Extending the Capabilities of the Cray Programming Environment with Clang-LLVM Framework Integration

Sadaf Alam, Benjamin Cumming and Ugo Varetto
Swiss National Supercomputing Centre (CSCS)
Lugano, Switzerland
alam,cumming,uvaretto@cscs.ch

Abstract— **Recent developments in programming for multi-core processors and accelerators using C++11, OpenCL and Domain Specific Languages (DSL) have prompted us to look into tools that offer compilers and both static and runtime analysis toolchains to complement the Cray Programming Environment capabilities. In this paper we report our preliminary experiences from using the CLang-LLVM framework on a hybrid Cray XC30 to perform tasks such as generating NVIDIA PTX code from C++ and OpenCL in a portable and flexible manner. Specifically we investigate how to overcome some of the limitations currently imposed by the standard tools such as the complete lack of C++11 support in CUDA C and outdated 32 bit versions of OpenCL. We also demonstrate how Clang-LLVM tools, for example, the static analyzer can bring additional capabilities to the Cray environment. Finally we describe how CLang-LLVM integrates with the standard Cray Programming Environment (PE), for instance, Cray MPI, perftools and libraries, and the steps required to properly install such tools on various Cray platforms.**

Keywords-Programming Environment (PE); Cray XC30; GPU; Clang; LLVM (key words); CUDA, OpenCL

I. INTRODUCTION

The Swiss National Supercomputing Centre (CSCS) recently upgraded its Cray XC30 system Piz Daint to accelerator-based nodes to give the fastest Top500 system in Europe as of November 2013. Each node of the system has an eight-core Intel Sandy Bridge processor with and an Nvidia K20x GPU with 14 SMX units each with 192 cores. At the same time, HPC application developers have been actively targeting other technologies such as Intel MIC, which could have over 60 cores. Mainstream CPU technologies such as Intel Ivy Bridge could have up to 10 cores per socket. In short, the amount of concurrency and parallelism has been increasing, and as a result portable programming languages and interfaces targeting these technologies are on the rise. In this paper, we augment the Cray Programming Environment (PE) with a framework called Clang-LLVM to develop portable parallel programs, domain specific languages and libraries [1][5].

Currently there are some challenges to developing portable applications in the Cray PE using OpenCL and C++11:

- OpenCL is a programming language designed for portable computing across different architectures, including GPU and multi-core [7]. However, the NVIDIA toolchain used in the Cray PE only supports 32-bit OpenCL up to version 1.1, with no plans to add support for 64-bit or more recent OpenCL versions, for example, the current 2.0 version.

- C++11 includes extensions to C++ for parallel programming, such as multi-threading and synchronization primitives [4]. In the Cray PE, the GNU Intel compiler toolchains provide comprehensive C++11 language features, yet they have not implemented many useful library features. Furthermore, only the Cray C++ compiler can generate GPU executables using OpenACC [6].

In order to address the above challenges, we target LLVM, which is a collection of modular tools and technologies that offer a framework for developing and reusing front-ends and back-end interfaces for different programming languages and target platforms. The Clang project implements a front-end C and C++ compiler and analyzer that use LLVM for code generation. Clang and LLVM have features that make them very attractive for developing tools that offer portable code generation:

- As of January 2014 the Clang front end has the only complete C++11 language and library support.

- The Clang front-end has native support for OpenCL.

- LLVM has many back ends, including 64-bit NVIDIA PTX and multi-core x86 processors (with support for many other architectures not discussed here).

- There are tools, for example, Clang static analyzer for C++ applications, which can offer code development capabilities that are currently unavailable on the Cray platforms [2].

In the paper we provide details on using the features of the Clang-LLVM toolchain listed above to compile C++11 and OpenCL code for different target architectures that have

LLVM backends. We will show how OpenCL and C++ code can be compiled using Clang-LLVM, and how to use the generated code in standard Cray PE toolchains, including interoperability with Fortran and C/C++ and MPI. We also outline the installation and usage of the Clang/LLVM toolchain within Cray PE. For instance, we demonstrate how existing Cray MPI library, performance tools (perftools) and platform optimized libraries can be used by the Clang-LLVM toolchain.

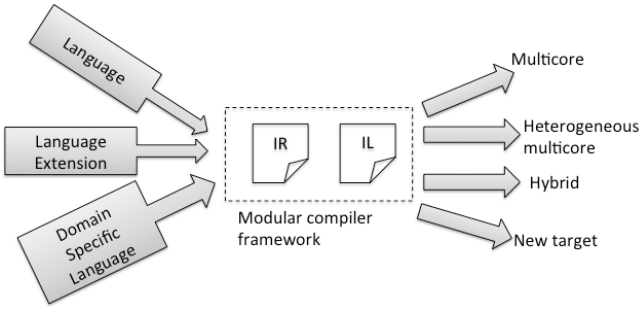

Figure 1: Our target environment for extending the Cray PE framework. A modular framework can accept different languages and language extensions for multicore and hybrid multicore programming. Portability to many platforms can be improved as vendors provide support for standard or commonly agreed Intermediate Representations (IR) or Intermediate Languages (IL).

In fact, this study provides us a proof of concept for supporting heterogeneous computing beyond OpenCL and C++11. As shown in Figure 1, the setup with Clang-LLVM and Nvidia LLVM backend is quite extensible. As new programming paradigms continue to develop for Clang-LLVM and the LLVM backends from vendors such as Intel, Nvidia, AMD, ARM and potentially others, a flexible and extensible code development environment can improve portability of user applications. For instance, Cray could potentially integrate the Clang compiler in its Programming Environment (PE) as done in the past for CCE, gnu, Intel and PGI compilers.

The layout of the paper is as follows: section II provides a brief description of key technologies that are presented in this paper. Step-by-step instructions on installing and configuring Clang-LLVM on the hybrid Cray XC30 platform is presented in section III. Section IV describes examples of code generation for C++11 and OpenCL using the Clang-LLVM on the hybrid Cray XC30 platform. In section V, we discuss work in progress in exploiting Clang code development tools, which are currently not available on the Cray platforms. Finally, a summary and future plans are provided in section VI.

II. KEY TECHNOLOGIES

Presently, there is a rich collection of code development technologies for parallel and hybrid platforms as multi-core processors with large cores counts and embedded systems with GPU devices have become mainstream. In HPC, MPI and OpenMP are the focus of attention. This trend has been changed for GPU based HPC systems, where attention is focused on CUDA. As a result, the current Cray PE integrates a number of OpenMP compilers for C, C++ (not C++11) and Fortran with MPI support. CUDA SDK is integrated in the Cray PE for systems with GPU devices (Cray XK7 and hybrid Cray XC30).

Here we introduce technologies that have a potential for developing portable parallel applications on a range of systems.

A. OpenCL

OpenCL (Open Compute Language) is an API that has been introduced in 2008 for heterogeneous programming paradigm where a GPU or any other accelerator device can work cooperatively with a CPU. One of the key ideas promoted for OpenCL was portability to different platforms. Computation is subdivided and expressed as concurrent tasks, which, according to the workload characteristics, can be then scheduled for GPU, accelerators and CPU resources. OpenCL codes can share resources with OpenGL, a standard for graphics, by allowing for the sharing of memory locations and data structures.

The OpenCL development framework is made up of three main parts:

1. Language specification defines how a kernel program is written, which can then be executed on the OpenCL enabled platforms. The OpenCL programming language is based on the ISO C99 specification with added extensions and restrictions. A new proposed specification targeted at supporting a subset of the C++11 language has recently been approved.

2. The platform-layer API gives the developer access to software application routines that can query the system for the existence of OpenCL-supported devices. This layer also lets the developer use the concepts of device context and work-queue to select and initialize OpenCL devices, submit work to the devices, and enable data transfer to and from the devices.

3. Runtime API: The OpenCL framework uses contexts to manage one or more OpenCL devices. The runtime API uses contexts for managing objects such as command queues, memory objects, and kernel objects, as well as for executing kernels on one or more devices specified in the context.

Further details on the OpenCL architecture, execution and memory models can be found in [AMD OpenCL intro]. OpenCL and CUDA have a number of similarities and some key differences. A tool called Swan can convert existing CUDA applications to OpenCL (http://www.multiscalelab.org/swan).

At CSCS, one of the large-scale GPU-accelerated applications called BigDFT has been developed using OpenCL [12]. This code has been used in production on CSCS Cray XK7 and Cray XC30 platforms. There are a number of development projects that are targeting OpenCL. Since Nvidia froze the support for OpenCL a couple of years ago and only supports the OpenCL 1.1 standard, CSCS is investigating alternate technologies to support users and users application development using OpenCL.

B. OpenCL SPIR (Standard Portable Intermediate Representation)

OpenCL SPIR is a portable binary distribution format for OpenCL programs, which is based on LLVM IR (Intermediate Representation) [8]. Figure 2 shows the flow of SPIR where a vendor can provide an SPIR implementation that could accept an SPIR input format. Vendors have freedom to implement the SPIR standard format on their target platforms.

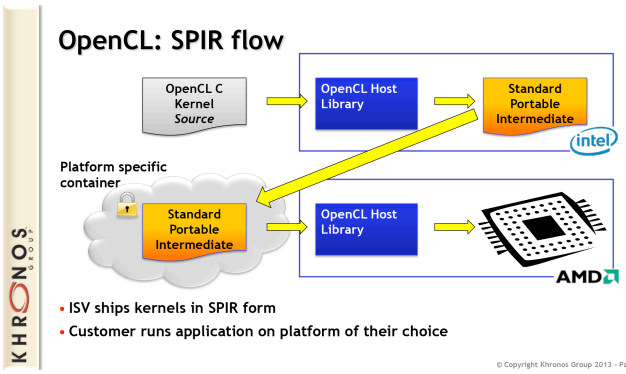

Figure 2: With the introduction of OpenCL SPIR, vendors and kernel developers can share OpenCL implementation in alternate formats [figure courtesy of Khronos group].

SPIR has been designed to address the following:

1. Sharing of kernels in non-string formats
2. On the fly compilation and code generation overhead
3. Flexibility for just in time format for efficient compilation on target platforms

PGI has recently released their OpenACC compiler for the AMD Radeon devices that exploits the SPIR interface, which in turn targets AMD Radeon LLVM backend. Note that PGI OpenACC compiler for the Nvidia GPU devices generate low-level C and CUDA codes.

Other technologies that are similar to SPIR are HSAIL and LLVM IL. The goals are similar, i.e. to attempt to come up with an intermediate, platform independent specification that multiple targets could use. Currently SPIR can be considered as a subset of LLVM IL. Heterogeneous System Architecture (HSA) is a standardization effort that is lead by AMD and other vendors to standardize HAS [3]. The HSA design allows multiple hardware solutions to be exposed to software through a common standard low-level interface layer, called HSA Intermediate Language (HSAIL). HSAIL provides a single target for low-level software and tools. AMD has recently released an APU called Berlin, which will be based on the HSA standard.

C. C++11

One of the major changes to the C++11 standard is multithreading support. Prior to C++11, this was only available via OpenMP and pthreads programming paradigms. The multithreading support in C++11 comes with an implementation of thread class, supporting classes and templates, and a memory and execution model. Supporting functions for memory consistency, for example, mutex, locks, atomics, etc. are available. Clang reached the full C++11 compliance before other compilers, therefore, early development work for C++11 was done with Clang 3.x and LLVM.

An example of a simple C++11 program:

```
#include <iostream>
#include <thread>

using namespace std;

void func(int x) {
    cout << "Inside thread " << x << endl;
}

int main() {
    thread th(&func, 100);
    th.join();
    cout << "Outside thread" << endl;
    return 0;
}
```

D. Nvidia LLVM and libnvvm

Starting from CUDA 4.1, NVIDIA based its CUDA C/C++ compiler on LLVM as well and recently contributed their NVPTX back-end to the LLVM open-source community [9]. The goal is to support language extensions for GPUs as well as additional targets for CUDA language. Nvidia compile SDK contains the following:

- An optimizing compiler library (libnvvm.so, nvvm.dll/nvvm.lib, libnvvm.dylib) and its header file nvvm.h are provided for compiler developers

who want to generate PTX from a program written in NVVM IR, which is a compiler internal representation based on LLVM [10].

- A set of libraries, libdevice.*.bc, that implement the common math functions for devices in the LLVM bitcode format.
- A set of samples that illustrate the use of the compiler SDK.
- Documents for the Compiler SDK (including the specification for LLVM IR, an API document for libnvvm, and an API document for libdevice), can be found under the doc sub-directory, or online.
- The optimizing compiler libraries, the libdevice libraries and samples can be found under the **nvvm** sub-directory, seen after the CUDA Toolkit Install.

Figure 3 shows the layout of the Nvidia compiler toolchain that could enable multiple language targets on GPU devices. NVCC compilation can be mapped on the toolcain. A CUDA application uses CUDA C/C++ "Front End", which is then fed into an LLVM based high level optimizer and PTX generator called CICC. PTX is the virtual instruction set for the Nvidia GPU devices. NVVM IR is a compiler IR (internal representation) based on the LLVM IR. The NVVM IR is designed to represent GPU compute kernels (for example, CUDA kernels). High-level language front-ends, like the CUDA C compiler front-end, can generate NVVM IR. The NVVM compiler (which is based on LLVM) generates PTX code from NVVM IR. libdevice.bc is a set of libraries that implement the common math functions for devices in the LLVM bitcode format.

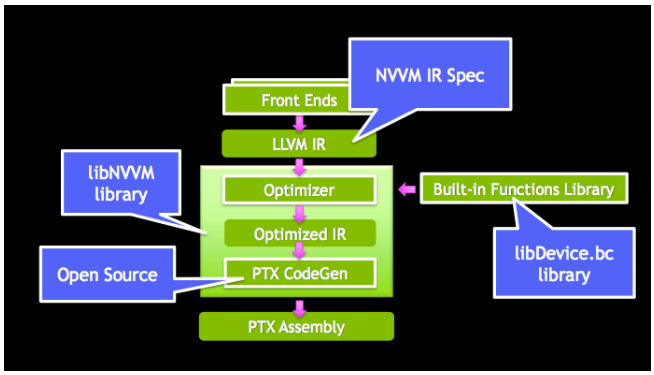

Figure 3: Control flow of Nvidia compiler SDK. CUDA C and C++ compilers are part of the Front End. NVVM IR is a compiler internal representation based on LLVM IR. Additional transformation and tuning steps are performed where LLVM specific technologies are used, for example. libDevice.bc, which is LLVM bitcode implementation of tuned math libraries. PTX instruction set is then finally converted into the machine executable instruction by the target device driver.

III. INSTALLATION AND INTEGRATION OF CLANG-LLVM

A. Overview of Clang-LLVM

LLVM is a modular and reusable compiler framework and toolchain. It used as an infrastructure to implement a broad variety of statically and runtime compiled languages, including the languages supported by GCC, Java, .NET, Python, Ruby, Scheme, Haskell, Julia, and many others. In the GPU computing domain, it has been used for the OpenCL programming language and runtime.

Clang is a compiler front-end for a number of languages and it uses LLVM backend. It has been has been now part of standard LLVM release since version 2.6. Clang supports a number of languages and language extensions. This includes C, C++ and OpenCL. Another benefit of Clang are built-in and extended toolchains. For example, Clang static analyzer can be used for C and C++.

The modularity of the LLVM compiler framework is achieved with a classical three-phase design as shown in figure 4. A front-end compiles a source language into the LLVM Intermediate Representation (IR). In the second phase, multiple analysis and optimization passes operate on this intermediate representation in order to improve the code. Finally, target specific back-ends transform the intermediate representation to another programming language, to assembler code or to machine code for a specific architecture. LLVM IR is a strongly typed low-level instruction set, designed for type, control and data flow analysis, and various code optimization and code restructuring transformations.

Note that the entire toolchain shown in Figure 4 is not part of LLVM. For instance, Nvidia develops CUDA C and C++ compilers. Likewise, Nvidia contributes nvptx backend to LLVM. Hence the CUDA compiler design fits into the classical, three-phase LLVM compiler based model, i.e. a front-end, an optimizer and a backend. Figure 3 also highlights this three-phases in the CUDA SDK.

Similarly, other compilers and processors' vendors, for example, AMD, contribute frontend, backend as well as IR. This three-phase design could be used to mix-and-match frontend, optimizers and backends from multiple vendors. For example, as mentioned in the previous section, for OpenACC directives based language, PGI develops a frontend. It then generates an intermediate form called OpenCL SPIR. AMD, the GPU vendor of the Radeon devices, provides a backend for OpenCL SPIR. Note that PGI OpenACC compilers for Nvidia GPU devices use a different implementation because there is no OpenCL SPIR available from Nvidia. In short, the three-phase design offers a great deal of flexibility for vendors, compiler and tools developers and end users.

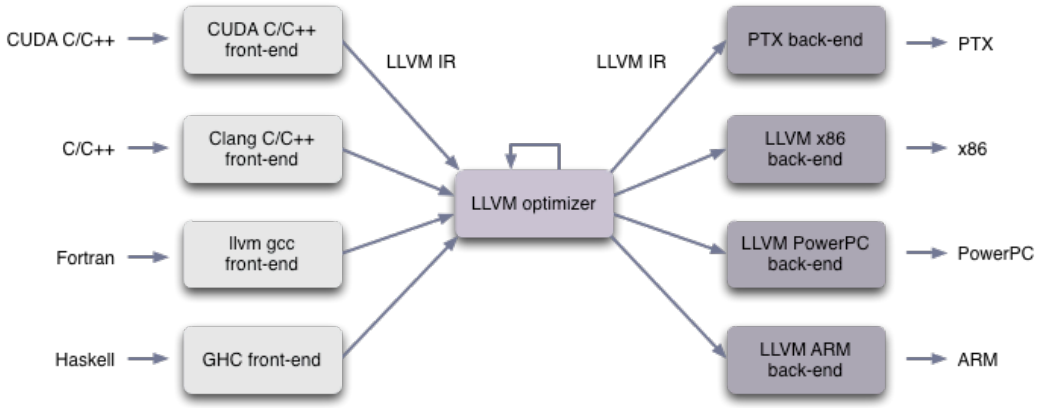

Figure 4: A high level view of the LLVM toolchain in a hybrid CPU and GPU environment. Multiple language targets can utilize the LLVM modular framework and multiple backend codes can be generated for increased portability on a variety of target platforms and devices [figure courtesy of https://www.quantalea.net/]

Code generation process with Clang-LLVM is shown in Figure 5. The frontend processes the input files, for instance, Clang frontend can process different input formats. Front end can parse, diagnose and validate input code. An Abstract Syntax Tree (AST) is then generated. This may be optional. Clang supports multiple source-to-source transformations. Different tools can be written to parse the Clang AST. AST to LLVM IR may comprise several steps as well. From an optimized IR, backend code generator then produces machine code for different targets.

B. Installation instructions

For this paper, Clang-LLVM version 3.4 was installed on a hybrid Cray XC30 platform called Piz Daint, which has an 8-core Intel Xeon CPU and an Nvidia Tesla K20X GPU based compute nodes. Compute nodes have CLE 5.1UP02 and 8.2 was the default programming environment. Step-by-step build instructions are as follows:

```
Clang-LLVM build instructions are available at:
http://clang.llvm.org/get_started.html

======================
get the sources
======================
# download llvm
wget http://llvm.org/releases/3.4/llvm-
3.4.src.tar.gz
tar -xzvf llvm-3.4.src.tar.gz

# download clang
cd llvm-3.4/tools/
wget http://llvm.org/releases/3.4/clang-
3.4.src.tar.gz
tar -xzvf clang-3.4.src.tar.gz
mv clang-3.4 clang

# download clang extra tools
cd clang/tools
wget http://llvm.org/releases/3.4/clang-tools-
extra-3.4.src.tar.gz
tar -xzvf clang-tools-extra-3.4.src.tar.gz
mv clang-tools-extra-3.4 clang-tools-extra

# download compiler rt
cd ../../../projects/
wget http://llvm.org/releases/3.4/compiler-rt-
3.4.src.tar.gz
tar -xzvf compiler-rt-3.4.src.tar.gz
mv compiler-rt-3.4 compiler-rt

# fix for SUSE header files
# fixes a compiler error
vim ./compiler-
rt/lib/sanitizer_common/sanitizer_platform_limits
_posix.cc
------------------------------------------------
---------------
... add the following wrapper around the sys/vt.h
header

#define new CSCSNEWFIX
#include <sys/vt.h>
#undef new
------------------------------------------------
---------------

# back to root path
cd ../..

======================
debug version
======================

module swap PrgEnv-cray PrgEnv-gnu
export CC=`which gcc`
export CXX=`which g++`

mkdir build_debug
cd build_debug

../llvm/ configure --prefix=<installation path> -
-enable-cxx11 --with-gcc-
toolchain=/opt/gcc/4.8.2/snos
make -j 8
```

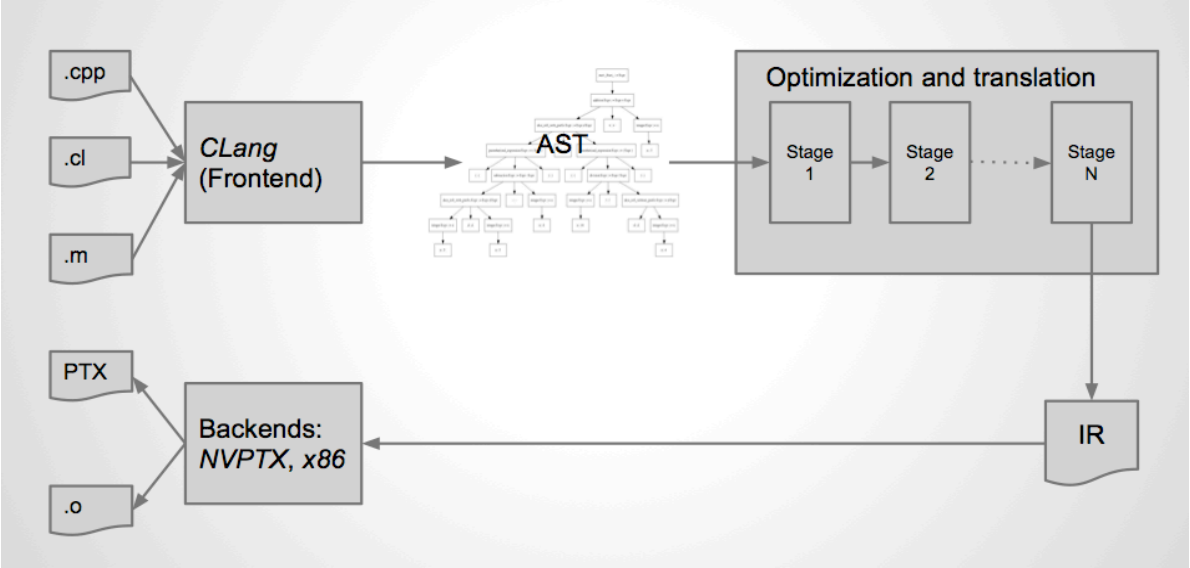

Figure 5: Code generation process with Clang-LLVM.

Build instructions for clang++ STL library, which is fully C++11 compliant and includes all the C++1y features approved so far:

```
> module load PrgEnv-gnu

> echo | g++ -Wp,-v -x c++ - -fsyntax-only

>    wget    http://llvm.org/releases/3.4/libcxx-
3.4.src.tar.gz

# uncompress then create a build directory (e.g.
libcxx-build) outside of the LLVM source tree

> cd <path to libcxx-build>

# Add clang to path and build with cmake

>export
PATH=<llvm path>/llvm-3.4/bin:$PATH

>module load cmake

>CC=clang CXX=clang++ cmake
<install-path> -G "Unix
Makefiles" -DLIBCXX_CXX_ABI=libstdc++
-
DLIBCXX_LIBSUPCXX_INCLUDE_PATHS="/opt/gcc/4.8.2/s
nos/include/g++;/opt/gcc/4.8.2/snos/include/g++/x
86_64-suse-linux"
-DCMAKE_BUILD_TYPE=Debug
-DCMAKE_INSTALL_PREFIX=<install path>

>make -j 8

>make install
```

C. *Integration into Cray environment*

On Piz Daint, four compilers are available as part of the Programming Environment (PE). These include the Cray compiler (CCE), GNU, Intel and PGI. These compilers are available with MPI compiler wrappers, cc for mpicc, CC for mpicxx and ftn for mpif90. In addition to the MPI library, Cray tools called perftools and numerical libraries have been prebuilt for these compiler wrappers.

In this study, we attempt to use the MPI with the Clang compiler. Since the Clang-LLVM is built with the gnu compiler, we tried using the GNU MPI libraries.

```
> clang hello_mpi.c -I
 /opt/cray/mpt/6.1.1/gni/mpich2-gnu/48/include -
L/opt/cray/mpt/6.1.1/gni/mpich2-gnu/48/lib  -
lmpich_gnu_48

> aprun -n 2 ./a.out
   Hello world from process 1 of 2
   Hello world from process 0 of 2
Application 52141 resources: utime ~0s, stime ~0s,
Rss ~3980, inblocks ~100, outblocks ~23
```

We also tried experimenting with the perftools-lite. With the default compile options, the code was not instrumented as expected. However, with the compile options shown below, perftool-lite output for the MPI message was generated.

```
   > clang mt.c -I /opt/cray/mpt/6.1.1/gni/mpich2-
gnu/48/include    -L/opt/cray/mpt/6.1.1/gni/mpich2-
gnu/48/lib          -lmpich_gnu_48          -
I/opt/cray/perftools/6.1.4/include -DCRAYPAT -g -O1
-B      /opt/cray/perftools/6.1.4/libexec64      -
L/opt/cray/perftools/6.1.4/lib64
   WARNING: CrayPat is saving object files from a
temporary      directory      into      directory
'/users/alam/.craypat/a.out/28946'
   INFO:    creating    the    CrayPat-instrumented
executable 'a.out' (sample_profile) ...OK
   WARNING: Can not locate shared object file
'librca.so.0'
   WARNING: Can not locate shared object file
'librca.so.0'

   > aprun -n 2 -N 1 ./a.out
```

```
   CrayPat/X:     Version   6.1.4   Revision   12502
03/10/14 10:11:17
   pat[WARNING][0]:     HW     performance     counter
multiplexing enabled
   Hello world from process 0 of 2
   Hello world from process 1 of 2

   ############################################
#
   #             CrayPat-lite Performance Statistics
#
   ############################################

   CrayPat/X:    Version  6.1.4  Revision  12502  (xf
12277)  03/10/14 10:11:17
   Experiment:                                  lite
lite/sample_profile
   Number of PEs (MPI ranks):       2
   Numbers of PEs per Node:         1  PE on each of
2  Nodes
   Numbers of Threads per PE:       1
   Number of Cores per Socket:      8

   …

   Wall Clock Time: 0.128458 secs
   High Memory:         34.93 MBytes
   I/O Write Rate:       1.24 MBytes/Sec
```

These examples demonstrate that it is feasible to integrate existing Cray tools together with Clang. We used `-craype-verbose` flag to gather Cray wrapper options. Likewise, we were able to do integration of the numerical libraries, for example, MKL. We plan on experimenting with libsci and libsci_acc (GPU accelerated version) in the future.

```
clang -o dgemm-example dgemm-example.c -DMKL_ILP64
-I    $(MNKLROOT)/include/        -Wl,--start-group
$(MKLROOT)/lib/intel64/libmkl_intel_ilp64.a
$(MKLROOT)/lib/intel64/libmkl_core.a
$(MKLROOT)/lib/intel64/libmkl_sequential.a   -Wl,--
end-group -lpthread -lm
```

In short, if Cray were to integrate Clang-LLVM in their PE, most of the existing tools and libraries would be readily functional.

IV. Code Genenration for OpenCL and C++11

The study is primarily motivated by the need to support the latest OpenCL standard and extensions for 64 bit architectures, which is currently not supported by the Nvidia SDK. A further goal was to understand the feasibility of using C++11 features such as multi-threading. In the following subsections, we demonstrate how Clang-LLVM can be used to address both tasks.

A. OpenCL to PTX Conversion and Execution on GPU

In addition to Clang-LLVM (including nvptx support) we use a library called libclc that contains built-in functions for the PTX targets. Detailed instructions are available from http://stackoverflow.com/questions/8795114/how-to-use-clang-to-compile-opencl-to-ptx-code.

Step # 1: Conversion of OpenCL code to LLVM IR using the Clang compiler

```
clang   -Dcl_clang_storage_class_specifiers   -
isystem libclc/generic/include -include clc/clc.h -
target  nvptx64-nvidia-cuda  -xcl  kernel.cl  -emit-
llvm -S -o kernel.ll
```

Step # 2: It is optional if no built-in functions or OpenCL kernel API calls are in the code. In case of built-in functions, we need to link the code to the libclc

```
llvm
link libclc/built_libs/nvptx64

nvidiacl.bc kernel.ll
o kernel.linked.bc
```

Step # 3: from the LLVM IR, the .ll file, we can use llc to generate ptx. Details for generating PTX from LLVM IR are available at http://llvm.org/docs/NVPTXUsage.html. Additional information on how to link with optimized match functions that are available in Nvidia libDevice.bc are available at this link.

```
llc -mcpu=sm_35 kernel.ll -o kernel.ptx
```

With the built-in functions, Clang compiler can be used to generate the PTX assembly:

```
clang -target nvptx64-nvidia-cuda
kernel.linked.bc -S -o kernel.nvptx.s
```

Once we have the PTX code, the next step is to execute it. Like OpenCL code execution, we can use the CUDA driver API for just in time (JIT) compilation. In order to do this, users have to write some code to initialize the device, pass parameters as needed by the kernel, etc. A complete, simple example is available at http://llvm.org/docs/NVPTXUsage.html

Once the driver code is available with necessary device and kernel initialization, the code can be compiled as any other C or C++ code using any C or C++ compiler that is available as part of the Cray PE. We can also use clang C and C++ compiler. CUDA include and runtime library paths must be specified on the compile line.

```
CC    sample.cpp    -o    sample    -O2    -g    -
I/opt/nvidia/cudatoolkit/5.5.20-
1.0501.7945.8.2/include                          -L
/opt/nvidia/cudatoolkit/5.5.20-
1.0501.7945.8.2/lib64/ -lcudart

> aprun ./sample
Using CUDA Device [0]: Tesla K20X
Device Compute Capability: 3.5
Launching kernel
Results:
0 + 0 = 0
1 + 2 = 3
```

```
…
```

B. C++11 code generation and execution on CPU

When building C++(11) code clang does not find the default headers so you have to pass them on the command line.

Step # 1: retrieve the list of standard g++ include search paths:

```
> echo | g++ -Wp,-v -x c++ - -fsyntax-only

/opt/gcc/4.8.2/snos/include/g++
/opt/gcc/4.8.2/snos/include/g++/x86_64-suse-linux
/opt/gcc/4.8.2/snos/include/g++/backward
/opt/gcc/4.8.2/snos/lib/gcc/x86_64-suse-
linux/4.8.2/include
/usr/local/include
/opt/gcc/4.8.2/snos/include
/opt/gcc/4.8.2/snos/lib/gcc/x86_64-suse-
linux/4.8.2/include-fixed
/usr/include
```

Step # 2: add the include files to an environment variable e,g, $CRAY_CLANG_INCLUDE_PATH in a text file like this:

```
>export CRAY_CLANG_INCLUDE_PATH ="-
I/opt/gcc/4.8.2/snos/include/g++..."
```

Step # 3:

```
> source set-clang-include-paths
```

Step # 4: compile (`-pthread` is required when including `<thread>`)

```
> clang++ $CRAY_CLANG_INCLUDE_PATH -std=c++11 -
stdlib=libstdc++ -pthread test.cc -o test.out
```

```
//test.cc


#include <iostream>
#include <thread>

static const int num_threads = 10;

//This function will be called from a thread

void call_from_thread(int tid) {
    std::cout << "Launched by thread " << tid <<
std::endl;
}

int main() {
    std::thread t[num_threads];

    //Launch a group of threads
    for (int i = 0; i < num_threads; ++i) {
        t[i] = std::thread(call_from_thread, i);
    }
    std::cout << "Launched from the main\n";

    //Join the threads with the main thread
    for (int i = 0; i < num_threads; ++i) {
        t[i].join();
    }

    return 0;
}
```

C. C++11 code generation and execution on CPU

In this case, we use clang++ with libc++.

```
> clang++ -I
<libc++ install path>/include/c++/v1
-L <libc++ install path>/lib
-std=c++11 -stdlib=libc++ -pthread test.cc -o
test.out
```

In order to run your executable you have to have the libc++ library in the linker path:

```
> export
LD_LIBRARY_PATH=<libc++ install
path>/lib:$LD_LIBRARY_PATH
```

If you put libc++.so in a place reachable from the compute nodes you should be able to run through aprun without problems, although the best solution would be to install it on the compute nodes themselves

V. TOOLS IN CLANG-LLVM

The Clang project uses a library-based architecture, with the different components of the front end maintained in separate libraries. When combined with the clear coding style of the Clang/LLVM project, these libraries are of ideal use and extended by tool developers. Tools based on these libraries can be divided into two categories: tools for developing C/C++ code (e.g. refactoring tools, code browsers and code completion), and testing/debugging tools (e.g. static analysis, memory leak/corruption testing).

Here we will look at some of these tools, though not all of them. We note that the memory and thread sanitizers have also recently become available in the GNU tool chain (as of version 4.8). However, there appears to be many more such tools being developed with Clang, most likely due to the ease of developing such tools using the Clang and LLVM libraries. As these libraries mature, we expect to see a very strong Clang-based tool ecosystem develop.

A. Developer tools

Because of the complexity of the C++ language, even tools that perform relatively "simple" tasks like reformatting require that the C++ code be parsed to perform accurately. More complicated tasks such as code refactoring, code

completion and highlighting potential errors while typing require compilation.

Here we give a quick overview of some of the tools that make developers lives easier on Cray systems, giving access to features that are normally associated with complex IDEs. Each of these tools can be used as the basis for adding such features to text editors, so that, for example, the Vim editor has wrappers for each tool.

ClangFormat is a set of tools for formatting C and C++ code. It can be used as a standalone command line tool, or integrated into editors (both vim and emacs require adding two lines to editor configuration files to integrate the tool).

ClangCheck is a small tool that can be used to perform basic error and warning checks on code snippets. For example

```
> cat test.c
void foo() { int a = 3 }
$ clang-check test.c --
/scratch/santis/bcumming/test.c:1:23: error:
expected ';' at end of declaration
void foo() { int a = 3 }
                      ^
```

ClangModernize is a tool for automatically converting C++ code to C++11, with support for features such as converting for loops to range-based loops and using the auto keyword.

B. Other Development Tools

The Clang toolchain makes it possible to provide tools such as code completion and refactoring, which are normally only found in large editors like Visual Studio, to text editors like vim and emacs. These tools are very important for programmer productivity, and while it is possible to use full-featured IDEs such as Netbeans and Eclipse, these are heavy and slow, and can be difficult to configure for working remotely on servers.

An example of such a project is YouCompleteMe (`github.com/Valloric/YouCompleteMe`), a plugin for Vim that performs code completion while typing. For C and C++ code it compiles the project in the background, along with the AST for all other relevant files in the project. With this information in memory, it can perform context-sensitive name lookup, accurately jump to the definition of types, variables and functions.

The default version of Vim (7.2) currently installed on Cray systems is not recent enough, requiring that Vim be compiled from source with Python support.

C. Testing and Debugging Tools

The Clang toolchain also has a set of tools, some of which are standalone tools, and others which are enabled via compiler flags, that facilitate proactive testing and debugging during the development process.

1) Clang Static Analyzer

The Clang Static Analyzer performs analysis of C/C++ code at compile time to automatically find bugs without running the code. It is compiled automatically when Clang is built, though it is not installed with "make install", which requires an additional step of copying the static analyzer into the clang bin path.

The static analyzer is simple to use from the command line. It provides the scan-build utility, and can by used with a make file as follows:

```
$ scan-build make
```

The scan-build utility over rides the CXX and CC environment variables to first compile each file (either clang or gcc), then executes a static analyzer to test the code. With this process, at the end of compilation the compilation will have been performed as usual, and a report will be generated with the results of the static analysis.

As an example, take the following buggy code, which produces a segmentation fault by dereferencing a NULL pointer

```
int main(void) {
    int *ptr = nullptr;
    ptr[10]++;
}
```

Running the scan-build on a simple makefile for this code produces the following message in the terminal window:

```
test.cc:3:5: warning: Array access (from variable
'ptr') results in a null pointer dereference
    ptr[10]++;
        ^~~~~~~~
1 warning generated.
scan-build: 1 bugs found.
```

Along with a HTML report, for a more detailed overview of the analysis that can be viewed in a web browser, shown in Figure 6.

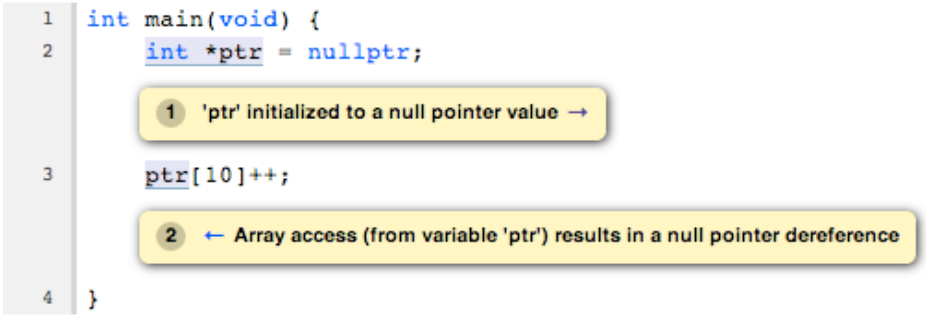

Figure 6: Sample output from the Clang static analyzer viewed in a web browser.

2) *Sanitizers*

The Clang compilers implement a set of "sanitizers", which are runtime tools for detecting memory and threading issues. The **AddressSanitizer** and **MemorySanitizer** features detect memory bugs, including but not limited to, out of bounds access, use after free, uninitialized reads and double free. The **ThreadSanitizer** detects common threading errors, such as race conditions. The sanitizers use compiler instrumentation, activated using compiler flags (e.g. -fsanitize=memory for MemorySanitizer), and custom runtime libraries that require linking with Clang instead of ld.

VI. Summary and Future Directions

In this report, we demonstrated extensibility to the Cray PE with the Clang-LLVM compiler framework for compiling OpenCL and C++11 applications. Our experiments show that Clang can be integrated into the Cray PE like any other compiler. This would be beneficial for advanced C++ developers, OpenCL users and other emerging languages and technologies that are widely supported by Clang-LLVM. Also, this framework offers additional tools for C++ analysis, currently not available on Cray systems. In short, Cray PE can significantly benefit from Clang-LLVM integration to the existing PE. Several Domain Specific Languages (DSLs) are implemented using LLVM. Details on how LLVM is currently used for developing a compiler framework for DSLs are available in [11]. DSL or Domain Specific Embedded Languages (DESL) could offer high-level abstractions that could potentially improve code developer productivity by offering code and performance portability across multiple targets. We plan on investigating emerging languages for heterogeneous computing using the LLVM framework. We also plan on exploring options for offering OpenCL compilation environment to the end users.

References

[1] Clang—a C language family frontend for LLVM: http://clang.llvm.org/

[2] Clang static analyzer: http://clang-analyzer.llvm.org/

[3] HSA (Heterogenous System Architecture) foundation and tools: http://www.hsafoundation.com/hsa-developer-tools/

[4] JTC1/SC22/WG21–The C++ Standards Committee: http://www.open-std.org/jtc1/sc22/wg21/

[5] The LLVM Compiler Infrastructure: http://llvm.org/

[6] OpenACC—directives for accelerators: http://www.openacc-standard.org/

[7] OpenCL: The open standard for parallel programming of heterogenous systems: https://www.khronos.org/opencl/

[8] SPIR: The Standard Portable Intermediate Representation for Device Programs: http://www.khronos.org/spir

[9] User guide for NVPTX backend: http://llvm.org/docs/NVPTXUsage.html

[10] Building compilers with libNVVM: presentation at GTC 2013. Available from: http://on-demand.gputechconf.com/gtc/2013/presentations/S3185-Building-GPU-Compilers-libNVVM.pdf

[11] Implemeting Domain-Specific Languages with LLVM: https://archive.fosdem.org/2012/schedule/event/400/99_DSLsWithLLVM.pdf

[12] Wen-mei W. Hwu. *GPU Computing Gems Emerald Edition* (1st ed.). Morgan Kaufmann Publishers USA, 2011.