Procedia Computer Science

Volume 29, 2014, Pages 1480–1490

ICCS 2014. 14th International Conference on Computational Science

A High Level Programming Environment for Accelerator-based Systems

Luiz DeRose, Heidi Poxon, James Beyer, and Alistair Hart

Cray Inc.

ldr@cray.com, heidi@cray.com, beyerj@cray.com, ahart@cray.com

Abstract

Some of the critical hurdles for the widespread adoption of accelerators in high performance computing are portability and programming difficulty. To be an effective HPC platform, these systems need a high level software development environment to facilitate the porting and development of applications, so they can be portable and run efficiently on either accelerators or CPUs. In this paper we present a high level parallel programming environment for accelerator-based systems, which consists of tightly coupled compilers, tools, and libraries that can interoperate and hide the complexity of the system. Ease of use is possible with compilers making it feasible for users to write applications in Fortran, C, or C++ with OpenACC directives, tools to help users port, debug, and optimize for both accelerators and conventional multi-core CPUs, and with auto-tuned scientific libraries.

Keywords: Programming environment, Hybrid system, accelerators, GPUs, OpenACC

1 Introduction

The current trend in the supercomputing industry is to provide hybrid systems with accelerators attached to multi-core processors. However, portability and programmability are critical hurdles for the widespread adoption of accelerated computing in high performance computing. The dominant programming frameworks for accelerator based systems today are CUDA and OpenCL. They offer the power to extract performance from accelerators, but with extreme costs in usability, maintenance, development, and portability. To facilitate the migration of applications to hybrid systems with accelerators attached to CPUs, users need a hybrid programming framework that is simple and portable across machine types. It is important that this programming framework allows users to maintain a single code base. In addition, the required optimization techniques should not be significantly different for "accelerated" nodes from the approaches used on current multi-core x86 processors. The OpenACC application program interface (OpenACC 2011) was created to address these issues. OpenACC is an open standard that started as a joint initiative between CAPS, Cray, NVIDIA, and PGI, and is now supported by more than 15 academic and industrial partners.

 Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2014

doi:10.1016/j.procs.2014.05.134

While programming interfaces like OpenACC are a crucial ingredient, a complete high-level software development environment is needed to make hybrid systems effective HPC platforms. This facilitates the porting and development of applications to run efficiently on either accelerators or CPUs. In this paper we present the Cray high level programming environment for accelerated computing, which tightly integrates compilers, tools, and scientific libraries. Compilers allow users to write applications in Fortran, C, or C++ with OpenACC directives; the tools help users port, debug, and optimize for accelerators as well as conventional multi-core CPUs; and auto-tuned adaptive scientific libraries provide optimized and accelerated performance while maintaining the standard APIs. The remainder of this paper is organized as follows: In Section 2 we briefly describe the OpenACC application program interface. In Section 3 we present the Cray programming environment for accelerated computing, focusing on the programmability features in the compiler, tools, and libraries. Section 4 presents performance results using a large scale application. Finally, we present our conclusions in Section 5.

2 OpenACC Overview

There is currently a wide variety of programming frameworks that support accelerator-based systems, with CUDA and OpenCL dominating. Both target the hardware at a low level and offer the power to extract high performance from the accelerator. However, the resulting codes are difficult to write and extend, and are also hard to port to other HPC platforms. This usually results in the developers having to maintain two distinct versions of the code, one targeted for the homogeneous system, and an accelerator specific version. In particular, in the case of CUDA this means the code will only generally execute on an NVIDIA GPU. An OpenCL code is more portable in theory, but its low-level often requires major program modifications when porting.

With the OpenACC application programming interface (API) the original Fortran, C or C++ application is annotated with directives or pragmas and, optionally, with calls to a runtime API that direct the compiler to generate kernels that execute on the attached accelerator(s). Support from multiple compiler vendors offers portability of applications between systems, reassuring continued ongoing support and development for the programming interface and enhanced opportunities for debugging and performance comparison. By the end of 2012, compilers from Cray (Cray Inc. 2013), PGI (Wolfe 2012), and CAPS (CAPS 2012) were already offering full support for the OpenACC 1.0 standard. By the end of 2013, Cray was already offering compilers supporting OpenACC 2.0, while GNU and PGI were reportedly working on support for this new version of the standard.

OpenACC compilers include a runtime component that implicitly manages the accelerator memory, insulating the user from much of the complexity found in other accelerator programming frameworks. However, sometimes it can be useful to know where data is held in the device memory. The OpenACC host data directive is the mechanism that allows subprogram calls within host data region to pass pointers in device memory rather than in host memory. This is particularly useful for the seamless interaction with scientific libraries, as described in Section 3.3. In addition, the host data directive allows interoperability with CUDA. This gives users the flexibility to construct hand-tuned CUDA kernels for finer control and improved performance in key areas of their application, processing data already held on the device. Since this paper focuses on a high level parallel programming environment for accelerator-based systems and not the OpenACC standard specifically, readers can refer to the OpenACC page (OpenACC 2011) for a complete description of the interface.

3 Cray Programming Environment for Accelerated Computing

The Cray high level programming environment for accelerated computing consists of tightly coupled compilers, libraries, and tools that interoperate and hide the complexity of the system. The compiler does the "heavy lifting" to split off the work destined for the accelerator and performs the necessary data transfers. In addition, it does optimizations to take advantage of the accelerator and the multi-core x86 hardware appropriately. Full debuggers with integrated support for the host and the accelerator are available with DDT (Allinea 2011) and TotalView (Gottbrath 2012). The Cray Performance Tools (DeRose, et al. 2008) provide profiling information for the whole application, which can be grouped by events, such as copies, kernels, etc., or mapped back to the source code by line number. A single performance report can include statistics for both the host and the accelerator, including hardware performance counters information, gathered with PAPI (PAPI team 2011).

The Cray Scientific Libraries use the Cray auto-tuning framework to select the best kernel for each task. With this scientific libraries interface, data copy and the accelerator or host execution placement are automatic.

3.1 Cray Compiler for Accelerated Computing

The Cray compiler was extended to support OpenACC with the goal of simplifying the complexity of hybrid code development. The user identifies regions of code via directives to be translated by the compiler for execution on the accelerator. The user has two mechanisms available for identifying "kernels" for the compiler; the parallel and kernels directives. The parallel directive is prescriptive and the compiler is given very little responsibility for discovering worksharing: all loops that the programmer wants workshared must have a loop directive on them. The kernels directive is more descriptive requiring the compiler to identify all of worksharing loops as well as all of the resulting accelerator kernels. The loop construct is allowed in the kernels construct so the programmer can guide the compiler when necessary.

The Cray compiler utilizes both vector and parallel analysis systems to identify loops for the kernels construct, and vector analysis to ensure that vectorization is chosen if possible for loop nests within parallel constructs. Once the kernels have been identified the compiler splits the code into separate compilation units, the kernels become accelerator routines and are replaced with launching code in the host code stream. The compiler translates the identified work share loops using MIMD and SIMD style parallelism, performs the data movements by allocating and freeing device memory as well as moving data to and from the device at the start and end of accelerated regions. The compiler will explicitly use the accelerator shared memory for reused data when directed by the programmer.

```
. . .
343. G 2 G--------< !$acc parallel num_workers(2)
345. G 2 G !$acc loop private(i,j,ij)
346. G 2 G g------< DO j = 2,jmax-1
347. G 2 G g g----< DO i = 2,imax-1
348. G 2 G g g ij = (j-2)*(imax-2)+i-1
349. G 2 G g g sendbuffz1(ij) = wrk2(i,j,2)
350. G 2 G g g sendbuffz2(ij) = wrk2(i,j,kmax-1)
351. G 2 G g g----> ENDDO
352. G 2 G g------> ENDDO
. . .
369. G 2 G--------> !$acc end parallel
370. G 2 !$acc update
```

Figure 1: Compiler feedback with accelerator related information

Compiler feedback is an extremely important tool for application porting and tuning. Users need to know the optimizations that were done, and more importantly, the optimizations that were inhibited due to particular code constructions. We extended the Cray compiler to provide extensive accelerator related feedback. Figure 1shows an example of the compiler listing with accelerator information for a data region of the Himeno benchmark (Riken 2001), a 3D Poisson equation solver. For each loop in the code the compiler provides an annotated listing with optimization and parallelization information according to the key shown in Figure 2.

```
Primary Loop Type          Modifiers
------- ---- ----          ---------
A - Pattern matched        a - atomic memory operation
                           b - blocked
C - Collapsed              c - conditional and/or computed
D - Deleted
E - Cloned                 f - fused
G - Accelerated            g - partitioned
I - Inlined                i - interchanged
M - Multithreaded          m - partitioned
                           n - non-blocking remote transfer
                           p - partial
                           r - unrolled
                           s - shortloop
V - Vectorized             w - unwound
```

Figure 2: Cray compiler listing legend

In addition, as shown in Figure 3, the listing presents positive and negative compiler messages (e.g., ftn-6263), as well as information on data movement (e.g., ftn-6415), which is critical for performance tuning on hybrid systems. This lets the user understand how the compiler will move data, how it schedule iterations between accelerator threads, whether it will parallelize loop nests, etc. Users can get additional information about specific compiler messages using the explain utility, which is depicted in Figure 4. The Cray Reveal tool (described below) presents an integrated, graphical view of this compiler information.

```
ftn-6413 ftn: ACCEL File = himeno_caf_acc.f08, Line = 292
  A data region was created at line 292 and ending at line 485.

ftn-6263 ftn: VECTOR File = himeno_caf_acc.f08, Line = 306
  A loop starting at line 306 was not vectorized because it
  contains a reference to a non-vector intrinsic on line 413.

ftn-6415 ftn: ACCEL File = himeno_caf_acc.f08, Line = 310
  Allocate memory and copy variable "wgosa" to accelerator,
  copy back at line 338 (acc_copy).

ftn-6405 ftn: ACCEL File = himeno_caf_acc.f08, Line = 343
  A region starting at line 343 and ending at line 369 was placed
  on the accelerator.

ftn-6430 ftn: ACCEL File = himeno_caf_acc.f08, Line = 346
  A loop starting at line 346 was partitioned across the thread
blocks.
```

Figure 3: Example of accelerator related compiler messages

```
> explain ftn-6415
ACCEL: Allocate memory and copy %s to accelerator, copy back at line
%s (acc_copy).

The compiler generated code to allocate memory on the accelerator for
the specified data at the starting line. The Accelerator data is
initialized from the host data. At the ending line, the host data is
updated from the accelerator and the accelerator memory is freed.
```

Figure 4: Explain utility in the Cray programming environment

3.2 The Cray Performance Tools for Hybrid Systems

To port applications to run efficiently on any system, application developers need a good set of performance tools to understand the behavior of the application on the system. There are a variety of performance tools that support accelerated computing, with the NVIDIA Visual Profiler (NVIDIA 2008), the TAU Performance System (Malony, et al. 2011), and the Vampir Trace (Dietrich, Ilsche and Juckeland 2010) being the most prominent ones. While all of these support OpenACC to some extent, none are tightly integrated with the compilation environment, which limits their usefulness.

The main design goal of the Cray performance tools for accelerated computing is to provide the user with an integrated performance measurement and analysis toolset that would view the hybrid system (host and accelerator) as a single entity, presenting the application performance data from both hardware components together. Figure 5 shows the Cray Apprentice2 performance overview for a hybrid code. The overview display is similar to the overview that is available for a run on a homogenous system, with the addition of accelerator specific information. The Function/Region Profile highlights the top three time consuming functions in the program, which could be on the host or on the accelerator. The Data Movement section provides the total amount of bytes "copied in" and "copied out" between the host and accelerator. The Profile pane has an additional bar for the accelerator. The "CPU" bar represents the overall wall clock execution time for the program, and also provides a breakdown of the percentage of time spent in computation, communication, and I/O. The added "GPU" bar provides a time relationship between the host and the accelerator, giving the user a feel for how much of the overall execution time was associated with the accelerator. A short GPU bar compared to the CPU bar indicates that the accelerator was often idle, while a tall GPU bar indicates that most of the execution time was associated with the accelerator. In addition, the GPU bar provides a breakdown of the time for the GPU, so the user can have a high-level view of the time spent executing kernels on the accelerator or moving data between the host and accelerator. The user can dive into any of the overview panes to obtain additional information such as host or accelerator times, number of occurrences of events, and hardware counter information.

Cray Apprentice2 can also display a GPU timeline view that correlates accelerator and host activities, as shown in Figure 6. For a given time slice it shows a histogram of the accelerator activity by wait, copy, or kernel time, the host callstack relative to the accelerator stream activity, and the host/accelerator execution overlap.

In addition, to assist users with code optimization, the Cray compiler and the Cray performance tools were extended to provide loop level profiling information, and to correlate source code with analysis to help identify key candidate areas for optimization. The compiler provides static analysis information and instrumentation hooks at the loop level, which are used by the CrayPat component of the performance tools to instrument and collect the information. This combination of static and runtime information is presented in the Reveal graphical user interface, shown in Figure 7. The right panes have a source code browser with annotated compiler information and compiler messages,

described in Section 3.1. The left pane displays performance information from all loops in the code. The loops can be sorted by time, so the user can quickly identify loops that are best candidates for parallelization and correlate with the source code on the right.

3.3 Scientific Libraries Support

Extensive work has been done by the scientific libraries community to support high performance computing with accelerators, both in industry and academia. Some notable examples are Magma (Agullo, et al. 2009), the NVIDIA CUDA Libraries (Barrachina, et al. 2008), and CULA (EM Photonics 2010). The main drawback of these libraries is that in general they require a specific API for the accelerator; therefore compromising portability.

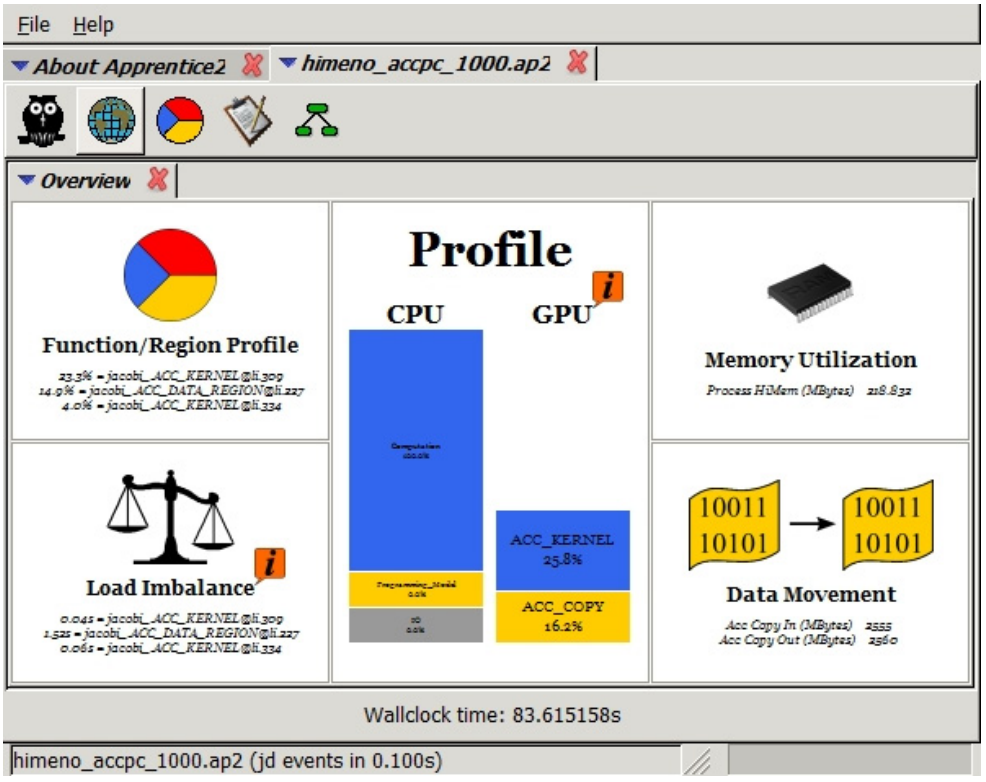

Figure 5: Cray Apprentice2 overview for hybrid systems

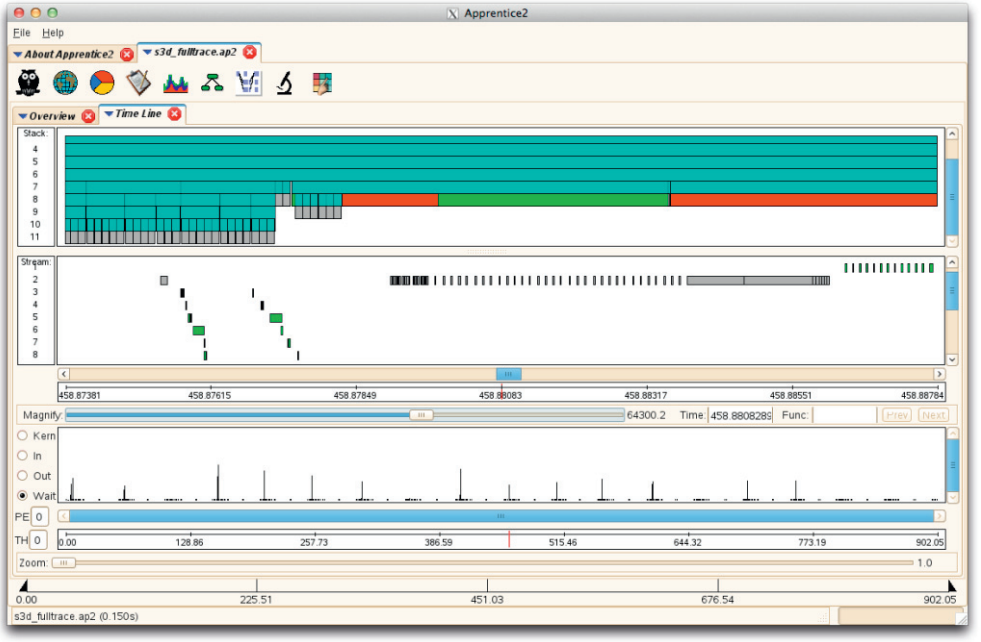

Figure 6: Cray Apprentice2 time line view

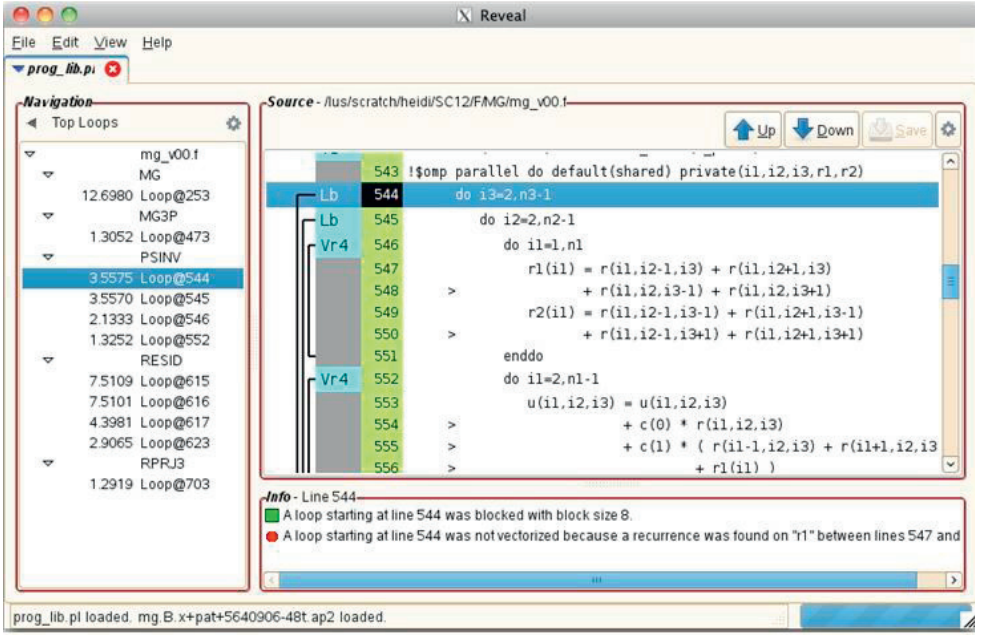

Figure 7: Visualization of integrated static and runtime information using Cray Reveal

One of the design goals of libsci_acc, the Cray scientific libraries for accelerators, was to provide extreme portability, in particular, between homogeneous and hybrid systems. It is critical for application developers to be able to maintain a single code base that could be used effectively on hybrid systems, as well as in homogeneous systems. The approach used in libsci_acc was to provide two sets of interfaces: the simple interface and the expert interface, which can be used in difference scenarios with minimized code modifications. The simple interface requires minimum or no code modification and can take advantage of the accelerator for performance improvement. The expert interface enable users to conduct fine grain code optimization to maximize performance. We find, however, that the expert interface is often not necessary to get the best performance, with the simple interface giving similar speed-ups.

The simple interface uses Cray's adaptation and auto-tuning technology. It contains the full set of BLAS and the major LAPACK routines for accelerator computing, with the same API as the original versions of these libraries. It automatically selects where to run the library (CPU, Accelerator, or Hybrid), depending on the problem, data location, and problem size. Figure 8 illustrates the adaptation in the simple interface. The library first verifies if the data is in host memory or in device memory. If it is in device memory the operation is executed on the accelerator. Otherwise, the library will check the problem size and will decide, based on auto-tuning analysis, if the operation should be executed as hybrid (host + accelerator), accelerator only, or host only.

Figure 9 displays libsci_acc's performance results for matrix multiplication (DGEMM) running on a Cray XK7, a hybrid system with AMD Interlagos CPUs and NVIDIA Kepler (K20) GPUs. The times collected on this example include the overhead for data transfer of the necessary data to or from the CPU. We observe that for most matrix sizes the hybrid DGEMM outperforms the accelerator-only version and that in several cases the hybrid code adds close to 100 GFlops to the total performance (corresponding to the performance of the CPU only code, shown on the *libsci* curve). We also observe that the hybrid approach does not appear to pay off for matrix sizes (M=N=K) smaller than 1792. This is because the computation is not large enough to justify the additional communication overhead in the hybrid approach. For these sizes the library was later modified to call the accelerator version only. Out of the reminding 26 data points, there were four cases, with medium sized matrices, where the accelerator version slightly outperformed the hybrid approach. We are analyzing these cases to fine tune the hybrid approach.

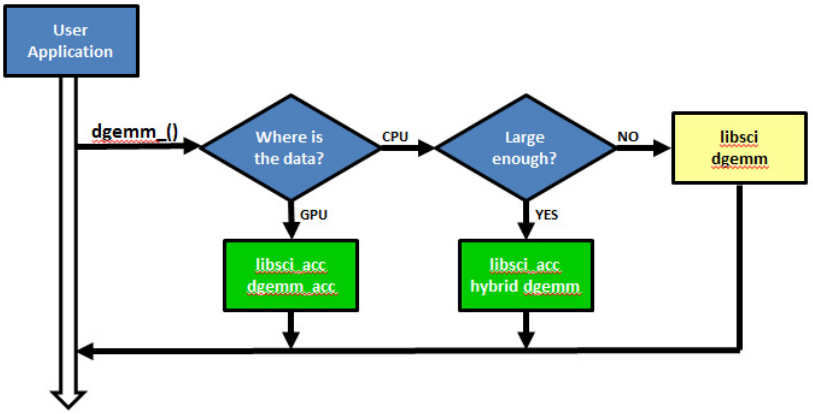

Figure 8: Adaptation in the libsci_acc simple interface

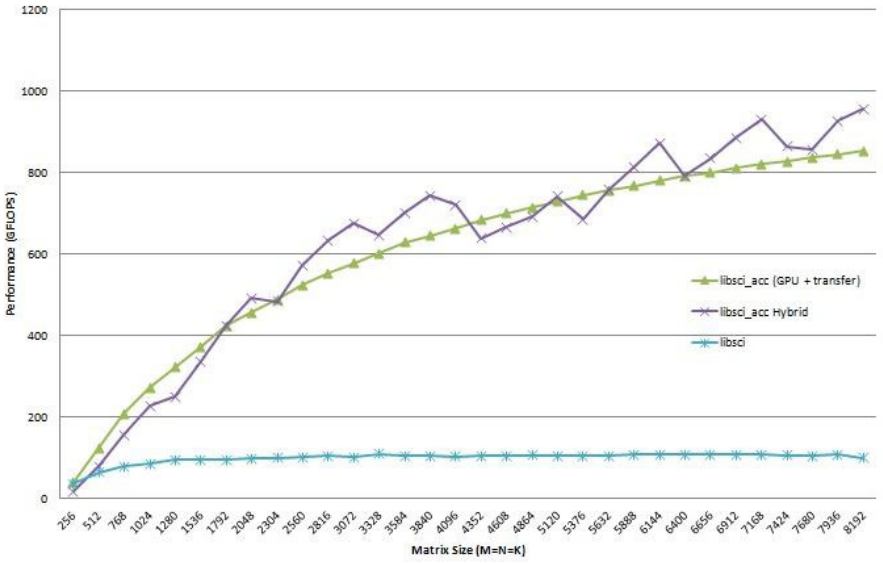

Figure 9: Performance of the libsci acc DGEMM on the Cray XK7

4 Experimental Results

One of the main design goals of the Cray programming environment for accelerated computing was ease of use. Therefore, we do not claim that it will outperform hand-coded CUDA code. We believe that users would be satisfied with the productivity enhancement even with a small performance gap. If needed, users can hand-tune critical kernels with the CUDA interoperability mentioned in Section 2. Our target is to generate hybrid applications that can achieve at least 80% of the performance of a hand-coded CUDA version of the same algorithm. To date, we have seen that is achievable, as we show next with a compute-intensive section of the GAMESS quantum chemistry package (Gordon 1995).

GAMESS is a computational chemistry package suite developed and maintained by the Gordon Group at Iowa State University. We demonstrate the performance of the Cray programming environment for accelerators by comparing with the best known hand-crafted CUDA code using the same algorithm, developed by ISU and NVIDIA. Under this project, the set of kernels called CCSD(T), a method to calculate electronic correlation energy in water clusters, was isolated from the larger GAMESS application and ported to run on accelerators. This set consists of the IJK-tuples, the IJJ-tuples, and the IIJ-tuples. For our comparison we used the IJK-tuples kernel, which contains iterations of communication, followed by various complex array transformations and matrix-matrix multiplies. Much of the data can be copied to the accelerator and left resident for all iterations. Other significant data can be calculated directly on the accelerator thereby saving PCIe bandwidth.

The IJK-tuples kernel has approximately 1700 lines of Fortran code, split into multiple subroutines in a number of separate source files. The kernel is initialized using data from an actual GAMESS execution. The hand coded CUDA version required approximately 1800 lines of new code. In contrast, the OpenACC implementation was developed using the original Fortran source by adding 75 OpenACC directives and doing some minor loop restructuring. The streaming of data transfers used in the OpenACC version was also similar to the CUDA version. Most of the data is moved to the accelerator before the iteration loop whose trip-count depends on the number of ranks. Within the

loop, three arrays are exchanged with other ranks, moved to the accelerator, and remain device resident for the remainder of the iteration.

Table 1 presents the time in seconds of the IJK-Tuples kernel written in CUDA and OpenACC, running on 16 nodes of a Cray XK6 system, a hybrid system with AMD Interlagos 2.1 GHz CPUs and NVIDIA Fermi (X2090) GPUs. It also shows the performance of the original code running on 16 nodes of a Cray XE6 system, which makes a fair comparison between the homogenous system and the hybrid system. We observe that the difference in performance between the CUDA version and the OpenACC version is only 3%, which is much better than our target of not more than 20% performance degradation. Both the CUDA and OpenACC ports were done before the availability of the NVIDIA Kepler GPUs. With the release of NVIDIA Kepler GPU late in 2012, we recompiled and rerun both versions of the code on a Cray XK7, without modifying any of the source codes. The results are also shown in Table 1; the OpenACC version was 12.5% faster than the CUDA version! Although surprising, this can be explained by the changes in both the architecture (specifically the memory subsystem) between Fermi and Kepler, and the CUDA Toolkit, from 4.1 to 5.0. This indicates that a CUDA code tuned for one micro-architecture, such as Fermi, might need to be re-tuned for a different micro-architecture, while a high level code may only need recompiling to exploit the tuning of the compiler and the libraries. We also ported the IJJ-tuples and the IIJ-tuples kernels to OpenACC. However, no corresponding CUDA versions existed at the time, and with the observed performance results, the GAMESS developers opted to complete the port to accelerators with OpenACC.

Cray XE6	Cray XK6 (IL + Fermi GPUs)		Cray XK7 (IL + Kepler GPUs)	
CPU 16 ranks per node	CUDA 1 rank per node	OpenACC 1 rank per node	CUDA 1 rank per node	OpenACC 1 rank per node
311 seconds	134 seconds	138 seconds	76.6 seconds	68.1 seconds

Table 1 : Time of the original version of the IJK-Tuples kernel running on 16 nodes of a Cray XE6 system and the CUDA and OpenACC versions running on 16 nodes of a Cray XK6 and XK7 systems.

5 Conclusions

In this paper we presented the Cray high level programming environment which aims to make accelerated supercomputers a productive platform for HPC. Cray compiler support for OpenACC directives allows users to write hybrid applications in Fortran, C, and C++. The compiler optimization then takes advantage of accelerator and multi-core x86 hardware appropriately. Performance measurement and analysis tools provide a single view of the system with statistics for the whole application. Auto-tuned adaptive libraries allow users to extract maximum performance from the hybrid system, whilst still using the same standard API. Our experiment results show that, with this programming environment, users can productively use high-level programming frameworks like OpenACC and reasonably expect to exceed 80% of the performance of low-level, hand tuned codes for the same algorithm.

References

Agullo, Emmanuel, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, and Julien Langou. "Numerical linear algebra on emerging." *Journal of Physics: Conference Series*, 2009: 180(1).

Allinea. November 2011. http://www.allinea.com/products/ddt.

Barrachina, S., M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Orti. "Evaluation and tuning of the Level 3 CUBLAS for graphics processors." *IPDPS*. 2008. 1-8.

CAPS. *CAPS Compilers*. 2012. http://www.caps-entreprise.com/products/caps-compilers/.

Cray Inc. *Cray Compiling Environment Release Overview and Installation Guide*. 2013. http://docs.cray.com/cgi-bin/craydoc.cgi?mode=View;id=S-5212-82.

DeRose, Luiz, Bill Homer, Dean Johnson, Steve Kaufmann, and Heidi Poxon. "Cray Performance Analysis Tools." In *Tools for High Performance Computing*, by Michael Resch, Rainer Keller, Valentin Himmler and Bettina Krammer, 191-199. Berlin: Springer, 2008.

Dietrich, Robert, Thomas Ilsche, and Guido Juckeland. "Non-intrusive Performance Analysis of Parallel Hardware Accelerated Applications on Hybrid Architectures." *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops (ICPPW '10)*. 2010. 135-143.

EM Photonics. *CULA Tools: GPU Accelerated Linear Algebra*. 2010. http://www.culatools.com/.

Gordon, Mark. *The General Atomic and Molecular Electronic Structure System (GAMESS)*. 1995. http://www.msg.ameslab.gov/gamess/.

Gottbrath, Chris. "Debugging and optimizing scalable applications on the cray." *Cray Users Group Meeting*. Stuttgart, Germany, 2012.

Malony, Allen D., et al. "Parallel performance measurement of heterogeneous parallel systems with gpus." *Proceedings of ICPP*. IEEE, 2011. 176–185.

NVIDIA. *Profiler User's Guide*. 2008. http://docs.nvidia.com/cuda/profiler-users-guide/index.html.

OpenACC. *OpenACC Directives for Accelerators*. 2011. http://www.openacc.org/.

PAPI team . *PAPI CUDA*. 2011. http://icl.cs.utk.edu/projects/papi/repository/index.php/CUDA.

Riken. *The Himeno Benchmark*. 2001. http://accc.riken.jp/2444.htm.

Wolfe, Michael. *OpenACC Features in PGI Accelerator Fortran Compilers—Part 1*. MArch 2012. http://www.pgroup.com/lit/articles/insider/v4n1a1a.htm.