Requirements Analysis for Adaptive Supercomputing using Cray XK7 as a Case Study

Sadaf R Alam, Mauro Bianco, Benjamin Cumming, Gilles Fourestey, Jeffrey Poznanvic, Ugo Varetto
Swiss National Supercomputing Centre
Lugano, Switzerland
{alam, mbianco, bcumming, fourestey, pozanavic, uvaretto@cscs.ch}

Abstract— **In this report, we analyze readiness of the code development and execution environment for adaptive supercomputers where a processing node is composed of heterogeneous computing and memory architectures. Current instances of such a system are Cray XK6 and XK7 compute nodes, which are composed of x86_64 CPU and NVIDIA GPU devices and DDR3 and GDDR5 memories respectively. Specifically, we focus on the integration of the CPU and accelerator programming environments, tools, MPI, numerical libraries as well as operational features such as resource monitoring, and system maintainability and upgradability. We highlight portable, platform independent technologies that exist for the Cray XE and XK, and XC30 platforms and discuss dependencies in the CPU, GPU and network tool chains that lead to current challenges for integrated solutions. This discussion enables us to formulate requirements for a future, adaptive supercomputing platform, which could contain a diverse set of node architectures.**

Keywords-Cray XK7, Cray XE6, Cray XC30, GPU, Xeon Phi, Adaptive computing, OpenACC, MPI, portability

I. INTRODUCTION

Several years ago, Cray introduced a vision of adaptive supercomputing, where diverse heterogeneous computing and memory subsystems would be integrated in a unified architecture. Such a system may include massively-multithreaded systems by Cray, FPGAs and recent instances of accelerator devices such as GPUs and Intel Xeon Phi. In fact, the most recent generation of Cray system called Cascade would support contemporary accelerator technologies.[1] There have been announcements for the support of NVIDIA GPU devices and Intel Xeon Phi [5] accelerators. Even today, Cray XK series platforms combine two sets of memory and processor architectures within a single node and system design. The Cray XK6 platform contains NVIDIA Fermi accelerators while the Cray XK7 platform has NVIDIA Kepler accelerators [6][12]. Using the Cray XK7 platform as a reference, in this report we analyze readiness of a unified programming and operational interface for the next generation of hybrid architectures.

CSCS has recently upgraded a 3-cabinets Cray XK6 system to an XK7 platform by upgrading the accelerator devices, NVIDIA Tesla GPUs. Cray has incorporated NVIDIA driver, programming and runtime interfaces into its Cray XE series operating and programming environment. Cray XK7 system now contains an accelerator device called NVIDIA Kepler K20X, which can deliver over 1.3 TFlops (double-precision) performance. This is about a factor of two improvement compared to the previous generation device, NVIDIA Fermi X2090. There have been no changes to the processor and memory configuration of the node. Together with the hardware upgrade, NVIDIA updated the programming interface to CUDA 5 and introduced new features such as HyperQ, dynamic parallelism, GPUDirect-RDMA and Tesla Deployment Kit [7]. Cray operating system namely CLE and the programming environment (PE) have also been updated together with the device upgrade, however, not all new features of the device are currently available on the system. In this report, we provide details on the integration challenges and analyze possible hardware and software dependencies.

The integration challenges of the Cray XK7 platforms provide us opportunities to characterize code development, execution and operational requirements of an adaptive supercomputing platform. A unified architecture should provide a common interface for:

- Code development and refactoring
- Compilation
- Tuning
- Debugging
- Scaling
- Production runs
- Management and operations
- Maintainability and upgradability

There should be a migration path for not only between different Cray platforms but also for applications that are being developed at local servers and clusters to the Cray environment. Figure 1 shows how different components of the system need to be adapted to provide a unified interface across multiple systems. There is an overlap between the components that are required for code porting, development and tuning, and for an environment where users submit production level jobs. Likewise, there are some tools that are unique for system administrators such as system-wide monitoring and diagnostics but then there are some resource management interfaces that will be required by both production level users and system administrators.

[1] Cray News release dated Nov 08, 2012 http://investors.cray.com/phoenix.zhtml?c=98390&p=irol-newsArticle&ID=1755982

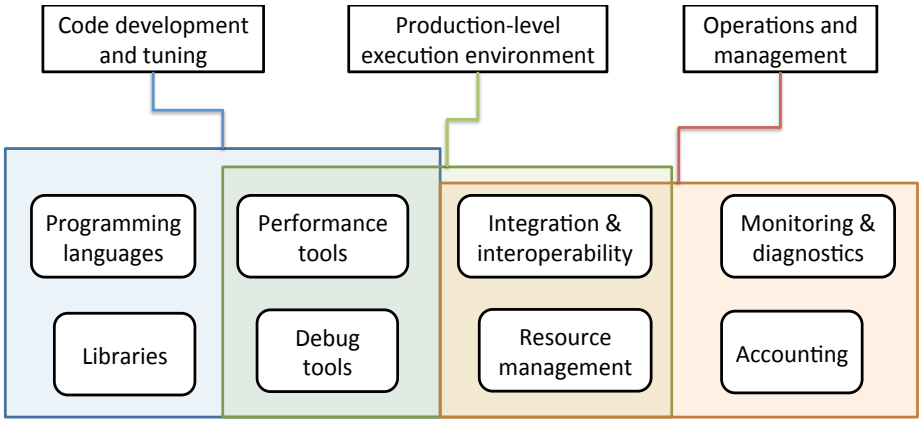

Figure 1: A high level view of code development, production and operational needs for an adaptive supercomputing platform, which may be composed of homogenous and heterogenous multi-core resources. Overlapping components for individual needs have been shown in the figure.

A thorough discussion on each individual component listed in Figure 1 is beyond the scope of this paper. In this paper, we attempt to highlight technologies that are portable across x86_64, NVIDIA GPU and Intel Xeon Phi systems and provide early experiences, status and results. We present the status of the technologies on the Cray XE6, Cray XK7 and Cray XC30 platforms [15][19]. Cray XC30 is the new generation of Cray system, which is composed of Intel processors, a new interconnect technology called Aries and a new topology named dragonfly [14]. Unlike the Cray XK series systems where the interface to the PCIe based accelerator technology is provided through a custom interface to the CPU, the Cray XC30 and its hybrid variants can be considered similar to a standard Linux cluster, where both accelerator and network interface is PCIe. Therefore, the Cray XC30 based hybrid processing nodes may provide a path to adaptive supercomputing or unified programming and execution environment. In this paper, we evaluate the requirements for two hypothetical systems with nodes containing either NVIDIA Kepler GPUs or Intel Xeon Phi accelerators and then discuss readiness of the Cray environment, only in the context of portable technologies.

The portable technologies presented and discussed in this paper are as follows:

- OpenACC/OpenMP
- OpenCL
- Libsci
- Accelerator aware MPI
- Performance tools (perftools)
- Debugger
- Resource management

The most widely used programming language for GPU devices namely NVIDIA CUDA has not been discussed because it is currently not portable to other devices. Similarly, CUDA related performance and debugging tools that are available within the NVIDIA SDK are not discussed in detail in this paper. Integration on external toolsets such as NVIDIA SDK into the Cray programming and execution environment is an important topic of discussion but it is beyond the scope of this paper.

The paper is organized as follows: section 2 provides a background to the Cray XK7 platform, portable technologies and current status. In section 3, we provide details on portable programming interfaces, multi-platform tools and utilities. A brief analysis for Intel Xeon Phi programming and execution environment is provided in section 4, together with a discussion on requirements for an adaptive supercomputing environment. We then summarize our findings in section 5 and provide a plan of work for the next steps, which are necessary for developing a unified architecture for future hybrid and non-hybrid Cray systems.

II. CRAY XK7 PLATFORM AND PORTABLE TECHNOLOGIES

A Cray XK7 node is composed of an AMD Interlagos processor socket, DDR3 memory, an NVIDIA K20X GPU and a Cray Gemini network interface. An AMD Interlagos processor is composed of 16 Opteron cores or 8 compute modules. The CSCS Cray XK7 system has 32 GBytes of DDR3-1600 memory. The K20X device is composed of 6 GBytes (non-EC) GDDR5 memory and has 14 SMX units. The device and the CPU are connected via a PCIe-Hypertransport bridge chip. The network connection is also via the Hypertransport link. A Cray XE6 node, in comparison, has two Interlagos sockets, has same amount of memory but twice the memory bandwidth. Two Cray XK7 and Cray XE6 nodes are connected to a single Gemnin chip. The schematic of the two systems is shown in figure 2.

In contrast, a Cray XC30 node is composed of two Intel Sandy Bridge sockets and a PCIe 3.0, 16x connection to the Aries interface chip. The CSCS Cray XC30 system has 32 GBytes of DDR3-1600 memory. Unlike the Cray XK7 and XE6 platforms, four compute nodes are connected to a single Aries chip. The network topology is also different. The Cray XC30 system has a dragonfly topology with optical connections. The operating environment on the two platforms is also different, CLE 4.x vs. CLE 5.x.

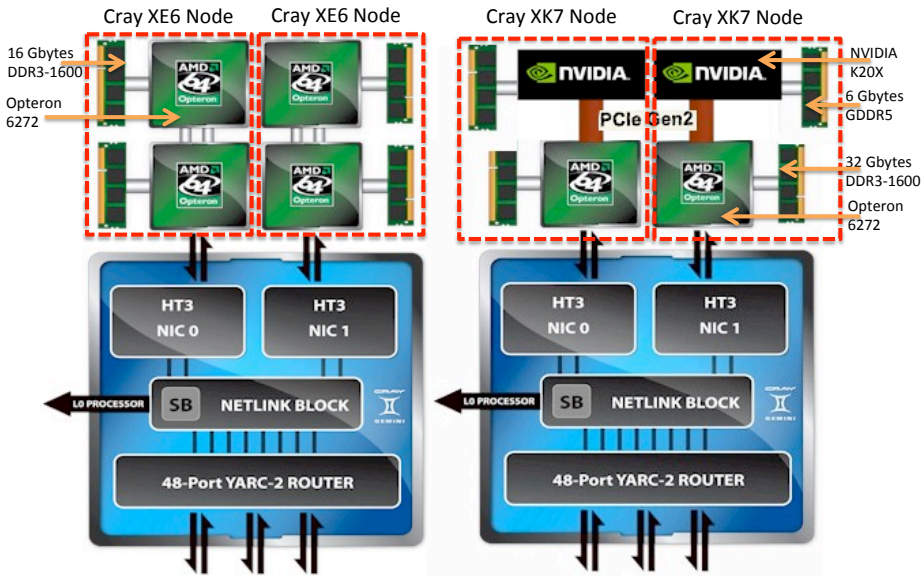

Figure 2: Comparison of Cray XE6 and Cray XK7 blades, each with two compute nodes and a Gemini interconnect chip [13].

The similarities and differences of the Cray XE6 and Cray XK7 platforms are shown in Figure 2. In terms of the software stack for programming and execution environment, there are a number of similarities and difference. Figure 3 shows the overlap and new features of the Cray XE6, Cray XK7 and Cray XC30 programming environment. Cray XK7 and Cray XE6 share the same CLE and I/O stack but differ in programming environment including the Cray numerical libraries (libsci), MPI and tools, which have extensions for the GPU devices. In other words, Cray XK7 platform offers a complete software stack of a Cray XE6 platform to allows for the multi-core only programming. To enable the GPU devices, NVIDIA GPU driver and CUDA SDK have been included. These in turn interface with other Cray and third part compilers and tools. Details on the portable components are provided in the next section.

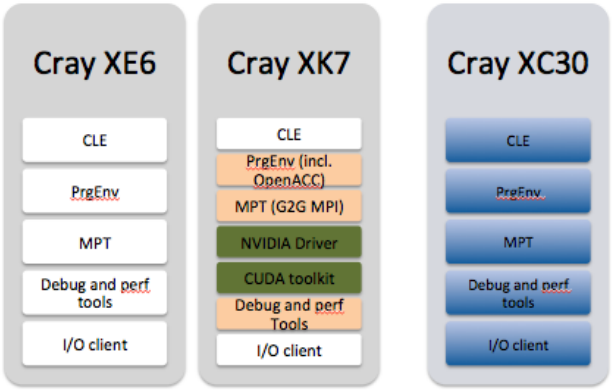

Figure 3: Comparison of software stack for Cray XE6, Cray XK7 and Cray XC30 platforms. Cray XE6 and Cray XC30 are homogenous multi-core platforms with distinct hardware and software stack.

As indicated earlier, the Cray XC30 programming and execution environment is similar to the Cray XE6 platform, but has been updated for the Intel processors and the Aries interconnect. The CLE and kernel are different and MPI has been tuned for Aries network topology and routing. In addition, there is Hyperthreading available on the processor nodes.

Table 1 provides the status and availability of the portable system components. OpenACC and OpenMP for accelerators are considered as incremental programming approaches for multi-core and hybrid multi-core systems such as NVIDIA GPU and Intel Xeon Phi [8][9]. OpenACC offers a set of standard directives and compilers are available from Cray, PGI and HMPP. The OpenMP consortium is currently reviewing extensions for accelerator devices. OpenCL is a platform independent API for multi-core and accelerator devices. There is a standard and device vendors such as NVIDIA, AMD and Intel provide OpenCL compilers for their respective devices. Libsci is a tuned numerical library from Cray, which has been extended as libsci_acc for GPU aware implementation of the BLAS routines [4]. Cray has also extended the MPI library such that MPI calls can be directly made onto the GPU pointers. In addition to programming languages, libraries and MPI, Cray performance tools called perftools have incorporated features that allow users to analyze and investigate issues that influence performance of accelerated codes. Specifically, there have been extensions for the investigation of OpenACC codes that have been developed using the Cray OpenACC compiler. CSCS has the Allinea DDT debugger for both scalable and accelerated code debugging [1]. DDT has also been extended for CUDA and OpenACC applications. It can also be used for multi-

core only MPI and OpenMP debugging, and Allinea has announced support for the Intel Xeon Phi [2]. On the CSCS Cray XK7 system, users automatically have access to the GPU resource. However, CSCS is interested in managing GPU devices and other accelerators as individual resources and would like to understand usage of GPU resources for accounting and resource allocation purposes. CSCS currently use SLURM as a resource manager and scheduler for all systems including the Cray XE6, Cray XK7 and Cray XC30 platforms. Currently GPU usage has not been requested and reported through the Cray ALPS interface to the SLURM database.

TABLE I. STATUS OF PLATFORM INDEPDENT TECHNOLOGIES WITHIN THE CRAY COMPILER ENVIRONMENT (CCE) AND THIRD PARTY TOOLS ON MULTICORE CRAY XE6 AND XC30 AND HYBRID MULTICORE CRAY XK7 PLATFORM WITH GPU DEVICES

	Status and details		
	Cray XK7	***Cray XE6***	***Cray XC30***
OpenACC	CCE, PGI & CAPS	PGI§	no
OpenMP for accelerators*	No	No	No
OpenCL	CPU + GPU	CPU§	No
Numerical libraries	libsci_acc/libsci	libsci	libsci
Accelerator aware MPI	CCE MPT	—	—
OpenACC debugger	Allinea DDT	—	—
Performance tools (Cray perftools, Vampir and TAU)	MPI, OpenMP, OpenACC, CUDA	MPI, OpenMP	MPI, OpenMP
Resource management & accouting (SLURM)	CPU only	CPU	CPU

* proposed extensions is currently under review
§ possible but currently not available

III. EVALUATION OF CRAY XK7 PLATFORM INDEPENDENT TECHNOLOGIES

A. OpenACC Accelerator Directives

In order to facilitate an incremental adoption of the accelerator devices, a few directive-based standards have been introduced. Cray compiler environment (CCE) provides support for the latest standard for accelerator programming called OpenACC. The OpenACC directives provide control for the following functionalities: regions of code to accelerate, data to be transferred to and from the device, and compiler hints for loop scheduling and cache usage. In the simplest form, an OpenACC code may comprise of a couple of additional statements:

```
!$acc parallel loop
DO i = 1,N
a(i) = i
ENDDO
!$acc end parallel loop
```

CSCS has been deeply involved in a number of programming models for heterogeneous node architectures. From our perspective, directive-based approaches (e.g. OpenACC) are one of the many tools in the toolbox that can help developers with porting their codes to future node architectures. For some of the parallel applications running at CSCS, there are clear benefits for retaining the original overall code structure and adding directives to control data movement and expose parallelism. Two examples of OpenACC success stories that run on CSCS systems include the COSMO weather code and the ICON climate model -- both of these prototype applications have shown significant speedups compared to their CPU implementations.

At CSCS, we provide application developers with all three commercial OpenACC compiler implementations: CAPS HMPP, the Cray Compiling Environment (CCE), and PGI Accelerator. Being able to swap between each vendor's implementation has proven to be invaluable during periods of heavy OpenACC application development. We have found that each vendor's implementation has its own strengths and weaknesses within the context of a given application.

Here are some of the challenges that we have experienced with using OpenACC on multi-node GPU applications:

- Multiple developers have inquired about utilizing OpenACC in their C++ codes. The upcoming OpenACC v2.0 standard is expected to provide some key support in terms of unstructured data lifetimes (e.g. for controlling data on the accelerator in constructors and destructors). Future support for deep copies will also be highly useful in this situation.
- One of our user's applications requires the ability to integrate separate CUDA and OpenACC components together in the same executable. For this, OpenACC's "host_data" clause is essential to make the OpenACC device array addresses available to CUDA. However, PGI has not yet fully supported this feature that is part of the OpenACC v1.0 standard.
- Lack of support for multi-dimensional C arrays required an application to be heavily modified to linearize its array accesses. Solutions to this general issue are currently being discussed and implemented by the compiler vendors.
- Getting access to elements of Fortran derived types within accelerator regions caused compile-time errors in an application. A workaround was found for CCE, but not for the other compiler tested. The compiler vendors are currently working on a general solution.
- A performance portability situation in one application has been identified where the developers had to fork their code into a CPU implementation and an OpenACC implementation. The developers found that widespread loop optimizations for the port to

OpenACC had a negative impact on the same code running on CPU targets. Along with some manual loop fusion, it seems likely that adding an OpenACC loop interchange directive to the standard would help with performance portability in some circumstances.

- Hardware portability questions: PGI and CAPS have announced support for architectures beyond Nvidia GPUs, but CCE hasn't announced any future plans for alternative accelerator targets. Also, it would be useful to developers if CCE was additionally available on local workstations.

Finally, one of the long-term goals of OpenACC is to lead to a more robust OpenMP standard for accelerator computing. If a future version of OpenMP includes sufficient support for a variety of accelerator architectures (which is currently unclear!), we believe that OpenACC-ported applications will be very well positioned to transition to the future version of OpenMP.

B. *OpenCL*

OpenCL is a set of open standards that have been introduced for developing programs for systems with heterogeneous compute units. Hardware vendors provide the standard conformant drivers. Hence, OpenCL codes can be executed on both CPU and accelerators. The programming model allows for both data and task parallelism. Like CUDA, there is a concept of parallel programming for a device where concurrent tasks can be grouped into work-items. OpenCL memory model is also somewhat similar to the CUDA memory model, where memory access options depend on how a data structure has been declared.

The Cray XK7 includes a standard CUDA 5 software development kit; from a software development perspective the main difference between the XK7 installation and standard Linux clusters is the fact that the nvcc compiler is not able to find the C/C++ host compiler specified by the loaded modules, it is therefore required for users to explicitly set the host compiler to use through the -ccbin <directory path of gcc executable>.

The SDK from NVIDIA also provides OpenCL implementation, which conforms to the 1.1 OpenCL standard and it is 32-bit only. It has to be noted that all the other OpenCL implementations currently available from other vendors such as AMD (CPU and GPU), and Intel (CPU, GPU and Xeon Phi) are 64-bit and 1.2 conformant. The HyperQ feature of CUDA 5 is also not available for OpenCL applications. On CSCS Cray XK7 platform, OpenCL is available for both CPU and GPU devices. We also provide a custom fix to imitate HyperQ behavior by allowing multiple MPI tasks or processes connect to a single GPU.

Our experience with OpenCL on different platforms indicates that on platforms other than the Intel Xeon Phi performance portability can usually be achieved by changing the kernel launch configuration (total number of threads and number of threads per work-item) and the size of data buffers without major changes to the kernel code. On the Intel Xeon Phi optimizations that work on both GPU and CPU like caching into __local memory do not seem to apply.

C. *Numerical Libraries*

The Cray Scientific Libraries package, LibSci, is a collection of numerical routines optimized for the targeted Cray platforms [4]. A subset of the routines are extended for accelerators, namely libsci_acc. The Cray LibSci collection contains the following libraries, which are automatically called when libsci is loaded:

- BLAS (Basic Linear Algebra Subroutines, including routines from the University of Texas 64-bit libGoto library)
- BLACS (Basic Linear Algebra Communication Subprograms)
- LAPACK (Linear Algebra Routines, including routines from the University of Texas 64-bit libGoto library)
- ScaLAPACK (Scalable LAPACK)
- IRT (Iterative Refinement Toolkit), linear solvers using 32-bit factorizations that preserve accuracy through mixed-precision iterative refinement
- CRAFFT (Cray Adaptive Fast Fourier Transform Routines)
- FFT (Fast Fourier Transform Routines)

In order to harness the full potential of hybrid systems, it is crucial to rely on routines that take advantage of both the CPU (e.g the SIMD unit) and the accelerator processing power (e.g GPGPU or Xeon Phi). However, because of the heterogeneous configuration of hybrid systems, hand-tuning routines can be extremely difficult or tedious, even for very simple algorithms like GEMM. Fortunately, Cray provides an accelerated numerical library (libsci and libsci_acc), which includes some functions that have been optimized to run in hybrid host, multithreaded and GPU accelerated configurations. The library has been designed to work in different modes: it can work without any code modifications where data transfers to and from the device are hidden from users and it allows modifications to enable code developers to hide data transfer latencies. For example, a user can make the following call (using the normal Lapack interface):

```
dgetrf(M, N, A, lda, ipiv, &info)
```

Depending on the size of A, libsci will either run the dgetrf routine on the host, the device or both so that total performance will be maximized on the node. If instead of CPU, the GPU device pointers are being passed the code will execute on the device:

```
dgetrf(M, N, d_A, lda, ipiv, &info)
```

Data must be transferred to the GPU prior to the call to improve performance. Finally, each routine has a device

(_acc suffix) and a host (_cpu suffix) interface to give a higher degree of control to the user. For instance, calling dgetrf_acc (resp. dgetrf_cpu) will force the execution on the device (resp. the host). The libsci accelerator interface can also be invoked within the directives environment:

```
   !$acc data copy(c), copyin(a,b)
   !$acc host_data use_device(a,b,c)
   call
dgemm_acc('n','n',m,n,k,alpha,a,lda,b,l
db,beta,c,ldc)
   !$acc end host_data
   !$acc end data
```

Table II lists the performance of libsci for the DGEMM routines, which has been highly tuned for different target platforms.

TABLE II. STATUS OF PLATFORM INDEPDENT TECHNOLOGIES WITHIN THE CRAY COMPILER ENVIRONMENT (CCE) AND THIRD PARTY TOOLS

Platform	***Peak floating-point performance (double-precision GFLOPS/s)***	***DGEMM performance using libsci/libsci_acc***
Cray XE6 (dual-socket AMD Interlagos)	269	228
Cray XK6 (NVIDIA X2090)	665	450
Cray XK7 (NVIDIA K20X)	1311	1180
Cray XC30 (dual-socket Intel Sandy Bridge)	333	315

D. Accelerator Aware MPI

MPI libraries such as MVAPICH2 [20], OpenMPI and recently Cray MPI have added support for CUDA memory pointers such that code developers do not need to explicitly transfer data between host and device memories. Like any platform specific MPI library from Cray, this MPI library provides an optimal interface to transfer data between the GPU devices over the high-speed interconnect. We provide details of a project that successfully exploit this feature.

Many HPC applications use domain decomposition to distribute the work among different processing elements. To manage synchronization overheads, decomposed sub-domains overlap at the boundaries and are updated with neighbor values before the computation begins. A subset of applications using domain decomposition is finite difference kernels on regular grids, which are also referred to as stencil computations, and the overlapping regions are called ghost or halo regions. Typically, these applications make use of MPI Cartesian grids and each process handles a regular multi-dimensional array of elements with halo elements.

Even though the data exchange pattern (neighbor exchange) is clearly defined when the Cartesian computing grid is defined, at application level there are many parameters that can vary: One is the mapping between the coordinates of the elements in the domain and the directions of the coordinates of the computing grids; the data layout of the domains themselves; the type of the elements in the domain; the periodicities in the computing grids to manage the case in which certain dimensions wrap around or not. Additionally, when we want to deal with accelerators, which typically have their own address space, we need also to specify where the data is placed.

While these parameters are application dependent, others are architecture/platform dependent. Other degrees of freedom are related to how to perform communication (e.g., asynchronous versus synchronous), what mechanism to use to gather and scatter data (halos) from the domains, etc. All these variables make the specification of a halo exchange collective operation quite complex. For this reason we have developed the Generic Communication Layer (GCL) to provide a C++ library of communication patterns directly usable by application programmers and matching their requirements flexibly. At the moment, GCL provides a rich halo exchange pattern and a generic all-to-all exchange pattern that allows specifying arbitrary data exchange but it is not fully optimized.

GCL has been designed as multi-layer software. At bottom layer (L3) the definition of a communication pattern involves only data exchange considerations, in which each process knows what to send to every other process involved. Above L3 there is a more application oriented pattern specification (L2), which deals with user data structures. For instance, the halo exchange communication pattern at level L2 can process any 2D or 3D arrays, of arbitrary element types, halo widths, and periodicities. To deal with more arbitrary data structures and applications, another layer (L1) had been devised to use high order functions to collect data and communicate, but this level has not been implemented yet and requires careful design, that we would like to engage with other partners. The interface at level L1 would resemble others found in [16][17][18].

We show here the comparison between XE6 and XK7 machines. We compare two halo-exchange patterns at level L2: halo_exchange_dynamic_ut and halo_exchange_generic. The first pattern assumes an arbitrary number of equally shaped arrays with same element types, halo widths and periodicities to be exchanged, but the address of memory will be known only when the data exchange will be executed. The user then needs to specify at pattern instantiation time:

1. The element types and number of dimensions of the arrays;
2. The memory layout of the arrays;
3. The mapping between dimensions of the arrays and the dimensions of the computing grid;
4. Where the data is stored (either host or GPU memory);
5. The description of the halos of the arrays in each dimension (in an order specified by the programmer convention specified by the layout of the arrays);

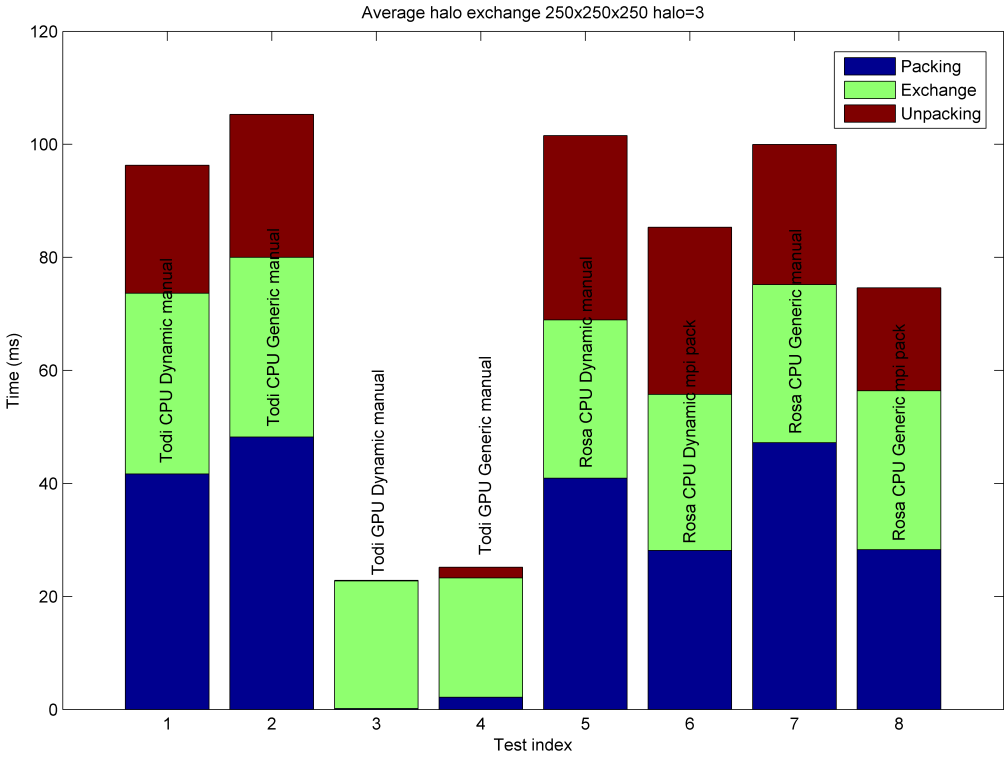

Figure 4: Comparison of Cray XE6 and Cray XK7 blades, each with two compute nodes and a Gemini interconnect chip.

The second pattern, halo_exchange_generic is more relaxed and allows the user to exchange arrays with arbitrary shapes, layouts, element types, and halos widths. For this reason at instantiation time the only information needed at instantiation time is the layout of the computing grid, the number of dimensions of the computing grid, the placement of the data (host or GPU), and an upper bound on the sizes of the arrays and halos, in order to allocate a sufficient amount of memory for gathering and exchanging data. At the moment of the exchange each array will be passed to the pattern as a field descriptor, which indicates the actual sizes for the data to be exchanged. Next Figure 4 shows the comparison of a data exchange on XK7 (Tödi) and XE6 (Rosa) when performing halo exchange on three 250x250x250 arrays, with halo width of 3 elements. The time shown includes gathering and scattering of data and the actual exchange (from array to array). As can be seen, even though XK7 MPI is not fully optimized for GPU handling, the time for collecting and placing data, makes the GPU performance very interesting in terms of performance. Rosa results shows also the results for two mechanisms for collecting and placing data, namely hand written loops and calls to MPI_Pack functions. In this case MPI_Pack (and the use of MPI_Datatypes) gives the best performance, but on XK7 this mechanism cannot be used when data is on the GPU. In this case, handwritten CUDA kernels are used to collect at source, and to place the data at the destination. CUDA memory copies are performed from/to host memory so to use MPI send and receive routines to exchange data. Other MPI implementations allow the use of GPU memory pointers, which make it possible to simplify the library code and improve performance by employing pipelining between GPU memory transfers and network injection.

E. Debuggers

We have had some success using the Allinea DDT and TotalView debuggers at scale on CPU-based systems at CSCS [1][10]. For example, there was a recent situation where a hybrid MPI+OpenMP application was hanging 10 minutes into its run across the full 12 cabinets of our Cray XC30 system. To troubleshoot this issue, we were able to attach DDT to the application in its hung-state. DDT's parallel call-stack feature pointed to an abnormal situation where thousands of processes were waiting at a barrier for a single process to finish work in a recursive function. Given this information, application developers were able to find the underlying cause of the problem.

Moving forward, we would like this success story to continue into the realm of heterogeneous computing. Allinea currently has support for Nvidia GPUs, and they have announced support for the Intel Xeon Phi architecture.

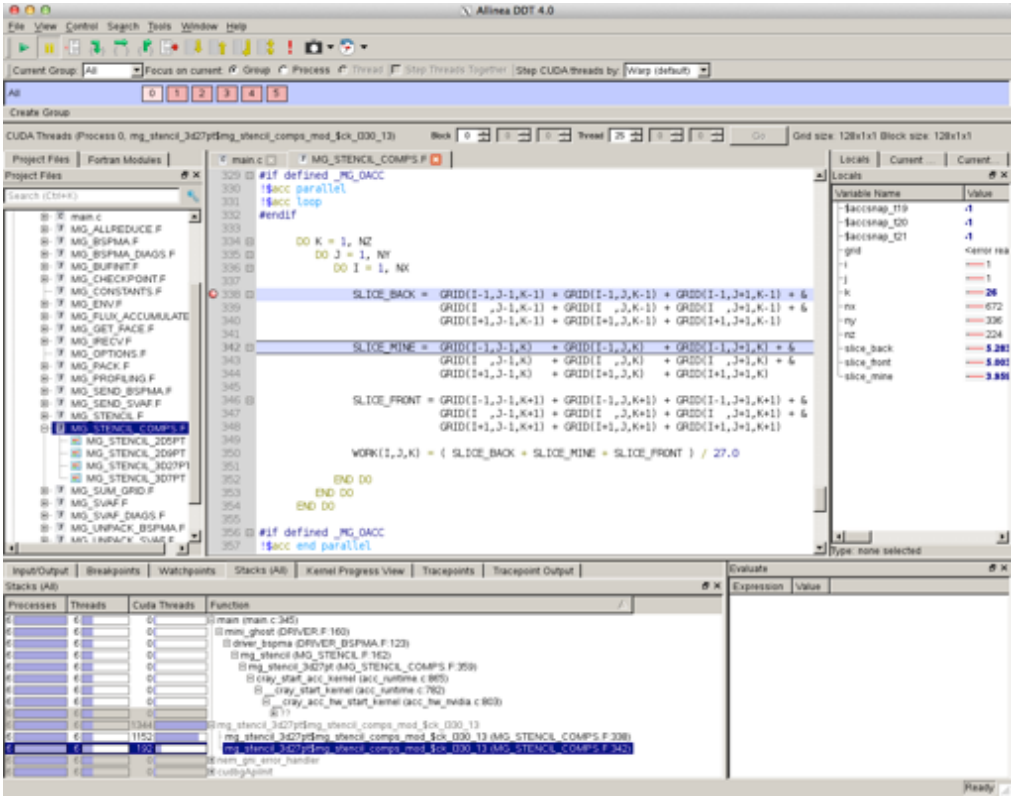

Figure 5: Output of the Allinea DDT debugger for an MPI+OpenACC application

To debug a multi-node GPU application, DDT gives the ability to step into GPU kernels and to inspect memory on the GPU. At CSCS, we have already had some luck using DDT to debug MPI+GPU applications. As a simple example, when investigating optimizations to the MPI+OpenACC version of the Mantevo miniGhost benchmark,[2] we were able to use DDT to pause the execution and inspect the variables directly before the code crashed on the GPUs, which is shown in Figure 5.

However, we have also had a number of challenges with debugging GPU applications over the past year. One example was related to the debugger freezing when attempting to step past CUDA API calls -- this was fixed with an updated version of Nvidia's cuda-gdb packaged within DDT 4.0. A currently unresolved issue involves an MPI+CUDA+OpenACC application that returns a "CUDA_ERROR_INVALID_DEVICE" error solely when running within the debugger. While more investigation is required, these types of errors have typically required timely updates to the Nvidia drivers on Cray systems. Additionally, there are ongoing OpenACC-specific debugging challenges -- CCE provides full device DWARF debug support, while the other vendors provide only partial support. Finally, users who want to debug their scalar non-MPI GPU codes on Cray systems run into DDT/aprun errors; the workaround is to add a dummy MPI_Init call in the code, but these types of usability issues may inhibit first-time users.

F. Performance Tools

Profiling tools are essential for hybrid computing because the additional complexity of both host and device computation makes analysis of application performance more challenging. Some of the tools discussed below, like CrayPAT and Vampir [11], use PAPI[3] to access hardware counters. Thus, these tools can be extended to provide hardware counter information for architectures that are supported by PAPI. This is currently the case for NVIDIA GPUs, which PAPI accesses via the CUPTI interface provided by NVIDIA. Support for Xeon Phi was added to PAPI 5.1.0, however we have been unable to test this support on the Xeon Phi test cluster at CSCS because the Xeon Phi devices require a kernel patch, which is incompatible with recent releases of the Xeon Phi software stack. Nonetheless, once these issues are resolved, profiling tools that use PAPI will naturally extend to support Xeon Phi.

[2] http://www.mantevo.org/

[3] http://icl.cs.utk.edu/papi/

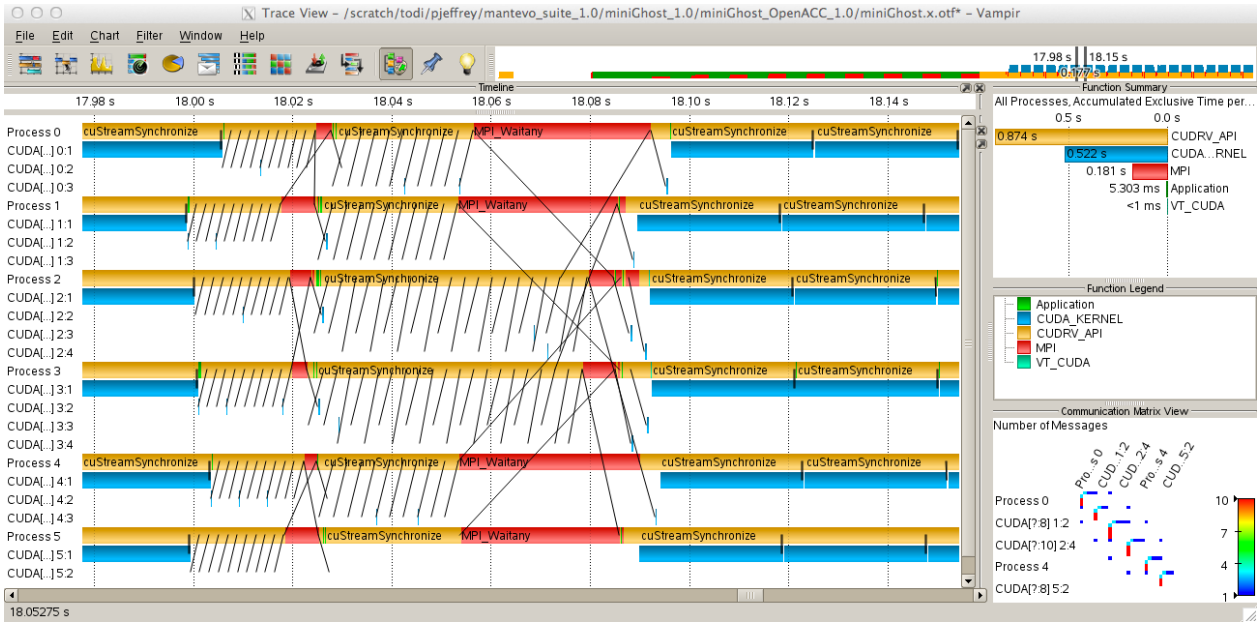

Figure 6:Timeline-style performance analysis of asynchronous operations between multiple host CPUs and accelerators.

A hybrid application that utilizes accelerators like GPUs or Xeon Phi may have several asynchronous events to keep track of: asynchronous MPI communication, asynchronous transfer of data between host and device, and asynchronous execution of kernels on host and device. Timeline-style profiling tools are particularly useful for gaining insight into asynchronous events as illustrated in Figure 6. While the tracing experiments performed by these tools are generally less scalable than profiling experiments, the information gathered from timeline-style performance tools at small to moderate node counts can be difficult to obtain by other methods. The two products used at CSCS, Vampir and TAU,[4] have had support for GPUs for some time already, and support for Xeon Phi should be possible by virtue of both tools also being based on PAPI.

Highly scalable tools like CrayPAT provide with an overview of where the code is spending its time. CrayPAT is very easy to use for GPU applications and can be run efficiently on thousands of nodes. Currently, support is best for OpenACC. CUDA applications are supported by CrayPAT; however, at least in the past, the visibility of the CUDA kernels was constrained to the surrounding function call (i.e. a single function containing multiple CUDA kernel launches will only show the single function in the profile and not the separate kernels). Additionally, CrayPAT supports accelerator hardware counters.

NVIDIA and Intel provide GUI-based performance analysis tools, namely Visual Profiler and VTune, that are familiar to developers on GPUs a Xeon Phi. These tools allow users to quickly gather and display useful device-specific performance data, in a format that is tailored to the accelerator. The visual feedback provided is very useful for understanding the aforementioned asynchronous processes, along with features unavailable from their command line equivalents (e.g. derived performance counter metrics in Visual Profiler). However, these tools require cluster compatibility mode (CCM), which is incompatible with the SLURM scheduling software used at CSCS. We hope to resolve these issues, because device-specific profiling tools are important.

G. Resource Managament, Accounting and Monitoring

An HPC center that deploys homogenous and heterogeneous computing platforms, integrating all platforms into its ecosystem for management and monitoring purposes within a 24/7 production environment is critical. Resource management and accounting is also central to users within a production environment where resources are allocated on a project by project basis. CSCS uses SLURM resource management system, which is deployed on all of its Cray and non-Cray platforms. Currently, on the non-Cray platforms, users can declare their intent by using the `gres` parameter in their job submission scripts. On the Cray systems, it is not required as there is a custom interface on compute node called ALPS. Furthermore, the GPU devices do not record usage statistics and cannot be managed like CPU devices for fine grain core and memory allocation. GPU devices are now gradually integrated into monitoring systems such as Nagios.[5]

[4] http://www.cs.uoregon.edu/research/tau/home.php

[5] http://www.nagios.org/

IV. REQUIREMENTS ANALYSIS AND READINESS

Now we consider readiness of the current Cray XK7 platform together with two hypothetical configurations of the Cray Cascade XC30 system, one with NVIDIA GPU devices and another with Intel Xeon Phi. We assume that the host CPU will be similar to a homogenous multi-core XC30 platform. The readiness of different platform-independent technologies is listed in Table III. Note that we only list technologies that are publicly available at the time of writing this report. For example, PGI and CAPS [3] have announced OpenACC compilers for the Intel Xeon Phi devices; however, currently PGI compilers are not available on the XC30 platform and we have not tested the functionality on a non-Cray, Intel Xeon Phi cluster. Likewise, only Intel MPI is available to use on a non-Cray cluster with Xeon Phi accelerators. We are unable to confirm whether Cray MPI will be extended to support all execution modes of the Intel Xeon Phi processors. In short, for the platform independent technologies listed in Table III, there is not enough evidence to establish how these will be transparently supported on two different heterogeneous XC30 platforms. Based on the publicly available information and experiences on the Cray XK7 and a non-Cray Intel Xeon Phi cluster, a substantial effort will be required to realize a unified environment across multiple platforms.

TABLE III. READINESS OF THE CRAY XK7 SYSTEM AND HYPOTHETICAL CRAY XC30 SYSTEM WITH GPU AND XEON PHI FOR PORTABLE SOFTWARE SOLUTIONS (EXCLUDING CUDA AND INTEL OFFLOAD AND INTRINSICS TECHNOLOGIES). AVAILABILITY OF GPU TOOLS IS EXTRAPOLATED FROM CRAY XK7 PLATFORM.

	Readiness of technologies		
	Cray XK7	***Cray XC30 + GPU$^{\delta}$***	***Cray XC30 + Xeon Phi$^{\delta}$***
OpenACC	Yes	Yes	No$^{\xi}$
OpenMP for accelerators*	No	No	No
OpenCL	Yes	Yes	Yes
Numerical libraries	Yes	Yes	No
Accelerator aware MPI	Yes	Yes	No
OpenACC debugger	Yes	Yes	No$^{\xi}$
Performance tools (Cray perftools, Vampir and TAU)	Yes	Yes	Yes$^{\xi}$
Resource management and accounting (SLURM)	No	No	No

* proposed extensions is currently under review
$^{\delta}$ hypothetical configuration
$^{\xi}$ products announced but currently unavailable

V. SUMMARY AND THE NEXT STEPS

Using Cray XK7 as a reference platform, Cray XE6 and Cray XC30 experiences and insights that have been gathered on non-Cray platforms with NVIDIA GPU and Intel Xeon Phi devices, we presented an overview of subset of platform independent, portable technologies that are central to code development environment and operational readiness of production platforms. While Cray XK7 and potentially a hypothetical Cray XC30 with GPU devices are on target to support the software stack for code development and execution environment, challenges remain in fully supporting integration of these systems into ecosystem and operational environment of an HPC center. We will therefore continue exploring options for completing the missing platform-specific components for an adaptive supercomputing environment both for Cray and non-Cray platforms. Our near term focus is support for OpenACC tool chain and tuned, accelerator aware MPI as well as integration of resource usage and monitoring utilities on the Cray XK7 platform. In addition, we would like to investigate how platform specific code development and execution toolsets can be fully integrated and supported on Cray platforms to allow for efficient migration from standard servers and Linux clusters with accelerator devices.

REFERENCES

[1] http://www.allinea.com/products/ddt/

[2] http://www.allinea.com/news/bid/88837/Allinea-releases-tools-for-Intel-Xeon-Phi-Coprocessor-developers

[3] http://kb.caps-entreprise.com/how-to-i-use-a-xeon-phi-with-caps-compiler/

[4] Cray libsci, http://docs.cray.com/books/S-2310-50/html-S-2310-50/z1026396697smg.html

[5] www.intel.com/xeonphi

[6] Kepler GK110 white paper available from www.nvidia.com

[7] http://www.nvidia.com/object/cuda_home_new.html

[8] http://www.openacc.org/

[9] http://www.openmp.org/mp-documents/TR1_167.pdf

[10] http://www.roguewave.com/products/totalview

[11] http://www.vampir.eu/

[12] Sadaf R. Alam, Jeffrey Poznanovic, Ugo Varetto, Nicola Bianchi, Antonio Penya, Nina Suvanphim. Early experiences with the Cray XK6 hybrid CPU and GPU MPP platform, In Proc. CUG 2012.

[13] Alverson, B., Roweth, D., Kaplan, L.: The Gemini System Interconnect. High Performance Interconnects Symposium. 2010.

[14] Alverson, B., Froese, E., Roweth, D., Kaplan, L.: Cray XC Series Network. Technical report WP-Aries01-112, Cray Inc. 2012.

[15] Faanes, G., Bataineh, A., Roweth, D., Court, T., Froese, E., Alverson, B., Johnson, T., Kopnick, J., Higgins, M., Reinhard, J.: Cray cascade: a scalable HPC system based on a Dragonfly network. Proc. International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12). 2012.

[16] Nick Edmonds, Douglas Gregor, and Andrew Lumsdaine http://www.boost.org/doc/libs/1_53_0/libs/graph_parallel/doc/html/index.html

[17] Harshvardhan, Adam Fidel, Nancy M. Amato, Lawrence Rauchwerger, "The STAPL Parallel Graph Library," In Wkshp. on Lang. and Comp. for Par. Comp. (LCPC), Tokyo, Japan, 2012.

[18] Douglas Gregor, Matthias Troyer. http://www.boost.org/doc/libs/1_53_0/doc/html/mpi.html

[19] Vaughan, C., Rajan, M., Barrett, R., Doerer, Do., Pedretti, K.: Investigating the Impact of the Cielo Cray XE6 Architecture on Scientific Application Codes. Proc. IEEE Int. Symp. on Parallel and Distributed Processing Workshops (IPDPSW '11). 2011.

[20] H. Wang, S. Potluri, M. Luo, A. Singh, S. Sur and D. K. Panda, MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters, Int'l Supercomputing Conf. (ISC), 2011.