Using Automated Code Generation to Support High Performance Extended MHD Integration in OpenGGCM

Kai Germaschewski and Joachim Raeder
Space Science Center and Department of Physics
University of New Hampshire, Durham, NH 03824

Abstract. Automatic code generation is a technique that takes the specification of an algorithm at a high abstraction level and turns it into a well-tuned computer code. For finite-volume / finite-difference based discretizations, this higher abstraction level can be a stencil computation. At the backend, the code generator features modules which generate optimal code for specific hardware architectures, for example conventional architectures (x86) using SIMD instructions (e.g. SSE2), or heterogeneous architectures like the Cell processor or GPGPUs. The definition of the computation is agnostic to the actual hardware used, as a high-performance implementation tailored to the specific architecture will be generated automatically.

The OpenGGCM code, a global magnetosphere model, has been converted to use an automatically generated implementation of its magnetohydrodynamics (MHD) integrator. The new version enables us to take advantage of the Cell processor's computational capability and also shows performance improvements of up to 2.3× on a conventional Intel processor. The code generation approach also facilitated the recent extension of the MHD model to incorporate Hall physics.

1. Introduction

One of the major barriers for entry for prospective supercomputer users is the difficulty in developing codes to run on increasingly complex architecture. The paradigm shift in scientific computation to heterogeneous architecture at all core counts (Brodtkorb et al. 2010; Crawford et al. 2008) is making this challenge much more pervasive. This drastic change in scientific computing is fueled by the fact that physical limitations in chip design have led to a saturation of single-core performance. Thus performance gains are driven by an increase in parallelism on multiple levels: a large number of distributed nodes, each featuring on or more multi-core processors, which have further parallelism in the form of instruction-level parallelism and single instruction, multiple data (SIMD) processing. Heterogeneous architectures like the STI (Sony, Toshiba, and IBM) Cell processor and GPUs (graphics processing units) hold the promise for the future of scientific computation.

Despite their advantages, heterogeneous architectures introduce significant challenges in constructing programs that are able to take full advantage of their theoretical performance. Achieving high utilization of the available floating point units involves, e.g., avoiding stalls waiting for data and employing SIMD instructions when applicable. In this paper, we present the design and performance of a toolkit for automatic generation of highly efficient code for stencil computations. Using this technology, a scientist can describe a PDE in a symbolic way, while the toolkit takes care of generating code

customized to a particular hardware architecture like the Cell processor or GPUs. As opposed to recent efforts that use domain specific languages (DSLs), e.g., Williams et al. (2008); Christen et al. (2011), here we extend the scripting language Python with the functionality required which facilitates extending functionality without having to redesign a DSL.

We demonstrate the potential of our approach for both achieving improved performance as well as higher productivity using a production science code: the OpenGGCM model.

The OpenGGCM global magnetosphere model

The Open Geospace General Circulation Model (OpenGGCM) is a global model of Earth's space environment based on magnetohydrodynamics (MHD) and is used to simulate the interaction of the solar wind with the magnetosphere, ionosphere, and thermosphere (Raeder et al. 2008). This model has been developed and continually improved over more than 15 years. It is also a community model that can be used for runs on demand at the NASA/GSFC Community Coordinated Modeling Center (CCMC, `http://ccmc.gsfc.nasa.gov`).

Advancing the MHD equations is by far the computationally most expensive part of the model, which is parallelized using domain decomposition and MPI. The legacy model is written in Fortran77 with a custom preprocessor and was typically run on clusters using up to 100s of cores, limiting resolution to $\sim 100 \times 10^6$ grid cells in a non-uniform Cartesian grid. These runs achieve resolutions of $\sim 0.1 R_E$ (Earth radii) near the Earth. However, the typical thickness of boundary layers and current sheets is about 1/10 of that value. Furthermore, as detailed later, including Hall physics in the model also substantially increases computational requirements.

2. Code generation toolkit design

Frontends. The purpose of the code generator's frontends is to provide an interface for application developers to define stencil computations. The code generator itself is a Python module that defines a rather straight-forward DSL on top of Python to specify stencil computations. Hence a user does not need to learn the Python language itself for simple tasks, even though the code he/she is writing is actually Python code. For instance, defining a 2D 5-point Laplacian is simply expressed as:

```
rhs = ((f(-1,0) - 2*f(0,0) + f(1,0)) / (dx**2) +
       (f(0,-1) - 2*f(0,0) + f(0,1)) / (dy**2))
```

While this basic interface was used in converting OpenGGCM to exactly match the existing discretization, the toolkit also provides a high-level interface allowing the user to specify equations closer to their mathematical description. E.g., in the Magnetic Reconnection Code (MRC), the equation $\partial_t \rho = -\nabla \cdot (\rho \mathbf{v})$, where $\mathbf{v} = \mathbf{p}/\rho$, is specified as:

```
V = 1./RHO * P # velocity V, momentum P, density RHO
rhs_RHO = - Divg(RHO * V)
```

This generates code using a finite-volume discretization, calculating fluxes on faces by flux-averaging from the cell centers. The MRC actually uses the ZIP average for the flux calculation, a variation which is specified by "`Divg(ZIP(RHO, V))`".

The frontend has facilities to specify complex stencil computations as they occur in OpenGGCM, including a Harten-Zwas type high-order/low-order switched scheme involving limiters which requires min/max and conditionals. Expressing the numerics as stencil computations makes it easier to change the underlying numerical scheme and physics-based set of PDEs, as it is now concisely expressed and separate from its implementation. The definition of the scheme takes up about 350 lines, while the original Fortran implementation required more than 2100 lines. This can enable major productivity increases – as shown later, implementing the Hall term was essentially done in a mere 8 lines of added code.

Backends. Backends are responsible for taking the abstract description and turning it into code for a particular architecture. Interoperability between existing applications and generated kernels is of major importance – the toolkit can generate implementations which are pluggable into existing legacy applications or kernels designed to interoperate with libraries like PETSc (Balay et al. 2004). The toolkit currently has two completed backends: conventional x86 processors with SSE2/SSSE3 SIMD extensions and STI Cell processors. Work on a GPGPU backend generating CUDA code is in progress.

Backends apply a sequence of transforms to the stencil computation that results in hardware-tailored code. Transforms can be shared between backends, e.g., the SSE2 and Cell backend share a conversion to SIMD: Both processor architectures provide SIMD instructions that enable computations on 4 (single precision) floats at a time. Auto-vectorizing compilers serve the same purpose, however in the case of the Cell processor, in our experience IBM's xlf and gfortran were missing many opportunities in the original code. Other transforms provide adaptation to a given data layout, e.g., array-of-struct vs struct-of-array vs array-of-struct-of-simd-vectors.

The Cell backend involves transforms that will subdivide the domain into smaller workblocks which get load-balanced to the SPEs (synergistic processing elements). SPEs stream through their assigned workblocks, loading and storing data by double-buffered DMA. A related transform can be employed on conventional processors: blocking subdivides the domain into cache-friendly blocks, and then loops over the blocks, which can provide significant performance gains (Kamil et al. 2006).

The transforms needed for efficient use of the Cell processor have been described in more detail in Germaschewski et al. (2008). The SPEs can only directly access a small amount of local memory (called "local store"), which is far too small to hold 3D MHD data. The main memory has to be accessed by explicit load and store DMA instructions with high latency, though having a separate DMA engine means that memory transfers can be overlapped with computation. The 3D computation was therefore replaced by streaming 2D slices in and out of local store concurrent with calculations. The data layout of the multi-component fields is critical to maintaining good DMA transfer rates and needed to be reorganized by the code generator to allow for fewer but larger contiguous transfers that are essential to achieving good performance.

Contrary to the common perception that stencil computations always have low computational intensity, an analysis of OpenGGCM shows 3.75 and 3.23 flops / byte for predictor and corrector, respectively. Exploiting this fairly high intensity requires reorganization of the code to avoid global temporary fields which was quite easily done with the help of the code generator, but would be rather tedious otherwise. In the existing code, calculating a single predictor or corrector step in the legacy code involves numerous loops over all grid points: For each fluid variable, one (nested 3D) loop calculates fluxes on cell centers at every grid cell. Then, a second loop takes the cell centered

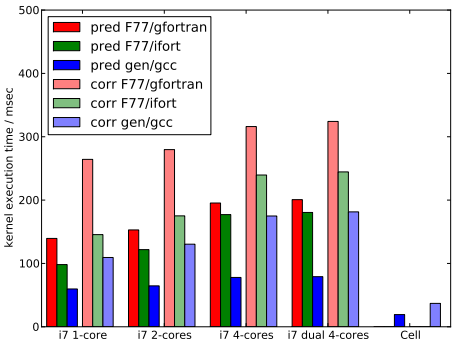

Figure 1. Performance for OpenGGCM kernels on Intel i7 and IBM PowerXCell 8i.

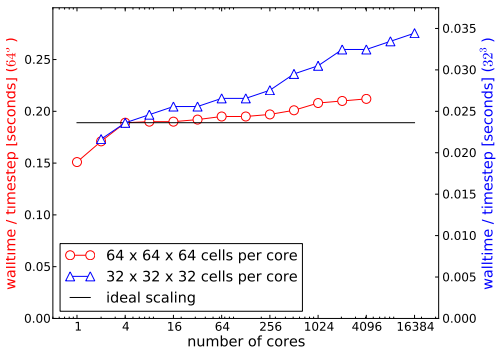

Figure 2. OpenGGCM weak scaling on the Cray XT4 franklin.

fluxes and interpolates them to cell faces. Finally, a third loop adds up all fluxes into each cell. This is repeated for all 5 fluid variables for terms written in divergence form, then additional terms are added in once again separate loops. The introduction of global temporary fields for the fluxes, etc., lowers the computational intensity substantially as mosts loops now only has few floating point operations per iteration, leaving the performance severely memory-bound. The code generator can substitute expressions which were originally calculated in separate loops into a subsequent loop, in the extreme case eliminating all loops but one. However, a disadvantage with this strategy is that quantities will be recomputed across grid points – e. g., in the old code, a flux is calculated once and then used to update both adjacent cells, while in the back-substituted code, the same computation would be performed again at the loop iteration that updates the second cell. To overcome this limitation, the code generator can "cache" field values by storing them temporarily in designated memory areas. x being the fast index, fluxes on the x-faces only need to be stored for one iteration, since the the next iteration handles the cell adjacent to the other side of the face. For y-faces, one row of storage is required, while for z-faces one plane is required. This method still stores temporaries and hence avoids recomputations, but it only uses only a small fraction of the memory otherwise used for 3D temporary fields, so that the temporaries can be kept in cache or the local store on the Cell, respectively, without ever being transferred to main memory.

3. Performance results

Figure 1 presents timing results of the main computational kernels in OpenGGCM on a single node. A series of four runs on an dual quadcore Intel Xeon E5520 (Nehalem) machine running Fedora 12 was performed to investigate the performance implications of running on modern multi-core machines. We show performance for the two main kernels, a spatially low-order predictor (dark-colored) and a switched low/high-order corrector (light-colored). The per-core problem size was kept constant at $128 \times 64 \times 64$ grid cells. Each kernel is run in three variants: Legacy F77 code was compiled with GNU gfortran 4.4.4, "`-O3 -ffast-math`", and Intel ifort 12.0.0, "-O3", enabling auto-vectorization in both cases. The auto-generated code was compiled by gcc 4.4.4 with "`-O3 -ffast-math`". For PowerXCell 8i, results are only shown for the generated code compiled with spu-gcc 4.1.1 with "`-Os`"; the Fortran code is extremely slow.

On a single Nehalem core, the generated predictor kernel is 2.34× and 1.64× faster than the original code compiled with gfortran and ifort, respectively. The generated code achieved 36.2% of peak floating point capacity, while the original code got 18.9% using ifort. Measurements were performed using PAPI (Terpstra et al. 2009). The actual performance gain is less than the improvement in floating point performance due to the fact that some redundant calculations are performed in the generated code, which gave overall better performance than storing and reloading intermediate values.

Going from a single core to a fully occupied dual-CPU node increases the performance gain of the generated code over the original. The multi-core efficiency for the generated core decreases from 100% (1 core) only to 75.4% (8 cores), while it decreases to 54.4% (8 cores) in the case of the ifort code. This is explained by the fact that the original code does have many more smaller loops and hence larger memory bandwidth requirements, leading the increased contention of L3 cache and RAM accesses. As the gfortran compiled code is slower to start with, contention is not as severe and 8-core efficiency is at 69.5%. On a fully loaded node, the generated code is hence a substantial 2.28× faster than what the best compiler achieves for the original.

The results are not as good for the more complex corrector kernel, though the generated code still provides performance improvement. Due to the wider stencils, more distinct memory streams are needed, exceeding the capability of the Nehalem hardware prefetcher, and attempts to overcome this using software prefetching did not reveal a consistent solution. The optimized generated version hence fuses fewer loops than in the predictor case in order to provide conditions where the hardware prefetcher performs well. On a single core, performance gain is 2.41× and 1.33× over the gfortran and ifort compiled original, respectively. Multicore efficiency on 8 cores is essentially the same at 60.3 % (generated) and 59.5% (ifort). gfortran shows much better multicore efficiency at 81.5%, but this is explained by the slow code and hence lower bandwidth contention in the first place.

The performance of the legacy code on the PowerXCell 8i processor is extremely slow, expressing a major disadvantage of heterogeneous architectures: Code not specifically optimized performs very poorly. The auto-generated code, however, performs quite well. Measurements of floating point performance show that we achieve up to $> 40\%$ of peak for one of the kernels. Although the Cell processor is more than three years older than the Intel i7, with properly tuned code it still beats the Intel processor by a substantial margin. Further analysis shows the the performance gains of the generated predictor kernel over the ifort compiled original can be attributed about equally to two kinds of transformations: A little more than half of the gain is due to just rewriting the loop bodies themselves while keeping the overall structure unchanged, while the remainder of the gain is achieved by fusing loops, allowing for increased computational intensity and additional opportunities for CSE.

While the code generator normally generates serial kernels for existing codes, it can also generate MPI parallelized code by using a bundled domain decomposition library; it was used in OpenGGCM to replaced the existing parallelization. Parallel scalability was investigated on NERSC's *franklin* machine, a Cray XT4 with 9,572 compute nodes. Each compute node is powered by a single quad-core AMD Opteron CPU, so we used the auto-generated SSE2 kernels. We performed two sets of weak scaling runs, one with a per-core spatial problem size of $64 \times 64 \times 64$ cells and a second one with $32 \times 32 \times 32$ cells. The 64^3 case represents a more typical use case for OpenGGCM, while the smaller local problem size allowed us to test scalability in a

```
if hall:
    d_i = Param("ggcmblock.d_i")
    # edge centered magnetic field
    bz_ec = b_ec(y, z, bb[0], b, v)
    by_ec = b_ec(z, y, bb[1], b, v)
    # edge centered current
    jy_ec = curr_ec(y, z, b)
    jz_ec = curr_ec(z, y, b)
    # subtract d_i J x B
    Ex -= d_i * (jy_ec * bz_ec -
                 jz_ec * by_ec)
```

Figure 3. Description of the Hall term used as input for the code generator.

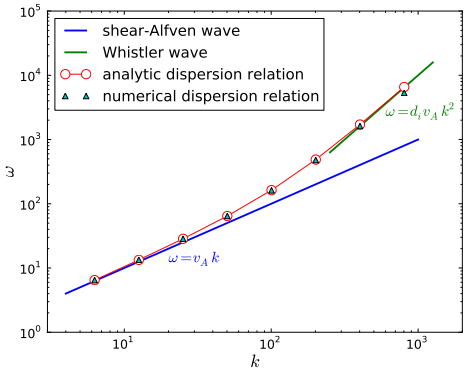

Figure 4. Shear-Alfvén–Whistler dispersion relation for $d_i = 0.01$.

more extreme case: employing more cores for a given global problem size with the goal of decreasing wall clock time for a given run, as might be crucial in a space weather forecasting scenario where results need to be available much faster than real time.

Figure 2 shows our weak scaling results. We plot typical wallclock time used per timestep. As the local problem size remains constant – the global domain size increases in proportion to the number of cores used – ideal scaling is represented by the flat line. We use the 4 core performance as the baseline since one compute node has one quadcore processor. We rescaled the 32^3 case by multiplying it by a factor of 8 to get an effective performance directly comparable to the 64^3 runs.

Parallel efficiency for the 64^3 problem run on 4096 cores is 89% – the scaling study had to be stopped at that point due to unrelated problems in the code handling even larger global grid sizes. As expected, the quite challenging 32^3 local problem size does not scale as well, though still achieving respectable parallel efficiencies of 73% at 4096 cores and 69% at 16384 cores.

4. Hall-MHD

At small scales, the one-fluid MHD model is not a good approximation for the true plasma behavior anymore. A first step in capturing more of the essential physics is to extend MHD by taking two-fluid effects into account in a Generalized Ohm's Law. In particular, at scales below the ion inertial scale d_i, ion and electron fluids separate, which can be taken into account by adding the Hall term $-d_i(\mathbf{J} \times \mathbf{B})$ into Ohm's Law.

Discretizing the Hall term is straightforward using our toolkit, shown in Figure 3. We employ essentially the same method that OpenGGCM uses for the $-\mathbf{v} \times \mathbf{B}$ term in Ohm's Law, which requires quantities to be interpolated onto cell edges of the Yee grid.

Figure 4 shows verification results for our Hall-MHD model. The numerical parameters are chosen to reflect a typical 3D global simulation setup. We used 512 grid points in the direction of wave propagation and an ion skin depth that is still reasonably well resolved: The length of the domain is 1, and we use $d_i = 0.01 \approx 5\Delta x$, where Δx is the grid spacing. The wave number $k = 2\pi m$ is varied choosing $m = 1, 2, 4, \ldots, 128$. The plot shows that we capture the transition from the nondispersive shear-Alfvén wave to the dispersive Whistler wave, as expected, at $kd_i \approx 1$. The circles show the analytic values for the frequency as a function of wave number while the triangles represent the

numerical results. Analytic behavior is matched very well, noticable deviation occurs only at the highest, coarsely resolved, wave numbers.

It is important to consider the computational cost of integrating the Hall-MHD model. Due the dispersive waves, the CFL conditions becomes substantially more stringent. However, the transition to a k^2 dispersion relation only starts at d_i which is a small scale only a few times larger than the grid scale, keeping the computational requirements, while substantial, feasible. In this particular benchmark, the timestep had to be decreased by a factor of 32 over the one-fluid MHD case. Considering that our auto-generated code for the Cell processor let's us run a well-resolved one-fluid case at about real time on our local IBM Cell cluster, running three hours of actual time in the Hall-MHD model will take about four days, which is certainly manageable.

5. Conclusions

We have presented code generation techniques that were used to adapt OpenGGCM to the heterogeneous PowerXCell 8i processor and enhance performance on multi-core processors, but apply more broadly to finite-difference based PDE solvers. Code generation offers a number of advantages: Only one code basis needs to be maintained, and definition of the mathematical model is separated out, increasing programmer productivity, while at the same time enhancing performance substantially. The improvements in OpenGGCM will enable Hall-MHD simulations at reasonable computational cost.

Acknowledgments. This research is supported by DOE grant DE-FG02-07ER46372 and NSF grants CNS-0855145 and OCI-0749125. Computational work has been performed on DOE NERSC and NSF Teragrid systems.

References

Balay, S., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., & Zhang, H. 2004, PETSc Users Manual, Tech. Rep. ANL-95/11 - Revision 2.1.5, Argonne National Laboratory

Brodtkorb, A. R., Dyken, C., Hagen, T. R., Hjelmervik, J. M., & Storaasli, O. O. 2010, Scientific Programming, 1

Christen, M., Schenk, O., & Burkhart, H. 2011, in 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011)

Crawford, C. H., Henning, P., Kistler, M., & Wright, C. 2008, in CF '08: Proceedings of the 5th conference on Computing frontiers (New York, NY, USA: ACM), 3

Germaschewski, K., Raeder, J., Larson, D. J., & Bhattacharjee, A. 2008, in Numerical Modeling of Space Plasma Flows: Astronum-2008, edited by N. Pogorelov, & G. Zank (ASP Conference Series), vol. 406, 223–230

Kamil, S., Datta, K., Williams, S., Oliker, L., Shalf, J., & Yelick, K. 2006, in MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness (New York, NY, USA: ACM), 51

Raeder, J., Larson, D., Li, W., Kepko, E. L., & Fuller-Rowell, T. 2008, Space Sci. Rev., 141, 535

Terpstra, D., Jagode, H., You, H., & Dongarra, J. 2009, Tools for High Performance Computing, pp. 157

Williams, S., Datta, K., Carter, J., Oliker, L., Shalf, J., Yelick, K., & Bailey, D. 2008, Journal of Physics Conference Series, 125, 012038