Programming Models for OpenPOWER Systems

Productive, Portable, High-Level GPU Acceleration

Tarique Islam (tislam@ca.ibm.com)
Software Developer - IBM XL Compilers

Wael Yehia (wyehia@ca.ibm.com)
Software Developer - IBM XL Compilers
XL Compiler Suite

- IBM XL C/C++ for Linux, V13.1.6 and IBM XL Fortran for Linux, V15.1.6
- Architectures: IBM POWER8 and POWER9
- Language support:
  - C11, C++11, and partial C++14
  - Fortran 2003, and partial F2008
  - OpenMP 3.1, most of OpenMP 4.5, some TR6
  - CUDA Fortran

Compiler Reference: [http://ibm.biz/XLC1316](http://ibm.biz/XLC1316)
[http://ibm.biz/XLFortran1516](http://ibm.biz/XLFortran1516)
How to use

- OpenMP programs: > xlC_r -qsmp=omp [-qoffload]
  > xlf_r -qsmp=omp [-qoffload]
- CUDA Fortran: > xlf_r -qcuda (or just xlcuf)
- CUDA Fortran + OMP: > xlf_r -qcuda -qsmp=omp -qoffload

Options:
- Best performance: -Ofast
- Debugging OpenMP: -qsmp=noopt -g -qfullpath
- To get line number info only: -g1 -qfullpath
- Offloading device arch: -qtgtarch=[auto|sm_35|sm_60|sm_70]

-qoffload enables GPU offloading of OpenMP target regions

/qtgtarch is needed if you are cross compiling or the sm level changed after the compiler was installed.
Debugging and Profiling

• Tools:
  - cuda-gdb
  - cuda-memcheck
  - nvprof
  - nvvp

• Debugging Tips:
  - Reduce testcase size.
  - Reduce parallelism: num_threads clause for parallel region, and thread_limit and num_teams clauses for teams region.
  - Reduce array sizes to rule out memory limit issues.
  - CUDA Fortran only: -qcudaerr=all to check return codes from all CUDA API calls.
  - Fortran only: -qcheck to check for array bounds.
  - Control per-device malloc limit by compiling with -qcuda option and exporting XLCUF_GPU_DATA_LIMIT=bytes.
  - Use -qport=c_loc if you get errors about C_LOC and the missing TARGET attribute.
  - Use printf (C/C++) and print (Fortran).
Fortran support and CUDA interop
CUDA C/C++

- Use XL C/C++ as host compiler for POWER CPUs when using nvcc:
  - Fully leverage advanced compiler optimizations for POWER
  - Sample invocation
    > nvcc -ccbin x1C_r -Xcompiler -Ofast t.cu

- Interoperability with OpenMP
  - Use is_device_ptr to access CUDA-allocated device memory in OpenMP
  - Use use_device_ptr to access OpenMP allocated device memory in CUDA
  - Calling OpenMP procedures from CUDA and vice-versa is not supported.
Fortran Language Support

- XL Fortran supports
  - Most of Technical specification 29113: interoperability of assumed-length, assumed-rank, assumed-type, allocatable, pointer, and optional arguments
  - Partial Fortran 2008: submodules, do concurrent, contiguous attribute, BLOCK construct, enhancement to ALLOCATE, STOP/ERROR STOP
  - Full Fortran 2003.

Reference: [https://ibm.biz/Fortran2008Status](https://ibm.biz/Fortran2008Status)  
[https://ibm.biz/FortranTS29113Status](https://ibm.biz/FortranTS29113Status)
CUDA Fortran support

- XL Fortran supports most of CUDA Fortran, with CUF kernels being the main missing feature.
- We intend our CUDA Fortran support to be fully compatible with PGI's, and we test our compiler against the PGI test suite to ensure compatibility.
- More details on the known issues and limitations with our CUDA Fortran implementation can be found at: http://ibm.biz/XLCUF_Limitations
CUDA Fortran and OMP

- A Fortran application can include both CUDA Fortran and OpenMP.
- CUDA Fortran variables can appear on OMP clauses.

```fortran
program p
  integer, parameter :: n = 1000000
  integer i
  integer, allocatable, pinned :: arr_p(:)
  integer, allocatable, managed :: arr_m(:)

  allocate(arr_p(n), arr_m(n))
  arr_m = 0

  !$omp target teams distribute parallel do map(from: arr_p) is_device_ptr(arr_m)
    do i = 1, n
      arr_p(i) = arr_m(i) + 1
    end do
end program p
```
Good Programming Patterns
Good Programming Patterns for OpenMP

- Compiler has a generic (runtime-library-dependent) and an SPMD (little to no runtime) codegen schemes.
- SPMD scheme is the closest to a CUDA kernels with least runtime calls.
- Help the compiler generate “SPMD” programs by:
  - Calls to unknown functions (definition not in CU) causes generic codegen.
  - Give the compiler inlining opportunities, e.g. making sure hot functions are defined in same compilation unit (CU) as their call sites.
  - Use `-qinline+<function-name>` to force inlining (mangled name for C++). For example `-qinline+foo` or `-qinline+_Z3fooi`
**Compiler-friendly Code pattern**

- Compiler can generate better code when functions can be inlined in an OMP TARGET construct. For inlining to happen, caller and callee must be in the same compilation unit.
- The following code-pattern prohibits SPMD code generation.

```fortran
module m
  integer, parameter :: N = 10
  contains
  subroutine mod_sub(x, y, z)
    integer :: x, y, z
    !$omp declare target
    z = 24 * x + y
  end subroutine
end module m

use m
integer :: x(N), y(N), z(N)
x = 10
y = 20
!$omp target teams distribute parallel do map(to: x, y) map(from: z)
do i = 1, N
  !$omp declare target
  z = 24 * x(i) + y(i)
call mod_sub(x(i), y(i), z(i)) ! This call can not be inlined
end do
end
```
Compiler-friendly Code pattern

- Placing caller and callee in the same module allows SPMD code generation.

```fortran
module m
  integer, parameter :: N = 10
  contains
  subroutine mod_sub(x, y, z)
    integer :: x, y, z
    !$omp declare target
    z = 24 * x + y
  end subroutine
  subroutine driver(x, y, z)
    integer :: x(N), y(N), z(N)
    !$omp target teams distribute parallel do map(to: x, y) map(from: z)
    do i = 1, N
      call mod_sub(x(i), y(i), z(i)) ! This call can be inlined
    end do
  end subroutine
end module m
```

```fortran
use m
integer :: x(N), y(N), z(N)
x = 10
y = 20
call driver(x, y, z)
end
```
Good Programming Patterns for OpenMP (Part 2)

- Use good-coalescing access patterns:
  - Use static schedule with chunk size of 1 for `distribute` and `do` loops.
  - Mind the loop order (example in next slide)
- Tune grid and block size using the `num_teams` and `thread_limit` clauses.
- Team private/local variable are put into shared memory by the compiler.
Good Programming Patterns for OpenMP (Part 2)

```c
int ar[2][3];
#pragma omp target teams thread_limit(4) num_teams(1)
    // ar in memory:
    {
        #pragma omp parallel for // i=0,1 distributed
        for (int i = 0; i < 2; ++i) // among 4 threads
            for (int j = 0; j < 3; ++j) //
                ar[i][j] += x; //
                    t0  t0  t0  t1  t1  t1
        
        #pragma omp parallel for // j=0,1,2 distributed
        for (int j = 0; j < 3; ++j) // among 4 threads
            for (int i = 0; i < 2; ++i) //
                ar[i][j] += x; //
                    t0  t1  t2  t0  t1  t2
        
        #pragma omp parallel for \ // [i,j]=[0,0],[0,1],[0,2]
            collapse(2) // [1,0],[1,1],[1,2]
        for (int i = 0; i < 2; ++i) // distributed among
            for (int j = 0; j < 3; ++j) // 4 threads
                ar[i][j] += x; //
                    t0  t1  t2  t3  t0  t1
        
        #pragma omp parallel for \ // [j,i]=[0,0],[0,1],[1,0]
            collapse(2) // [1,1],[2,0],[2,1]
        for (int j = 0; j < 3; ++j) // distributed among
            for (int i = 0; i < 2; ++i) // 4 threads
                ar[i][j] += x; //
```
Fortran Array-access

- Accessing assumed-size and deferred-shape arrays involves overhead.
  - In general, bounds are not known at compile-time.
  - Bounds and storage of a deferred-shape array can change at runtime.
- Compiler can generate better code for assumed-shape arrays in case they are known to be contiguous at compile-time.

```fortran
subroutine zaxpy_explicit_shape(start, end, len, x, y, z)
  integer(kind=8), intent(in) :: start, end, len
  real(kind=8), intent(in) :: x(len), y(len)
  real(kind=8), intent(out) :: z(len)
  integer(kind=8) :: i

  !$omp target teams map(to: x, y) map(from: z)
  !$omp distribute parallel do
  do i = start, end
    z(i) = 24 * x(i) + y(i)
  end do
  !$omp end target teams
end subroutine zaxpy_explicit_shape
```

```fortran
subroutine zaxpy_deferred_shape(start, end, x, y, z)
  integer(kind=8), intent(in) :: start, end
  real(kind=8), intent(in), contiguous :: x(:), y(:)
  real(kind=8), intent(out), contiguous :: z(:)
  integer(kind=8) :: i

  !$omp target teams map(to: x, y) map(from: z)
  !$omp distribute parallel do
  do i = start, end
    z(i) = 24 * x(i) + y(i)
  end do
  !$omp end target teams
end subroutine zaxpy_deferred_shape
```
Questions
Extra slides (not part of presentation)
Compiler architecture

**XL C/C++ Frontend**
- CPU code is aggressively optimized for POWER
- CUDA Toolkit optimizes device code
- XL’s optimizer sees both host and device code

**XL Fortran Frontend**
- W-Code (XL IR)
- High-Level Optimizer
  - Data flow, loop, other optimizations
- CPU/GPU W-Code Partitioner
- CPU W-Code
- POWER Low-level Optimizer
  - Low-level Optimizations
  - Register Allocation + Scheduling
  - POWER Code Generation

**Libraries**
- XL C/C++ Libraries
- XL Fortran Libraries
- CUDA Device Libraries
- CUDA Runtime
- CUDA Driver

**System Linker**
- Executable for POWER/GPU system

**W-Code to LLVM IR translator**
- LLVM Optimizer
- PTX CodeGen
- PTX Assembler
- nvlink

**nvlink**
- PTX
- libNVVM

**NVVM**
- LLVM with NVIDIA enhancements

**XL Device Libraries**

**POWER Low-level Optimizer**
- CPU/GPU W-Code
- Register Allocation + Scheduling
- POWER Code Generation

**High-Level Optimizer**
- Data flow, loop, other optimizations

**XL’s optimizer sees both host and device code**
OpenMP for everything

- Extracting maximum performance:
  - To program a GPU: you have to use CUDA, OpenCL, OpenGL, DirectX, Intrinsics, C++AMP, OpenACC.
  - To program a host SIMD unit: you have to use Intrinsics, OpenCL, or auto-vectorization (possibly aided by compiler hints)
  - To program the CPU threads, you might use pthreads, C/C++11, OpenMP, TBB, Cilk, Apple GCD, Google executors

- With OpenMP 4.0:
  - You can use the same standard to program the **GPU**, the **SIMD** units, and the **CPU** threads.
OpenMP program mapping to CPU+GPU hardware

Generally speaking:

OpenMP League of teams => CUDA grid
OpenMP team => CUDA block
OpenMP Thread * => CUDA thread
OpenMP SIMD lane => CUDA thread

*Nested parallel regions do not spawn new threads; instead work sharing occurs among the CUDA threads in our OpenMP impl.
OpenMP and CUDA: is_device_ptr

- Use **is_device_ptr** to pass device allocated memory (e.g. from cudaMalloc) to an OpenMP target region
- Also useful for passing unified memory pointers to a target region

```c
// cuda_alloc.cu
int * AllocAndInitialize(int init, int length) {
    int *d_data;
    cudaMalloc(&d_data, length * sizeof(*d_data));
    InitKernel<<<nBlk, nThd>>>(data, init, length); //Set all to init
    return d_data;
}

// omp_kernel.cc
void DoSomething() {
    const int length = 1024;
    int *devMemFromCuda = AllocAndInitialize(5, length);

    #pragma omp target is_device_ptr(devMemFromCuda)
    for (int i = 0; i < length; ++i) {
        devMemFromCuda[i] = devMemFromCuda[i] * 2;
    }
}
```

Example: Sharing device storage between OpenMP and CUDA with is_device_ptr
OpenMP and CUDA: use_device_ptr

- Use **use_device_ptr** to pass mapped device memory from OpenMP to another programming model (e.g. CUDA)

```
// cuda_launch_kernel.cu
void LaunchCUDAIncrement(int *data, int length) {
    ...
    IncrementKernel<<<nBlk, nThd>>>(data, length);
}

// omp_map_and_call_cuda.cc
void DoSomething() {
    const int len = 1024;
    int data[len] = {0,};

    #pragma omp target data map(data[:len]) use_device_ptr(data[:len])
    {
        LaunchCudaIncrement(&data, len);
        #pragma omp map(data[:len])
        for (int i = 0; i < len; ++i) {
            data[i] = data[i] * 2;
        }
    }
}
```

Example: Sharing device storage between OpenMP and CUDA with use_device_ptr