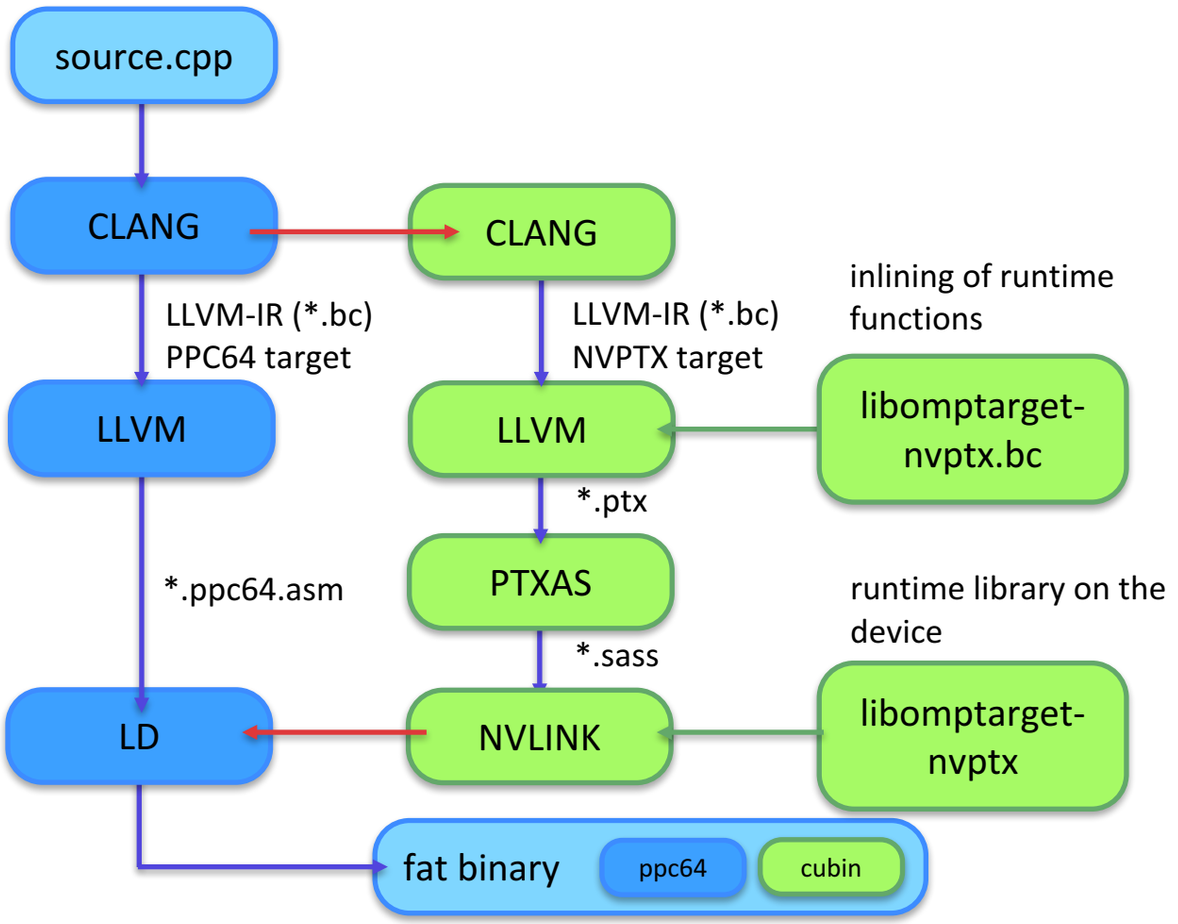
Using OpenMP 4.5 in the CLANG/LLVM compiler toolchain

January 2017, Oak Ridge National Laboratory

Gheorghe-Teodor Bercea
IBM TJ Watson Research Center
Yorktown Heights, NY

The CLANG compiler toolchain



- CLANG takes as input a **source file** and produces a fat binary which contains the compiled code for the **host** and **device**.
- On OpenPower the:
  - **host** is a Power8 or Power8' CPU
  - **device** is an NVIDIA K40 GPU or a P100 GPU (Pascal)
- CLANG can be used to compile:
  - plain C/C++ code (host)
  - CUDA code (host + device)
  - **C/C++ code containing OpenMP 4.5 directives** (host + device)

CLANG compiler location

CLANG module:

```
clang/2.9.cuda8
```

Availability on ORNL system:

```
/sw/summitdev/llvm/20161021/clang/0.2.9.cuda8/bin/clang++
```

CLANG options

- CLANG typically supports all GCC compiler flags.
- Compiling C++ files is similar to GCC:

```
clang++ a.cpp -o a.out
```

- Compiling C++ with OpenMP for **host only**:

```
clang++ a.cpp -fopenmp -o a.out
```

- Compiling C++ with OpenMP for **host and device**:

```
clang++ a.cpp -fopenmp –fopenmp-targets=nvptx64-nvidia-cuda -o a.out
```

- On ORNL systems, CLANG with device offloading is enabled by loading modules:

```
clang/2.9.cuda8 cuda/8.0.44
```

- Additional compiler flag for ORNL system:

```
––cuda-path=${OLCF_CUDA_ROOT}
```

Setting up the OpenMP runtime

- OpenMP header can be passed as a compiler option:

```
-I<clang_install_path>/omprtl
```

OR

```
export C_INCLUDE_PATH=<clang_install_path>/omprtl
export CPLUS_INCLUDE_PATH=<clang_install_path>/omprtl
```

- OpenMP library:

```
-L<clang_install_path>/omprtl/lib
```

OR

```
export LIBRARY_PATH=<clang_install_path>/omprtl/lib
export LD_LIBRARY_PATH=<clang_install_path>/omprtl/lib
```

- On ORNL systems:

```
-I${CLANG_OMP_INCLUDE}
-L${CLANG_OMP_LIB}
```

LOMP

- The default runtime delivered with IBM's CLANG compiler is LOMP (Lightweight OpenMP).
- LOMP is a proprietary code which aims to dramatically reduce overheads.
- LOMP is available here:

```
$CLANG_OMP_LIB
```

- A debug version of LOMP is also available here:

```
$CLANG_OMP_PATH/lib-debug
```

  - The additional output generated when using the debug version of LOMP logs every action performed by the runtime library.

Runtime variables

- OpenMP parallelism uses *teams* and *threads* the number of which can be adjusted using:
  - clauses - on the `teams` construct

    `num_teams (teams construct)`

    `thread_limit (teams construct)`
  - using runtime environment variables:

    `OMP_TEAMS_LIMIT [non-standard]`

    - set the maximum number of teams

    `OMP_NUM_TEAMS [non-standard]`

    - set the default number of teams

    `OMP_NUM_THREADS`

    - set the number of OpenMP threads

    `XLSMPOPTS='TARGETTHREADLIMIT=num'`

    - set the number of threads if thread_limit is not specified

    `XLSMPOPTS='TARGETNUMTHREADS=num'`

    - set the default number of threads

    `XLSMPOPTS='TARGET=MANDATORY | DISABLED | OPTIONAL'`

    - force running on the device, disable running on the device, run if possible

Modules on ORNL system

```
-bash-4.2$ module available

---------------------------- /sw/summitdev/modulefiles/compiler/llvm/20161021 -----------------------------
   essl/5.5.0                   spectrum_mpi/10.1.0      spectrum_mpi/10.1.0.2-20161130 (L,D)
   essl/5.5.0-20161110 (L,D)    spectrum_mpi/10.1.0.2

---------------------------------- /sw/summitdev/modulefiles/core ----------------------------------
   DefApps                  (L)    gcc/4.8.5            (D)    pgi/16.5              xalt/0.7.5        (L)
   clang/2.9.cuda8          (L)    gcc/5.4.0                   pgi/16.7              xl/160617c
   cmake/3.6.1                     gcc/6.2.1-20161006          pgi/16.9              xl/161005
   cuda/8.0.35                     gcc/6.2.1-20161021          pgi/16.10        (D)  xl/20161117
   cuda/8.0.44              (D)    gcc/6.2.1-20161129          ppedev/2.3.0          xl/20161123       (D)
   darshan/3.1.0                   git/2.9.3                   python/2.7.12         xlflang/20160609
   darshan/3.1.2            (D)    hsi/5.0.2.p5         (L)    python/3.5.2     (D)  xlflang/20161130
   enzo/6.0.9                      htop/2.0.2                  scorep/3.0            xlflang/20161206  (D)
   forge/6.1.1-spectrum-mpi        makedepend/1.0.5            tmux/2.2
   forge/6.1.1-38291384f15c (D)    mercurial/3.9.1             vim/8.0

------------------- /sw/summitdev/lmod/6.5.3/rhel7.2_gnu4.8.5/lmod/lmod/modulefiles/Core ------------------
   lmod/6.5.3    settarg/6.5.3
```

ORNL environment after executing command `module load clang/2.9.cuda8 cuda/8.0.44`:

```
CLANG_LIB=/sw/summitdev/llvm/20161021/clang/0.2.9.cuda8/lib
OLCF_CLANG_ROOT=/sw/summitdev/llvm/20161021/clang/0.2.9.cuda8
CLANG_OMP_PATH=/sw/summitdev/llvm/20161021/clang/0.2.9.cuda8/omprtl
CLANG_PATH=/sw/summitdev/llvm/20161021/clang/0.2.9.cuda8
CLANG_OMP_INCLUDE=/sw/summitdev/llvm/20161021/clang/0.2.9.cuda8/omprtl
CLANG_BIN=/sw/summitdev/llvm/20161021/clang/0.2.9.cuda8/bin
CLANG_INCLUDE=/sw/summitdev/llvm/20161021/clang/0.2.9.cuda8/include
CLANG_VERSION=0.2.9.cuda8
CLANG_OMP_LIB=/sw/summitdev/llvm/20161021/clang/0.2.9.cuda8/omprtl/lib

OMPI_FC=xlflang
OMPI_DIR=/sw/summitdev/spectrum_mpi/10.1.0.2.20161130
OMPI_CC=/sw/summitdev/llvm/20161021/clang/0.2.9.cuda8/bin/clang
OMPI_CXX=/sw/summitdev/llvm/20161021/clang/0.2.9.cuda8/bin/clang++
```

Setting up the environment

```
> cat new.env
CLANG_EXEC_FOLDER=$CLANG_BIN
export DEVRTLS=$CLANG_LIB
export LD_LIBRARY_PATH=$CLANG_OMP_PATH/lib:$LD_LIBRARY_PATH
export LIBRARY_PATH=$DEVRTLS:$LIBRARY_PATH
export PATH=${CLANG_EXEC_FOLDER}:$PATH
export OMPI_CC=${CLANG_EXEC_FOLDER}/clang
export OMPI_CXX=${CLANG_EXEC_FOLDER}/clang++
```

Compiling a.cpp

```
> module load clang/2.9.cuda8 cuda/8.0.44

> . new.env

> clang++ -O3 -I${CLANG_OMP_INCLUDE} -fopenmp –fopenmp-
targets=nvptx64-nvidia-cuda -L${CLANG_OMP_LIB} ––cuda-
path=${OLCF_CUDA_ROOT} a.cpp -o a.out
```

Debugging advice

- When first porting an application to OpenMP 4.5 ensure that the application first runs on **host**.
- Once correct results are achieved on host, attempt the offloading of the kernels one by one running the application after every new offloaded region.
- To check that a kernel has been offloaded use **nvprof:**

```
nvprof ./a.out arg1 arg2
```

  or you can run with **-v** and check the NVLINK report.

- Tools which might help with debugging: **gdb** (host) or **cuda-gdb** (on device but limited features available).

```
gdb ./a.out

gdb-prompt$> r arg1 arg2
```

- To see all intermediate compilation stages add the flags to the compile line:

```
-v -save-temps
```

- In case of errors re-run the last command under **gdb**.

```
gdb clang++

gdb-prompt$> r <lots of args> a.bc
```

Register counts (for NVIDIA devices)

- Each GPU thread requires a number of allocated registers. Compiling with

  `-v`

  will print the number of registers per thread used by each kernel:

```
nvlink info    : Function properties for '__omp_offloading_802_1bc227f__ZL23InitStressTermsForElemsR6DomainPdS1_S1_i_1413':
    nvlink info    : used 18 registers, 0 stack, 896 bytes smem, 376 bytes cmem[0], 0 bytes lmem
```

Name of enclosing function

Line number of the kernel, in this case line 413

- The number of registers per thread is capped, by default, to 64. To impose a different limit the following compiler flag is required:

  `-Xcuda-ptxas -maxrregcount=128`

- The number of registers may affect **occupancy** and can therefore make a difference in terms of performance.

Compiling Fortran code with `xlflang`

- `xlflang` is a wrapper for the invocation of:
  - XLF's Fortran Front End (FE)
  - a translator that transforms the output of XLF FE to input to LLVM.
  - LLVM
- Consequently, `xlflang` **forwards user-level compilation flags to LLVM** except some specific flags used by the XLF FE and the translator.

- To use the compiler run the following command first:

```
module load clang/2.9.cuda8 cuda/8.0.44  xlflang/20161206
```

- Additional compiler flag for ORNL system:

```
--cuda-path=${OLCF_CUDA_ROOT}
```

- Make sure that previous `new.env` script also has following path:

```
export LD_LIBRARY_PATH=<path to xlf>/lib:$LD_LIBRARY_PATH
```

- The runtime setup is similar to CLANG.

- Debug by using the debug version of LOMP or by invoking the LLVM AST-print functionality:

```
ASTPRINT=1 xlflang a.f
```

Device offloading performance checklist

- Data mapping (C/C++ & Fortran): reduce number of data transfers between host and device. OpenMP mapping constructs such as **`target enter/exit data, target update, declare target`** keep data on device from one target region to another unless the user advises otherwise.
- Inlining of runtime functions (C/C++): `libomptarget-nvptx.bc` muct be in `${CLANG_OMP_LIB}` folder.
- Use combined constructs wherever possible to produce CUDA-like code (C/C++):
  - **`#pragma omp target teams distribute parallel for`**
- Make sure enough parallelism is available: use the **`collapse(k)`** clause to collapse **`k`** perfectly nested loops and increase the number of parallel iterations (C/C++ & Fortran).
- Mapping loop iterations to *threads* on host vs. device using the **`schedule`** clause (C/C++ & Fortran). Defaults:
  - **on host:** *chunked* schedules increase **locality**;
  - **on device:** chunk size of 1 ensures **coalesced memory accesses**.

Device offloading performance checklist

- Thread binding on host (C/C++ & Fortran).
- On device, choose the number of teams and threads (C/C++ & Fortran). Setting the number of *teams* and *threads* using the **`num_teams(K1)`** and **`thread_limit(K2)`** clauses can be used to customize individual loops.
- Use OpenMP support for parallel reductions (C/C++).
- If the device mapped data does not overlap, the compiler flag **`-fopenmp-nonaliased-maps`** will favour usage of faster non-coherent loads (C/C++).
- CLANG compiler flag: **`-ffp-contract=fast`** uses fuse multiplies and adds wherever profitable, even across statements. Doesn't prevent PTXAS from fusing additional multiplies and adds (C/C++).

Performance outcomes

- LULESH:
  - ALE3D hydrodynamics engine proxy application (LLNL).
  - Comparison against CUDA version. Kernels are represented by single loops offloaded to an NVIDIA K40 GPU using:

    ```
    #pragma omp target teams distribute parallel for
    ```

    - Schedule has chunk size 1 for memory access coalescing.

    - Custom numbers of teams and threads for each loop.

    - Avoid redundant data movement between host and device inside the timestepping loop.
  - For kernels below the 64 registers per thread threshold we achieve performance comparable to CUDA code. We also show that for larger kernels, the performance is similar to CUDA code if register pressure is not a problem.
- Cardioid:
  - Cardioid is an explicit solver for a reaction-diffusion equation (focus on the reaction part)
  - Offloading one of the data parallel kernels of, the Rush-Larsen kernel, achieved:

    50% of double-precision floating point peak on a K80 NVIDIA GPU.

    46% of double-precision floating point peak on a P100 (Pascal) GPU.

Thank you