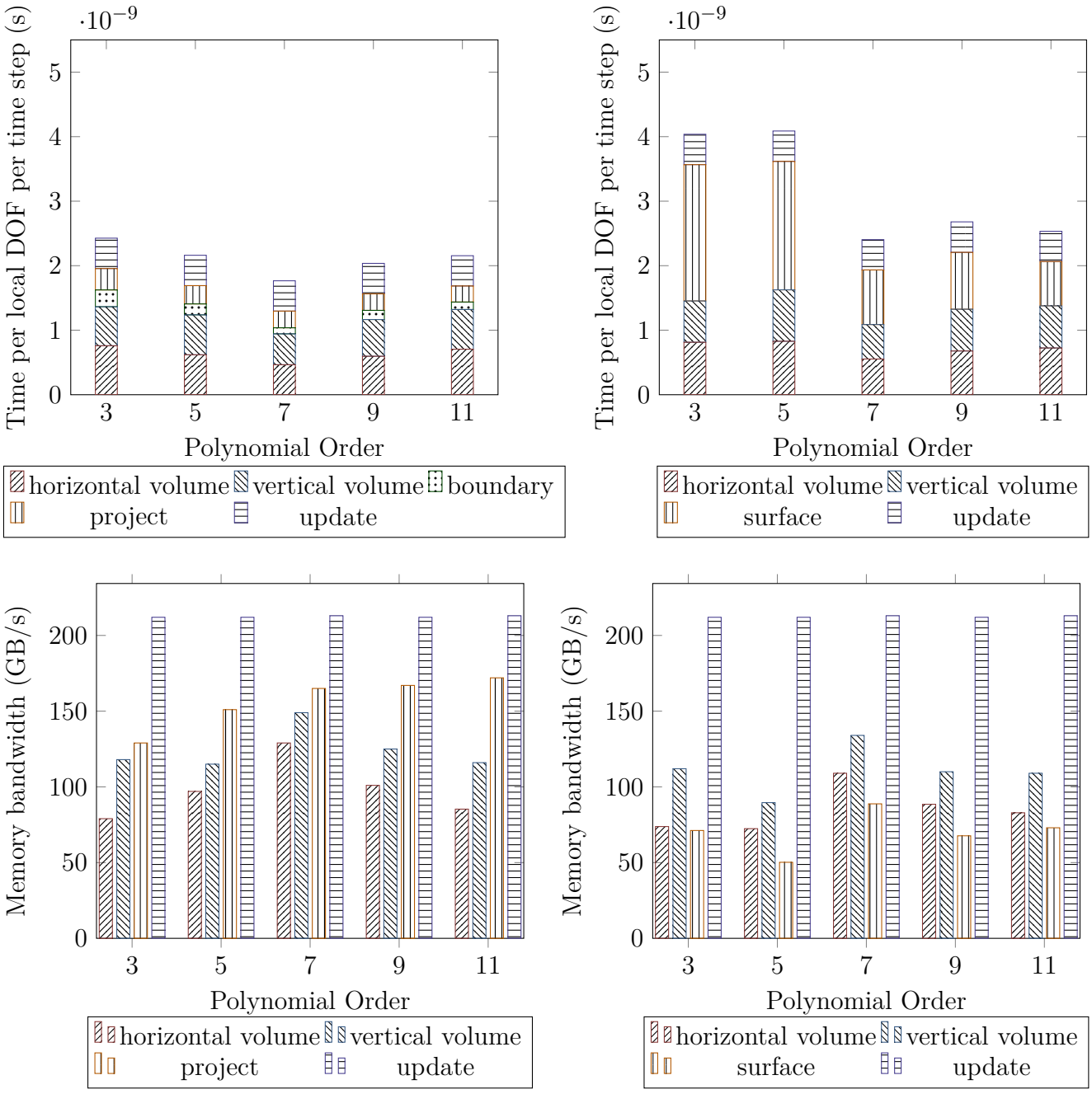

Figure 3: (Left) Continuous Galerkin; (Right) Discontinuous Galerkin: Preliminary performance results for the hand written computational kernels solving compressible Euler equations running in single precision using CUDA (via OCCA2) on a Nvidia Titan Black.

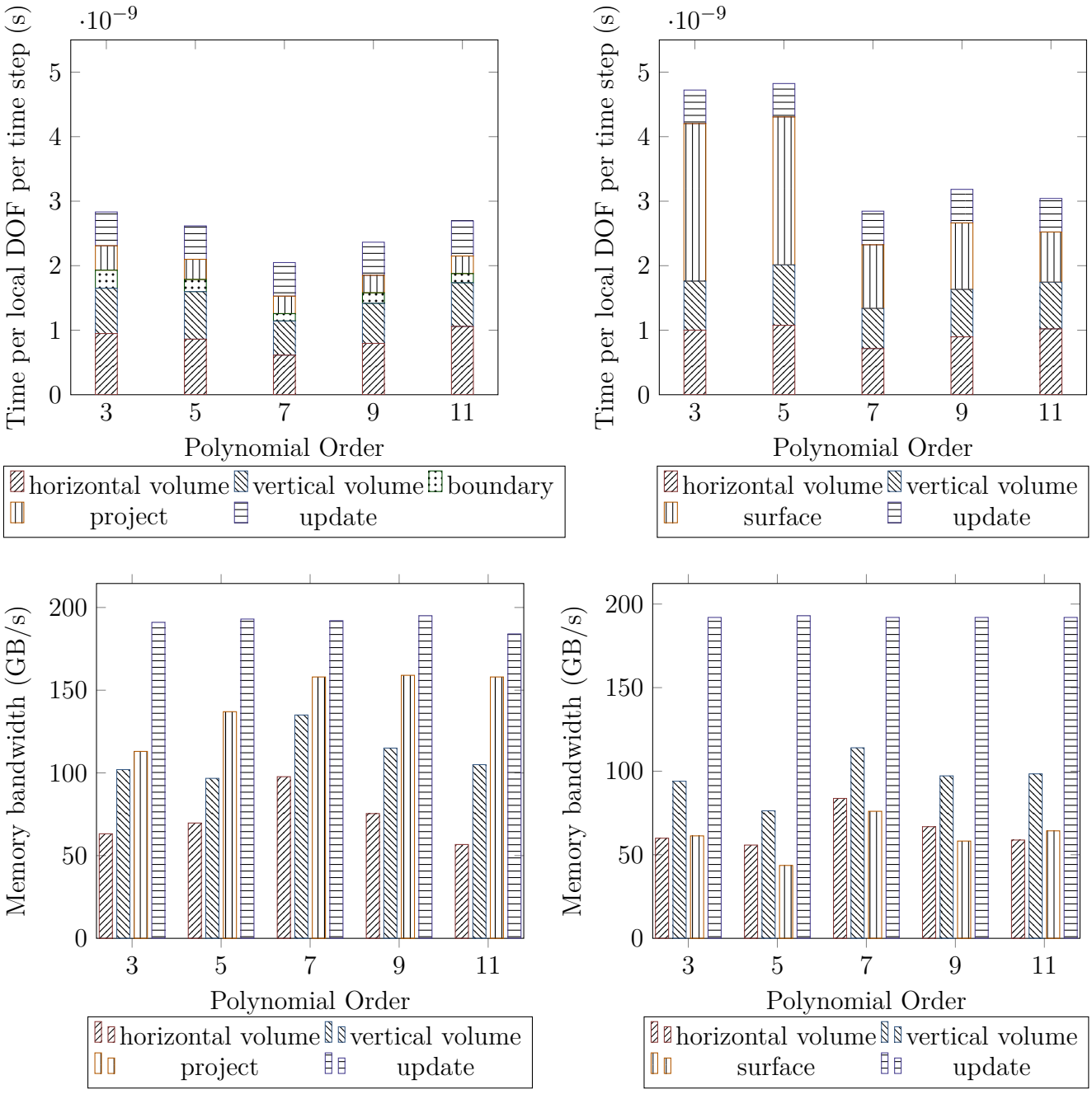

Figure 4: (Left) Continuous Galerkin; (Right) Discontinuous Galerkin: Preliminary performance results for the hand written computational kernels solving compressible Euler equations running in single precision using OpenCL (via OCCA2) on an Nvidia Titan Black.

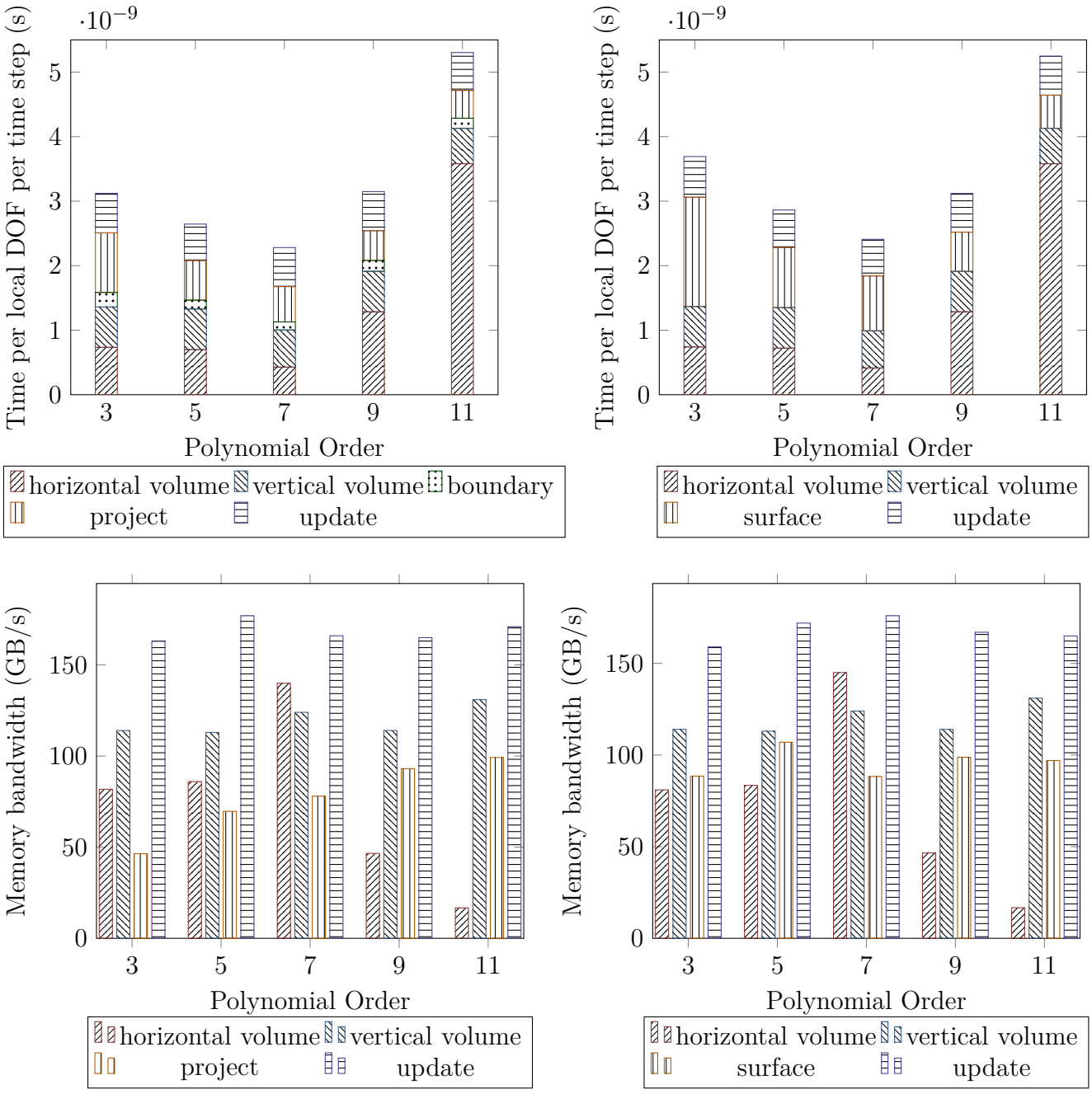

Figure 5: (Left) Continuous Galerkin; (Right) Discontinuous Galerkin: Preliminary performance results for the hand written computational kernels solving compressible Euler equations running in single precision using OpenCL (via OCCA2) on an AMD 7970 Ghz Edition.

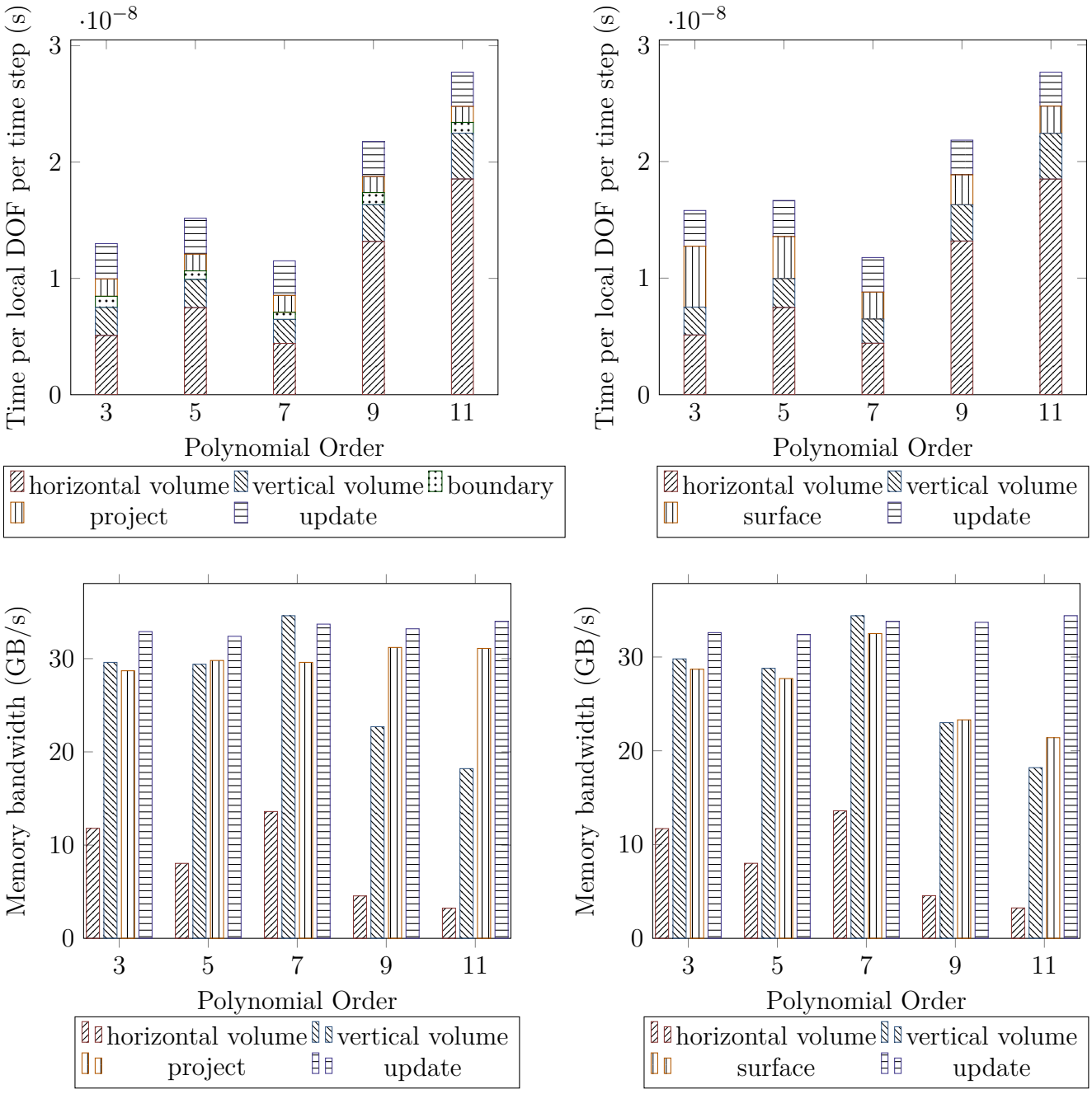

Figure 6: (Left) Continuous Galerkin; (Right) Discontinuous Galerkin: Preliminary performance results for the hand written computational kernels solving compressible Euler equations running in single precision using OpenCL (via OCCA2) on dual Intel Xeon E5-2650 v2 CPUs.

IMPACT/APPLICATIONS

Ensuring that the U.S. gains and maintains a strategic advantage in medium-range weather forecasting requires pooling knowledge from across the disparate U.S. government agencies currently involved in both climate and weather prediction modeling. The new computer architectures currently coming into maturity have leveled the playing field because only those that embrace this technology and fully commit to harnessing its power will be able to push the frontiers of atmosphere-ocean modeling beyond its current state. The work in this project is critical to developing and distributing the knowledge of accelerator-based computing that will support the use of the new platforms in many of the climate, weather, and ocean laboratories around the country.

TRANSITIONS

Improved algorithms for model processes will be transitioned to 6.4 as they are ready, and will ultimately be transitioned to FNMOC.

RELATED PROJECTS

The Earth System Modeling Framework (ESMF) together with the NUOPC Interoperability Layer form the backbone of the Navy ESPC software coupling infrastructure. We will enable the many-core mini-apps and NUMA to be used as components in the Navy ESPC by implementing them as a NUOPC compliant ESMF components. This will bring our work the ESPC community enabling coupling to codes from other projects such as HYCOM and Wavewatch III.

REFERENCES

Duarte, M. et al. (2013). "A Numerical Study of Methods for Moist Atmospheric Flows: Compressible Equations". In: *ArXiv e-prints*. arXiv: 1311.4265 [physics.ao-ph].

Klöckner, A. (2014). "Loo.py: transformation-based code generation for GPUs and CPUs". In: *Proceedings of ARRAY 2014: ACM SIGPLAN Workshop on Libraries, Languages, and Compilers for Array Programming at ACM PLDI 2014*. Edinburgh, UK. URL: http://arxiv.org/abs/1405.7470.

Klöckner, A. and T. Warburton (2014). "Loo.py: applications of transformation-based code generation". In: (in preparation).

Klöckner, Andreas et al. (2012). "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation". In: *Parallel Computing* 38.3, pp. 157–174. DOI: 10.1016/j.parco.2011.09.001.

Satoh, Masaki (2003). "Conservative Scheme for a Compressible Nonhydrostatic Model with Moist Processes". In: *Monthly Weather Review* 131.6, pp. 1033–1050. DOI: 10.1175/1520-0493(2003)131<1033:CSFACN>2.0.CO;2.

Swaminarayan, Sriram (2011). *Exaflops, Petabytes, and Gigathreads... Oh my!* DoE Advanced Scientific Computing Research (ASCR) 2011 Workshop on Exascale Programming Challenges.

URL: http://science.energy.gov/~/media/ascr/pdf/research/cs/Programming_Challenges_Workshop/Siriam%20Swaminarayan.pdf.

Zhou, Y.C. and G.W. Wei (2003). "High resolution conjugate filters for the simulation of flows". In: *Journal of Computational Physics* 189.1, pp. 159–179. DOI: 10.1016/S0021-9991(03)00206-7.

PUBLICATIONS

A. Klöckner (2014). "Loo.py: transformation-based code generation for GPUs and CPUs". In: *Proceedings of ARRAY 2014: ACM SIGPLAN Workshop on Libraries, Languages, and Compilers for Array Programming at ACM PLDI 2014*. Edinburgh, UK. URL: http://arxiv.org/abs/1405.7470