Early Experience using the Cray Compiling Environment

Nathan Wichmann, **Chris Brady**, *and* **David Whitaker**
Cray Inc. and **Ed D'Azevedo**, *Oak Ridge National Laboratory*

ABSTRACT: *In 2008, Cray released its first compiler targeted at the X86 instruction set and over the last several months code developers have begun to test its capabilities. This paper will briefly review the life of the Cray compiler, how to use it, and its current capabilities. We will then present performance numbers, for both some standard benchmarks as well as real applications.*

KEYWORDS: *Compiler, CCE*

1. Introduction

Cray Inc. has had a long tradition of providing high performance compilers to its high performance customers. Dating back to the early 1980's, Cray compilers have been known for their vectorization and parallelization capabilities while more recently, with the addition of cache and multi-core architectures, several techniques have been added to maximize the use of the cache, minimize memory bandwidth requirements, and minimize the overhead of parallel regions. However the compiler had been targeted primarily at Cray's traditional vector machines along with some additional RISC based processors. Until recently it had never targeted a CISC based processor such as those produced by Intel and AMD.

As part of its Cascade project, Cray began to investigate adding a new "back-end" to the compiler to generate CISC instructions. This investigation quickly began to focus on an open source compiler called LLVM, short for Low-Level-Virtual-Machine, originally started out of the University of Illinois and more recently championed by Apple Computer.

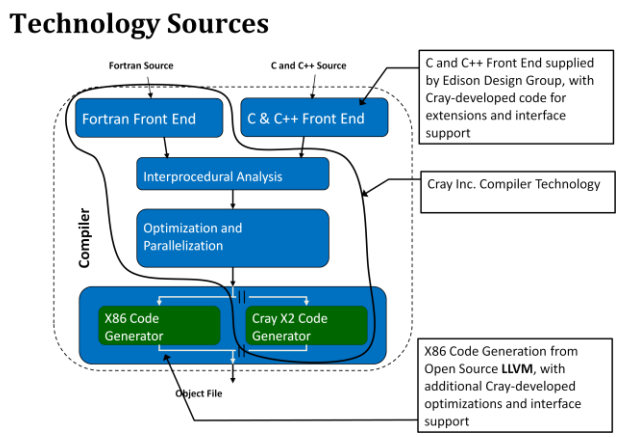
Initial results and the progress of combining LLVM with the Cray compiler were better than expected and, after several months of additional investigation, the decision was made to move forward and productize a Cray compiler targeting the Opteron processor in the Cray XT product line. As a result the Cray Compiling Environment (CCE) Version 7.0 was release in December of 2008.

Basic Components of a Compiler

Most compilers can be broken down into 3 phases with perhaps stages in each phase.

Phase 1 is often referred to as the "front end" and takes files written in computer languages most programmers are familiar with, in this case Fortran, C, and C++, and translates all of those into an intermediate, internal representation used by the next phase of the compiler. The Cray Compiling Environment uses an internally developed Fortran front-end for all Fortran code and front end made by the Edison Design Group for all C and C++ code.

Phase 2 is the largest, and perhaps most complicated phase and is where most of the code transformations takes place. There can be many different names for this phase and its many stages but this is where dependence analysis, loop vectorization and parallelization, loop transformations, and interprocedural analysis and inlinimg are all done. In CCE, almost all of the code and technology are from, or decendents from, the traditional Cray compilers from years past and can therefore take advantage of Cray's considerable experience with vectorization and parallelization.



Firgure 1: Phases and components of the compiler

Phase 3 is call the "back end" and translates the compilers internal representation into an assembly language targeted as a specific processor type. Here Cray has always be had to create their own back ends to targets its proprietary vector processors, and it is here where Cray might have had to create a new back end to translate to the CISC assembly language used in the AMD Opteron processor. Instead, Cray uses the back end component of LLVM, translates from the Cray to LLVM representation, and lets LLVM generate the assembly code. Creating the links to attach these two components together was much less work than creating its own back end and Cray can take advantage or any ongoing enhancements and updates to the open source code.

Using CCE

Using the Cray Compiler Environment is simple and straightforward. If it is installed on your system there should be a module called PrgEnv-cray. A simple "module avail PrgEnv-cray" will indicate its presence. If it is available simply load the module PrgEnv-cray, or if you already have a PrgEnv you first have to unload that other PrgEnv because one cannot have two different PrgEnvs loaded at the same time.

At default, CCE will target a now relatively old dual core Opteron. Most customers have quad-core Opterons and are concerned with getting the most out of those systems. To target that processor make sure that you have the module "xtpe-quadcore" loaded. This not only sets the proper flags for the compiler, but also makes sure the proper libraries are loaded and linked into the executable.

Once you have all the modules loaded the normal "ftn" and "cc" commands will invoke the Cray compiler. Cray's philosophy is to optimize as much as possible at default, so when first porting and optimizing no options are necessary or recommended. One option that is recommended is using the "-rm" option for fortran and the "-hlist=m" option for C. These options will generate what we call "listing files", files that are basically copies of the original source but annotated to indicate what optimizations and transformations the compiler did to your code. It is here where you will look to see if particular loops were vectorized or unrolled.

Current Capabilities

When trying out a new compiler a developer wants to know its current capabilities. Not surprisingly, one of our current capabilities and strengths revolves around vectorization. Cray has always be experts in vectorization and that continues with this compiler. CCE will tend to vectorize more than other compiler, throttled by what we feel we can profitable vectorize. This is an area where we are intentionally aggressive and may vectorize loops which are not profitable to vectorize. If one encounters such a situation, one can prevent vectorization of that loop by placing a "!dir$ nextscalar" directive just before the loop.

For shared memory programming we are currently supporting OpenMP 2.0. One unique capability that we have is support for nesting of OMP regions. So if you have multiple, different levels of parallelism it may be beneficial to attack both levels rather than only one or the other.

With this compiler we are also supporting the Partitioned Global Address Space, or PGAS, programming models. This are often know as Co-Array Fortran and Unified Parallel C. With these programming models one can directly access memory anywhere in the system. For now, this implementation focuses only on functionality and performance; in fact we expect performance to be poor due to the underlying hardware and the youth of this feature. We expect performance to improve over time and, in particular, to improve with the introduction of the Gemini network.

Making codes run as fast as possible means doing whatever is possible to reduce memory bandwidth and hide memory latency. CCE does automatic cache blocking, management of what stays in cache, prefetching, loop interchange and fusion, and much more.

Performance of GTC

GTC is a plasma fusion simulation code developed by Professor Zhihong Lin of UC Irvine. It is a 3-D particle-in-cell 32-bit precision code where the torus is decomposed into between 32 and 64 slices in the torodial direction and then the particles are distributed in any given slice across groups of processors. GTC has several different computational characteristic in the code including stirde-1 copies, strided memory loads and stores, high computation intensity, gather/scatter, and sorting a packing.

The main routine is known at the pusher, and it is responsible for making the calculations that push particles around. There are two loops in the pusher that are of particular importance. The first loop contains groups of 4 statements with significant indirect addressing.

```
e1=e1+wp0*wt00*(wz0*gradphi(1,0,ij)+wz1*gradphi(1,1,ij))
e2=e2+wp0*wt00*(wz0*gradphi(2,0,ij)+wz1*gradphi(2,1,ij))
e3=e3+wp0*wt00*(wz0*gradphi(3,0,ij)+wz1*gradphi(3,1,ij))
e4=e4+wp0*wt00*(wz0*phit(0,ij)+wz1*phit(1,ij))
```

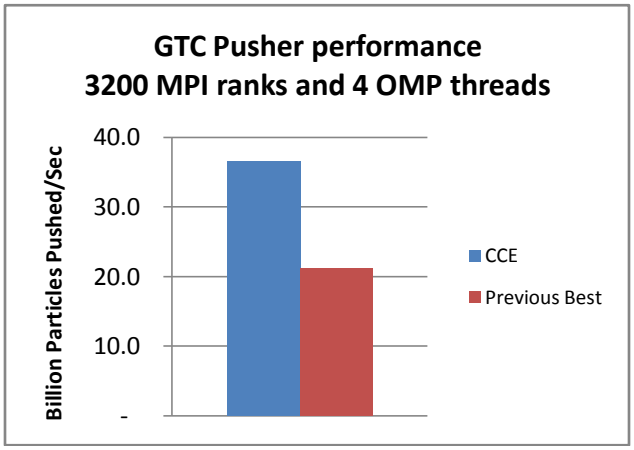
The key thing to recognize is that these 4 statements are nearly identical except that they use 3 different elements of gradphi and one element of phit. Fortunately gradphi and phi are not modified during the pusher phase itself so we were able to copy gradphi and phit into a new variable with 4 elements in the first dimension. One can then rewrite the 4 statements into a single statement using array syntax:

```
ev(1:4)=ev(1:4)+wp0*wt00*(wz0*tp(1:4,0,ij)+wz1*tp(1:4,1,ij))
```

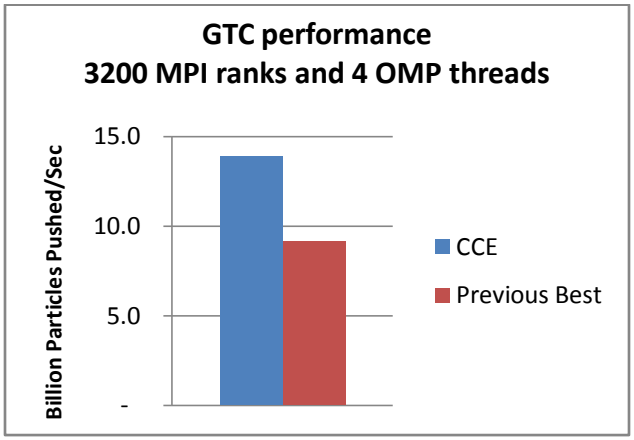
This loop is not exactly the correct number of trips to be fit inside vector sse instruction. CCE does what is called a vector shortloop, eliminates the loop structure and just uses vector operations.

The second loop nest is large, computationally intensive, but also has strided loads and computed gathers. CCE automatically vectorizes this loop. While the strided loads and computed gathers hurt vector performance, it is more than offset by numerous computations in the loop going faster.

The net result of the optimization and automatic vectorization is a 1.75x improvement in speed of the pusher kernel when compare to the previous best using a different compiler.



The entire code is much more than just the pusher, and CCE does very well on the rest of the code as well, without any modification.



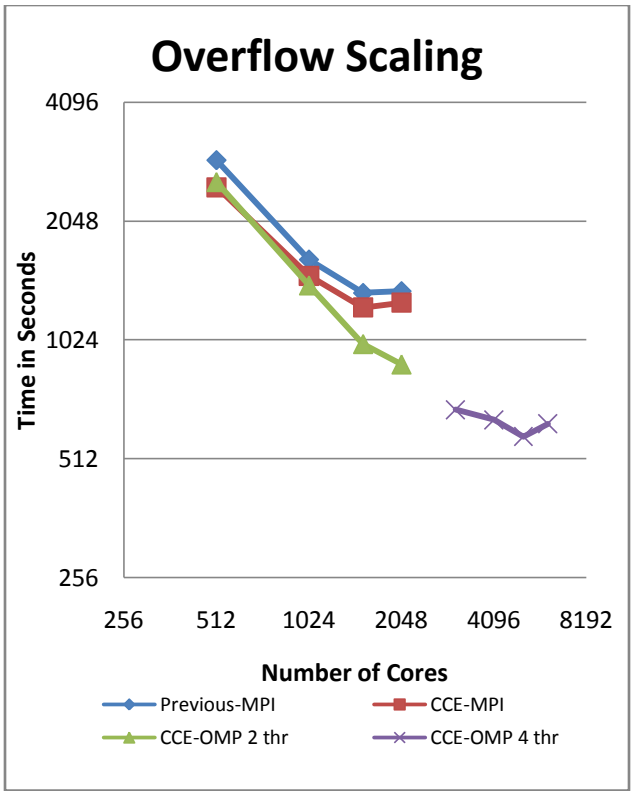
Performance of Overflow

OVERFLOW is a three-dimensional Navier-Stokes flow solver that uses a series of over-set structured grids. Each structured grid is composed of a series grid points in a block configuration. The grid block has dimensions of JD, KD and LD.

OVERFLOW uses the SPMD (Single Program, Multiple data) method for parallelism. MPI is used for exchange data between each instance of the OVERFLOW program. In OVERFLOW, parallelism is is implemented at a high level, by going parallel on multiple grid blocks. During a single time step, OVERFLOW will compute flow quantities in grid blocks in parallel. At the end of the time step, chimera boundaries are updated for each grid block.

OVERFLOW, as part of a pre-processing step, will divide the collection of grid blocks into parallel groups. The number of parallel groups will match the number processors being used in the parallel run. Scaling is limited due to load balance issues at over 1024 mpi ranks. The code has OpenMP at a high level

Overflow was run several different ways, using only MPI to use multiple cores, and then using OpenMP on first 2 then 4 threads underneath MPI.



What we found was that just switching to use CCE made the code about10% faster. You can see that when using only MPI the code went its fastest using about 1500 cores. However if we used MPI in conjunction 2 OpenMP threads we were able to keep scaling until 2048 cores. Finally, if we used 4 threads, we hit a maximum speed using 5120 cores. Scaling us is not great up to that point, but it is possible to go about 2.3 times faster than what was possible using only MPI.

Performance of PARQUET

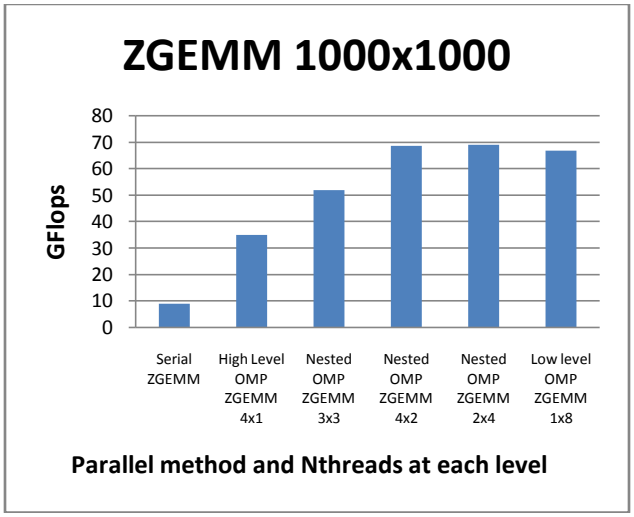
PARQUET is materials science code who's main kernel consists of 4 independent matrix multiplications using the complex data type. The code scales to thousands of MPI ranks, but after at that point it runs out of parallelism and cannot scale further.

Because our MPI scaling is limited, we want to use shared memory parallelism to effectively have as powerful nodes as possible, allowing us to use many more cores than would be possible using only MPI. To date, OpenMP was used to do the 4 matrix multiplications in parallel. This, however, was not able to use all 8 cores in an XT5 node. With the number of cores in a processor, and thus the node, ever increasing, we need a new way of attacking the shared memory parallelism available. We decided we wanted to use multi-level OpenMP to not only do the independent matmuls in parallel, but to do each matmul using multiple cores. The resulting code can be seen below:

```
!$omp parallel do …
do i=1,4
   call complex_matmul(…)
enddo

Subroutine complex_matmul(…)
!$omp      parallel do private(j,jend,jsize)!
num_threads(p2)
   do j=1,n,nb
     jend = min(n, j+nb-1)
     jsize = jend - j + 1
     call zgemm(transA,transB,m,jsize,k,              &
          alpha,A,ldA,B(j,1),ldb, beta,C(1,j),ldC)
   enddo
```

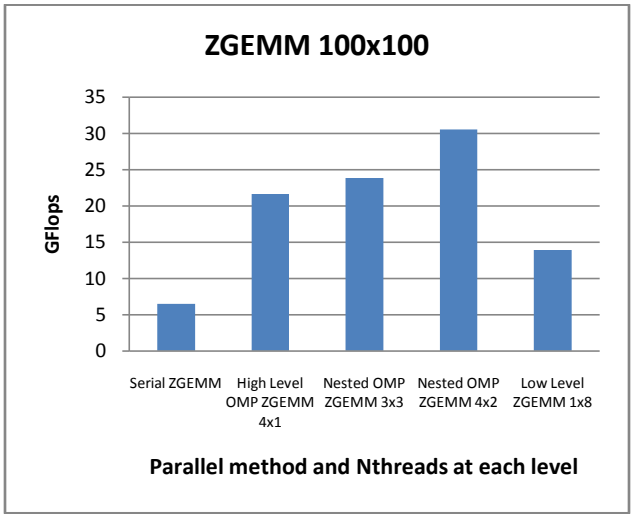
We were then able to compare the performance of several different runs. Our baseline was running each zgemm serially, with no shared memory parallelism. Next we ran the 4 zgemms in parallel, as is done in the current version of the code. There we can see that indeed we run about 3.9 times faster than doing the zgemm serially. Than we made a series of multi-level OpenMP runs using a 3x3, 4x2, and 2x4 distribution where the first number represents parallelism across independent zgemms and the second number represents the number of threads used to calculate a single zgemm.



What we found is that in fact the 4x2 distribution was almost a perfect 2 times faster than using only a single level of OpenMP.

We also wanted to know what the performance graphs would look like if the matrixes were much smaller. The graph below shows the performance if the zgemms were only 100x100. There we see that the 4x2

distribution was still 1.4 times faster than using only a single, high level OpenMP implementation.



One obvious question is why didn't we just do OpenMP inside the zgemm and forget about the high level OpenMP? In the 1000x1000 case in fact the 8-way low level OpenMP had a very good speed-up over the serial case, but you can see that it is not quite as good as the 4x2 multi-level case. But we can really see a difference in the 100x100 case. In that situation the 8-way zgemm not only did not match the performance of the 4x2 distribution, it was about 33% slower than using the high level OpenMP.

Conclusions

The Cray Compiling Environment is a new, different and interesting compiler with several unique capabilities. Despite its young age, several codes are already taking advantage of CCE to go faster or attack new levels of parallelism. CCE is constantly being developed to improve its current capabilities and to add new features. Users should consider trying CCE if they think they could take advantage of the capabilities of the Cray Compiling Environment.

Acknowledgments

The authors would like to thank colleagues and the compiler developers for supporting this effort and answering many questions. We would also like to thank Oak Ridge National Laboratories for the computer time to run some of these codes and generate some of the data.

About the Authors

Nathan Wichmann is member of Performance Team at Cray Inc. and is also a member of the Cray Center of Excellence at Oak Ridge National Laboratories. He has a particular interest in single cpu optimization and optimization done by the compiler. David Whitaker is a member of the Cray Performance Team who specializes in Aerospace CFD applications. Dr. Whitaker received his Ph.D. in Aerospace Engineering from Virginia Polytechnic Institute and State University. Chris Brady is a member of the Cray Performance Team at Cray. Ed D'Azevedo is the group leader for the Computational Mathematics Group at the Computer Science and Mathematics Division at the Oak Ridge National Laboratory. He obtained his PhD at the Department of Computer Science at the University of Waterloo. His interests are in high performance scientific computing, numerical linear algebra and optimal mesh generation.