Tri-Hybrid Computational Fluid Dynamics
on DOE's Cray XK7, Titan.

Aaron Vose[†], Brian Mitchell[*], and John Levesque[‡].

Cray User Group, May 2014.

GE Global Research: *mitchellb@ge.com — Cray Inc.: †avose@cray.com, ‡levesque@cray.com.

Abstract — *A tri-hybrid port of General Electric's in-house, 3D, Computational Fluid Dynamics (CFD) code TACOMA is created utilizing MPI, OpenMP, and OpenACC technologies. This new port targets improved performance on NVidia Kepler accelerator GPUs, such as those installed in the world's second largest supercomputer, Titan, the Department of Energy's 27 petaFLOP Cray XK7 located at Oak Ridge National Laboratory. We demonstrate a 1.4x speed improvement on Titan when the GPU accelerators are enabled. We highlight key optimizations and techniques used to achieve these results. These optimizations enable larger and more accurate simulations than were previously possible with TACOMA, which not only improves GE's ability to create higher performing turbomachinery blade rows, but also provides "lessons learned" which can be applied to the process of optimizing other codes to take advantage of tri-hybrid technology with MPI, OpenMP, and OpenACC.*

1 Introduction

The General Electric Company relies on numerical simulations using Computational Fluid Dynamics (CFD) to design high performing — low fuel burn, low noise, and highly durable — turbomachinery blades for jet engines, gas turbines, and process compressors. These simulations span the range of inexpensive, e.g. a few CPU cores running for minutes, to very expensive, e.g. thousands of CPU cores running for days at a time. By ensuring that these calculations run efficiently on modern supercomputers, GE achieves engineering productivity and ensures that high fidelity simulations can be completed in time scales that impact the design process. GE relies on in-house computational resources for day-to-day design support and utilizes the supercomputers at the US National Laboratories, e.g. Oak Ridge and Argonne, to demonstrate the value of expensive low Technical Readiness Level (TRL) [7] computational approaches.

GE's in-house CFD solver is named TACOMA. TACOMA is a 2^{nd} order accurate (in time and space), finite-volume, block-structured, compressible flow solver, implemented in Fortran 90. Stability is achieved via the JST scheme [1] and convergence is accelerated using pseudo-time marching and multi-grid techniques. The Reynolds Averaged Navier-Stokes (RANS) equations are closed via the k-ω model of Wilcox [8]. TACOMA achieves a high degree of parallel scalability with MPI. For example, GE demonstrated a large scale calculation at Oak Ridge National Laboratory on the Jaguar Cray XT5 supercomputer utilizing 87K MPI ranks in 2011.

In 2013, US Department of Energy's Oak Ridge National Laboratory operationalized a new supercomputer named Titan, which is a 27 petaFLOP Cray XK7 whose compute nodes combine AMD Bulldozer CPUs and NVidia Kepler GPUs [3]. Understandably, Oak Ridge National Laboratory desires that codes running on Titan efficiently utilize the CPUs and GPUs on the compute nodes.

A number of programming paradigms exist for porting codes to the GPU, the most important of which are CUDA [2], OpenCL [6], and OpenACC [4]. For TACOMA, it was decided that the most straightforward porting path was to use OpenACC since it could build

```
do k=1,n3
  do j=1,n2
    do i=1,n1
      df(1:3) = comp_dflux(i,j,k)
      R(i,  j,  k  ) += df(1) + df(2) + df(3)
      R(i-1,j,  k  ) -= df(1)
      R(i,  j-1,k  ) -= df(2)
      R(i,  j,  k-1) -= df(3)
    end do
  end do
end do
```

Figure 1: Pseudocode showing a 3D recurrence. — *As originally implemented, TACOMA utilized a loop structure which contained a recurrence in each of the three dimensions of the result array. GE's OpenMP port utilizes a coloring scheme to avoid this, while Cray's OpenACC port uses a two-loop solution with index translation, as seen in Figure 2.*

on previous work porting TACOMA to OpenMP [5].

Towards this end, General Electric and Cray have collaborated to create a tri-hybrid parallel version of TACOMA that combines three parallelization technologies, namely MPI, OpenMP, and OpenACC. Combining all three of these technologies is critical to extracting maximum performance. We highlight the following key "lessons learned" during the OpenMP and OpenACC porting and optimization process.

1. OpenMP directives should be added before OpenACC because OpenMP scoping work can be largely reused when porting to OpenACC. Additionally, OpenMP is easier due a lack of data motion concerns.

2. OpenACC data motion should be mostly ignored in the beginning, with focus instead on bare kernel performance. Optimization of data motion will likely require examination of the interactions of multiple kernels. This inter-kernel examination is best performed after the kernels involved are completed and have known data requirements.

3. Real world codes often do not express enough of the underlying algorithms' parallelism. Expressing this parallelism is key to achieving good performance. Additionally, some techniques that work well for OpenMP (cell coloring algorithm) are not adequate in the case of OpenACC (a loop fissure and index translation was required).

2 Optimization Work

2.1 Porting to OpenMP

2.1.1 Loop Parallelization for Host CPUs

In 2013, GE extended TACOMA's parallelization to include OpenMP. This was accomplished by carefully porting significant loops that are executed during a multi-grid iteration. Approximately 250 loops were ported by adding appropriate OpenMP parallel directives and data clauses. The motivations for this port were three fold. First, GE wanted the ability to exploit the shared memory on a given compute node. Second, GE wanted to reduce the amount of additional block splitting that was required to ensure adequately parallel decomposition, since the additional blocks consume extra memory and incur computational overhead. Third, GE understood OpenMP to be a gateway for a future OpenACC port.

The programmatic approach to the porting effort, involving multiple engineers at separate global sites, exploited the fact that OpenMP operates on a loop by loop basis. This allows a single loop to be ported at a time and verified for correctness, as each loop's performance is independent of the other loops. To help enable this — and to provide hooks for anticipated future debugging — the `if` clause was used in all OpenMP directives, e.g. `c$OMP parallel do if(use_use)`, to allow for incremental debugging of correctness and performance issues.

Given the block-structured nature of the solver, typical loops consist of a triple loop over `i`, `j`, and `k`. Three general types of these triple loops were encountered which are distinguished by the locality of the op-

```
do k=1,n3
  do j=1,n2
    do i=1,n1
      df(i,j,k,1:3) = comp_dflux(i,j,k)
    end do
  end do
end do
do k=1,n3
  do j=1,n2
    do i=1,n1
      R(i,j,k) += df(i,  j,k,1) + df(i,j,  k,2) + df(i,j,k,  3)
      R(i,j,k) -= df(i+1,j,k,1) + df(i,j+1,k,2) + df(i,j,k+1,3)
    end do
  end do
end do
```

Figure 2: Pseudocode showing the recurrence resolved. — *Cray's OpenACC port uses a two-loop, two-kernel solution with index translation. The first computes all the fluxes, and the second applies them. As no two iterations assign to the same location, there is no recurrence in either of these loops. The edge cases don't matter here, as the surface elements of the result array are never actually read later on in the code. While this approach does require the use of more memory to store the intermediary fluxes as well as the overhead of two kernel launches, it can run efficiently on the GPU due to the large amount of parallelism available in the loop nests.*

erations done inside the loop:

1. Loops with strong locality, e.g.
   ```
   q(i,j,k) = q0(i,j,k) +
              dt(i,j,k) * R(i,j,k)
   ```
2. Loops that involve storing to neighbors, e.g.
   ```
   dflux = q(i+1,j,k) - q(i,j,k)
   R(i+1,j,k) = R(i+1,j,k) + dflux
   R(i,  j,k) = R(i+1,j,k) - dflux
   ```
3. Loops that involved reductions, e.g.
   ```
   l2norm = l2norm + R(i,j,k)**2
   ```

Loops of type (1) can be ported to OpenMP in a straightforward manner. Loops of type (3) are also easily managed with reduction clauses; however this is not always possible due to the use of Fortran 90 complex data structures, thus custom reduction loops were hand crafted as required.

Loops of type (2) require special attention, since the form shown above contains a recurrence, which becomes a race condition when naively parallelized. Loops of this type account for approximately 50% of the CPU time in TACOMA. What these loops are doing is computing the flux (mass, momentum, and energy) `dflux` on the cell face between two cell volumes (`i,j,k` and `i+1,j,k` in the example above), and then accumulating this flux into the net residual `R` stored at the cell centers.

There are two ways to address the race condition. The approach used by GE was to color the cell volumes and then assign a thread to each color. Each thread is then responsible for computing `dflux` on all faces associated with the cell centers of the assigned color, and accumulating the residual. The downside of this approach is a small amount of redundant calculations of `dflux` for faces between nodes with different colors. The second method, which is used in the OpenACC port and discussed in more detail later, is to split each of these loops into two. The first loop computes and stores `dflux` and the second loop accumulates it.

With these three primary loop types successfully parallelized, GE demonstrated the use of hybrid MPI and OpenMP at scale on computations using in excess of 100K total threads (total threads = MPI ranks $\times$ OpenMP threads per rank).

2.1.2 Lessons Learned from OpenMP Porting

The hybrid MPI–OpenMP port of TACOMA serves as a good starting point for the addition of OpenACC directives, enabling TACOMA to run efficiently on accelerator devices such as GPUs. A key lesson learned in this process is that OpenMP should be added before OpenACC. Not only is OpenMP easier as it doesn't require as much analysis of data location, but the scoping performed when adding OpenMP can be largely reused when adding OpenACC as well.

2.2 Porting to OpenACC

2.2.1 OpenACC Kernel Creation from OpenMP Loops

With OpenMP additions completed by GE, work on a GPU accelerated port began in earnest. First, the most time-consuming routines are identified using the Cray Performance Analysis Tool (CrayPAT). In addition to finding the top routines, CrayPAT collects and reports loop-level statistics including min, max, and average iteration counts. The number of loop iterations is critical information to have when attempting to optimize for OpenACC, as each iteration typically forms one thread on the GPU. If there are not enough iterations in a loop, there will be a paucity of threads, resulting in poor performance. In cases where a number of nested loops are to be parallelized, being limited by the number of iterations in an individual loop can be avoided by collapsing the loop nest with the OpenACC `collapse` directive. This directive was used frequently while porting TACOMA.

Once candidate loops have been identified with help from the CrayPAT tool, OpenACC directives can be added to the source code to express the existing loop-level parallelism. During this process, the scoping information from the previous OpenMP work can be largely reused. In particular, variables with `private` scope in OpenMP almost always remain so scoped in OpenACC. The OpenMP reductions will usually be the same as well. However, the variables with `shared` scope in OpenMP are typically those which will require data motion between host and device in OpenACC. These variables can be given a specific OpenACC data motion directive or be left unscoped, leaving the data motion responsibility to the compiler toolchain. Cray's compiler makes very good decisions here, but additional hand optimization of data motion is required to get optimal performance from the vast majority of real-world codes.

While on the topic of real-world codes, it should be mentioned that it is often the case — and certainly was for TACOMA — that expressing the *existing* loop-level parallelism with OpenMP and OpenACC is not enough to achieve good performance. Often, the underlying algorithm contains more parallelism than is available in a given implementation of that algorithm. Thus, in addition to expressing the parallelism in the original code, one often needs to change the implementation so more of the algorithm's inherent parallelism is able to be expressed. An example of this can be seen in the pseudocode in Figure 1, where the original code contains a recurrence preventing parallelization. The first solution to this limitation was created by GE during porting to OpenMP. GE introduced a coloring scheme, whereby each OpenMP thread would be responsible for a region in the result array having a particular color. While the coloring method worked well for the case of OpenMP, it prevented proper vectorization on the GPU with OpenACC. Thus, Cray devised a solution compatible with OpenACC, as depicted in Figure 2. First, the loop is split into two parts: the computation of `dflux` and its accumulation into the net residual array. With this split, the computation of `dflux` no longer contains a recurrence and can be parallelized. The remaining recurrence in the accumulation is removed by an index translation: that is, $\big(a[i]\text{-=}b[i]; a[i\text{-}1]\text{+=}b[i]\big) \implies a[i]\text{+=}b[i\text{+}1]\text{-}b[i]$. Thus, the loop no longer has multiple iterations assign to the same index in the result array a, and the recurrence is removed. Normally such an index translation would introduce some edge cases, but in the case of TACOMA, the surface cells of the output array are never read and don't need to be fully computed.

When adding OpenACC directives to the source code, care must be taken to ensure the compiler is parallelizing the loops properly. Depending on the target compiler and accelerator device, there are a number of levels of parallelism available. In particular, the Cray compiler will by default partition a loop across thread blocks, across the threads within a block, or both. In code with non-trivial loop structures, there can be many ways to perform the partitioning, so care must be taken to ensure the compiler is partitioning in a reasonable way. Cray's compiler provides detailed output in the form of "compiler listing" files, which consist of annotated versions of the source code with ASCII art as well as textual messages. These listing files can be examined to ensure proper loop partitioning.

Finally, the resulting code with OpenACC directives can be executed and profiled. At this early stage, data motion should be mostly ignored when profiling and optimizing OpenACC kernels. Optimization of data motion will require examination of the interactions among multiple kernels. This inter-kernel examination is best performed later, when the kernels involved are completed and have known data requirements. The optimization that can most frequently and most easily produce a large improvement in performance is the tuning of the number of threads per thread block for a given loop with the OpenACC `vector` clause. Compilers

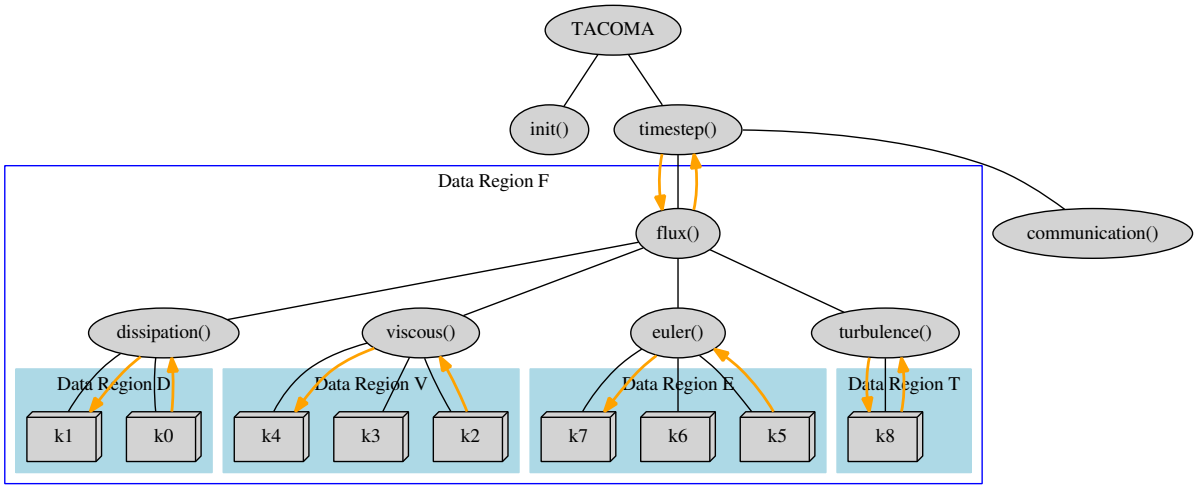

Figure 3: Calltree highlighting OpenACC data regions. — *A large data region (big blue box) can be seen at the level just below the primary timestep loop, where all the kernels in the routines further down the call tree are encompassed. In this region of the code, data that is only needed by the GPU and data that is constant for at least a timestep can be loaded into GPU memory once and be reused for the duration of the timestep. This transfer of data to GPU memory upon entry to data regions and of results to the host upon exit from data regions is represented by the orange arrows in the figure.*

differ in the default vector size, with the Cray compiler utilizing 128 threads per thread block. In the case of TACOMA, a number of loops show nearly double the performance when using a vector size of 256, while some loops perform worse.

2.2.2 OpenACC Data Motion Optimization

Once good kernel performance is achieved, data motion can be examined. To reduce the amount of data that is transferred between host and device, OpenACC data regions are created around adjacent kernels to keep data used by more than one kernel on the device. This first step is depicted by the small light blue data regions in Figure 3. Ideally, the data regions can be merged and moved up the call tree to the primary timestep loop. There, a data region with relatively few lines can wrap all the kernels further down the call tree. This placement in the calltree can be seen by the location of the large blue data region in Figure 3. However, the ideal of merging all or most of the data regions into a larger one isn't always realistic due to I/O, communication, or serial code. In the more pragmatic case, the data regions can be nested hierarchically. In the case of TACOMA, a large data region is created as depicted by data region F in Figure 3. Here, data with a lifetime of at least a timestep can be loaded into GPU memory upon entry and stay on the device for the lifetime of many kernel executions.

With loop-level parallelism expressed and host/device data motion optimized, parallelism among data transfers and kernels is expressed with the OpenACC `async` clause. This allows data to be prefetched before a dependent kernel is launched, and allows multiple kernels from a rank to run on a node's GPU in different streams concurrently. Finally, Cray's CUDA-proxy feature is used, allowing sharing of each GPU among multiple MPI ranks. This allows different MPI to OpenMP ratios and improves OpenACC performance by hiding data transfer latency; a kernel from one rank can execute while data motion for another rank takes place.

2.2.3 Lessons Learned from OpenACC Porting

A key lesson is to ignore data motion and to focus on bare kernel performance during this stage. Optimization of data motion will likely require examination of the interactions among multiple kernels. This inter-kernel examination is best performed a single time, after the kernels involved are completed and have known data requirements. Another key lesson learned is that expressing the parallelism of the underlying algorithm is critical to achieving good performance. At times the code does need to be restructured to allow this. Additionally, some

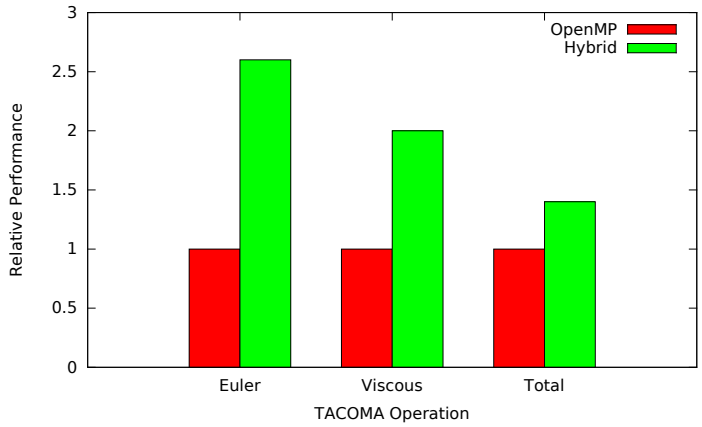

Figure 4: Performance of TACOMA operations. — *The relative total performance and relative average performance of accelerated operations in TACOMA are presented in the figure, with the performance of the MPI–OpenMP version as the baseline. Here, the label "Hybrid" refers to the combination of all three technologies: MPI, OpenMP, and OpenACC. The last column, representing the total runtime of the job, shows a performance improvement of 1.4x.*

techniques that work well with today's OpenMP (cell coloring algorithm) may not work well with OpenACC and GPUs.

During the porting effort, a number of limitations with the OpenACC 2.0 standard were encountered. The most important of which is the lack of support for Fortran derived data types in the regions of code turned into kernels for the accelerator. Working around this issue results in a moderate amount of extra coding, as sections using derived types must be "sanitized". This is typically achieved by acquiring a pointer to the desired data member within the derived type, and passing this pointer to the OpenACC directives instead of the original data member.

In addition to the issues with derived data types, OpenACC has restrictions with subroutine and function calls. OpenACC relies heavily on the compiler's ability to inline function calls within GPU accelerated regions. While the OpenACC 2.0 standard does provide some support for the calling of routines, there are many limitations. In particular, each routine to be called needs to have directives added describing what combination of the three levels of available OpenACC parallelism to use: gang, worker, and vector. To avoid this additional coding work, we make use of as much automatic inlining as possible. Fortunately, the Cray compiler does a very good job of inlining huge routines without issue. For the few cases where a routine was unable to be inlined due to I/O or similar issues, a sanitized version able to be inlined could be constructed.

3 Experimental Results

3.1 Test Platform

At the current stage of the TACOMA OpenACC porting effort, no changes have yet been made to the communication performed by MPI. Thus, the MPI overhead of the hybrid version will remain unchanged in absolute terms compared to the OpenMP version as TACOMA is scaled in node count. For this reason, experiments are designed to highlight the performance improvement the addition of OpenACC provides over OpenMP at a number of realistic node counts, without an explicit focus on scalability. Thus, the tests performed on Titan do not require many nodes, but do represent a realistic amount of work per node.

Tests are performed on Titan [3], the DOE's Cray XK7. Each of Titan's 18,688 compute nodes contains a 16-core AMD Opteron 6274 CPU and an NVIDIA Tesla K20 GPU Accelerator, resulting in a peak performance of 27 petaFLOPs. The K20 GPU supports up to two concurrent streams, so tests used two MPI ranks per node. This allows some overlap of computation and communication, as data transfer for one rank can take place while a GPU kernel from another rank is running. Thus, each MPI rank runs as a hybrid with eight CPU threads and a GPU that is shared with the one other rank

on its node.

3.2 Performance Results

The speedup of selected operations accelerated in TACOMA, as well as the total speedup, is presented in Figure 4. In the results presented, these operations represent just over 45% of the total runtime of the OpenMP version of TACOMA. These operations are reduced to 26% of the total runtime of the hybrid version. We demonstrate an average speedup of 2.6x in euler and 2.0x in viscous, with an overall improvement of 1.4x.

4 Future Work

While the GE-Cray collaboration has resulted in promising performance improvements, there are additional opportunities for improvement and optimization. First, the current measured speed increase of 1.4x reflects the portion of the code that is currently considered adequately tested and debugged. Additional code that is in the process of being debugged should improve the speed up to at least 1.6x.

Beyond that, the data transfer between the host and device is still sub-optimal. Additionally, the code could be modified to make the MPI communication take place from within OpenACC regions. This would reduce the visibility of data motion, as Crays MPI layer can perform the data transfer directly from GPU memory. The current implementation requires the data to make a stop in the host memory area first. Finally, there are a number of remaining computations in TACOMA that could see improvement when moved to the GPU device, e.g. we can expand the amount of computation performed while the data is still resident on the GPU.

5 Conclusion

The addition of OpenACC optimizations provides improved performance, enabling larger and more accurate simulations than were previously possible with TACOMA. This not only improves GE's ability to create higher-performing turbomachinery blade rows, but provides three key lessons which can be applied to optimizing other software applications as well. First, port to OpenMP before OpenACC. Second, optimize OpenACC data motion last. Finally, be sure enough of the underlying algorithms' parallelism is being expressed.

6 Acknowledgments

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. The encouragement of Suzy Tichenor and Jack Wells to undertake the OpenACC porting activity is greatly appreciated.

The efforts of Graham Holmes, Rajkeshar Singh, Shalabh Saxena, Kunal Soni, and Vasant Jain of General Electric and Mamata Hedge and Sudhir Kumar of QuEST Consulting to perform the OpenMP port of TACOMA are gratefully acknowledged. Mark Braaten of GE contributed the idea of the coloring approach used in the OpenMP port. Finally, advice from Nvidia's Jeff Larkin and proofreading help from Ryan Steier is appreciated.

References

[1] JAMES, A., SCHMIDT, W., AND TURKEL, E. Numerical solution of the euler equations by finite volume methods using runge-kutta time-stepping schemes. *AIAA* (1981).

[2] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.

[3] OAK RIDGE LEADERSHIP COMPUTING FACILITY. Ornl debuts titan supercomputer, Mar. 2014.

[4] OPENACC ARCHITECTURE REVIEW BOARD. OpenACC application program interface version 2.0a, Aug. 2013.

[5] OPENMP ARCHITECTURE REVIEW BOARD. OpenMP application program interface version 3.0, May 2008.

[6] STONE, J. E., GOHARA, D., AND SHI, G. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test 12*, 3 (May 2010), 66–73.

[7] US DEPARTMENT OF ENERGY. Technical readiness assessment guide, Sept. 2011.

[8] WILCOX, D. C. Reassessment of the scale determining equations for advanced turbulence models. *AIAA 26(11)*.