LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-726979

Issues Identified During September 2016 IBM OpenMP 4.5 Hackathon

D. F. Richards

March 15, 2017

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Issues Identified During September 2016 IBM OpenMP 4.5 Hackathon

Author: David Richards (richards12@llnl.gov)
Date: 15-Mar-2017

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

1 Introduction

In September, 2016 IBM hosted an OpenMP 4.5 Hackathon at the TJ Watson Research Center. Teams from LLNL, ORNL, SNL, LANL, and LBNL attended the event. As with the 2015 hackathon, IBM produced an extremely useful and successful event with unmatched support from compiler team, applications staff, and facilities. Approximately 24 IBM staff supported 4-day hackathon and spent significant time 4-6 weeks out to prepare environment and become familiar with apps. This hackathon was also the first event to feature LLVM & XL C/C++ *and* Fortran compilers.

The time and effort invested in the hackathon pays substantial dividends. In particular:

- The compiler team gets experience with real applications and fussy developers
- Developers get personalized support for their applications
- Standards committee gets feedback to drive needed improvements

This report records many of the issues encountered by the LLNL teams during the hackathon. A draft version of this document was prepared for the OpenMP face-to-face meeting held in Japan in October, 2016 and was used to help drive improvements to the standard. According to IBM approximately 20 compiler bugs found during the hackathon are now fixed (as of 12/7/16).

2 Issues Identified (Cardioid)

2.1 Incorrect code generation

We were forced to use the clang compiler for the entire hackathon. The code to transfer our simple data structure to the target (listed below) triggered a bug in the xlC compiler that caused it to produce incorrect code. When transferring a C data structure with a pointer to a dynamic array, we chose to first copy over the whole data structure to the GPU with a simple bit-wise copy, then replace the GPU structure's pointer (which now pointed to host memory) with a new device-allocated dynamic array. This confused the xlC compiler—it saw that we were mapping into a piece of memory that had already been and tried to optimize out the second dynamic allocation. We notified the compiler developers and were told that the problem could not be fixed during the hackathon.

```
struct PADE {
  int l;
  int m;
  double* coef; //dynamic array of size l+m
};

const int fit_length=31;
PADE fit_[fit_length];

vector<double*> state_;

TT06Dev_Reaction::TT06Dev_Reaction(…){
    ...

    #pragma omp target enter data map(to: fit_[:fit_length])
    for (int i = 0; i < fit_length; i++) {
        #pragma omp target enter data map(to: fit_[i].coef[:fit_[i].m +
fit_[i].l])
    }

    const int nState = state_.size();
    double ** stateAlias = &state_[0];
    #pragma omp target enter data map(to: stateAlias[0:nState])
    for (int istate = 0; istate < nState; istate++) {
        #pragma omp target enter data map(to:
stateAlias[istate][0:nCells_])
    }
}
```

2.2 No way to place data in constant memory on the device.

Cardioid needs to evaluate rational polynomials with constant coefficients. Ideally, these polynomial coefficients should be read from input at run time. For best performance, these coefficients need to be placed in constant memory within the GPU (a 4x difference in speed). However, the OpenMP standard does not provide a

capability to specify that certain memory needs to be in GPU constant memory. Here is the code example we sent to IBM that exposes the problem.

```
#pragma omp declare target
    const int Tau_m = 18;
    const double Tau_a[] = {/*An array of 18 numbers*/};
#pragma omp end declare target

int main(){
   ...

   #pragma omp target teams distribute parallel for thread_limit(128)
   for (int ii=0; ii<nCells; ii++){
      sum1 = 0;
      for (int j = Tau_m-1; j >= 0; j--)
         sum1 = Tau_a[j] + x*sum1;

      double tauR = sum1;
      m_gate[ii] += (mhu - m_gate[ii])*(1-exp(-tauR));
   }
}
```

The CORAL compiler had a non-standard __attribute(3) type that we were told could designate a variable declared in a **target** region to be put into constant memory, but this approach is not portable. In our case, trying to use the extra keyword triggered compiler bugs that we never resolved.

```
#pragma omp declare target
    const int Tau_m = 18;
    double __attribute((address_space(3)))Tau_a[] = {/*An array of 18
numbers*/};
```

To get the best performance, we had to embed the polynomial coefficients directly within the **target parallel for**. The resulting code is extremely hard to read and maintain. Under OpenMP 4.5 the code would have to be changed and recompiled every time we wanted to change the interpolating polynomials.

2.3 Features related to task pragma and pipelined kernel launches are not yet implemented

In Cardioid, the 12 gate equations are all data parallel and can be run concurrently. Under CUDA we could pipeline these kernel launches asynchronously, but parallel tasking is currently not implemented in the CORAL compilers.

2.4 Lack of valid introspection when using inlined templates in a device section.

We attempted to use templates to improve the maintainability of the code used in the gate equations:

```
struct padeHack_mMhu {
    enum { l = 10, m = 5};
    static constexpr double a[] = {{/* an array of 15 numbers */};
};
constexpr double padeHack_mMhu::a[15];

struct padeHack_mTau {
    enum {l = 1, m = 18};
    static constexpr double a[] = {{/* an array of 18 numbers */};
};
constexpr double padeHack_mTau::a[19];

template<class hacker>
inline double padeFuncTry_(const double x) {
    double sum1 = 0;
    for (int j = hacker::l + hacker::m - 1; j >= 0; j--)
        sum1 = hacker::a[j] + x * sum1;
    if (hacker::l < 2) return sum1;
    double sum2 = 0;
    int k = hacker::m + hacker::l - 1;
    for (int j = k; j >= hacker::m; j--)
        sum2 = hacker::a[j] + x * sum2;
    return sum1 / sum2;
}

...
  //use the template:
  double mMhu = padeFuncTry<padeHack_mMhu>(Vm);
...
```

The templatized version above was 4x slower. Inspection of the assembly revealed that unnecessary synchronization calls were generated before the inlined code for the template. IBM theorized that due to the ordering of internal passes in the compiler's optimizations, the compiler was being more conservative than was warranted for this case. IBM promised to fix the problem.

2.5 Compiler bugs

We tried mapping our static polynomial coefficient arrays into constant memory by declaring const/constexpr arrays with the non-standard __attribute(3) type at the file scope and within a **declare target** section. This triggered a segmentation fault in the compiler.

2.6 Cannot find/link CUDA float functions in the math library

In one attempt to improve performance, we converted all the variables in the kernel from double to float. However, compiler failed to find/link CUDA float function, expf(), in the math library.

2.7 Changing the variables from double to float doesn't improve performance with current compiler and machines.

We tried to change our application to use floats instead of double precision to test if we could further speedup the kernels. The theory was that our memory loads would improve and we might be able to take advantage of both the float32 units and the float64 units on the GPU at the same time. We replaced doubles with floats everywhere we could—our rational polynomial approximations required double precision to produce correct results. The resulting code was slower than the double version of the code and we did not have time to dig into the assembly to figure out why, so we abandoned the attempt.

2.8 Compiler flag "-ffp-contrast=fast" is needed to generate fused multiply adds with the CORAL compiler.

We verified this by checking the assembly code both before and after using this option.

2.9 Compiler flag "-fopenmp-nonaliased-maps" improves the performance of our kernels by 16%.

This option auto-adds the "restrict" keyword to all target pointers, which was fine for our test problem. Equivalent speedups could be had by using the restrict keyword with the codebase.

3 Issues Identified (Quicksilver)

3.1 Hardware mapping and thread affinity are difficult to manage

Is there anything we can do to make this easier? Can we have better defaults or is there just no good default choice? The current system of cryptic scripts, commands, and environmental variables is very hard to use. Can we do anything to help users verify they're getting the mappings and affinities they want?

3.2 CPU only performance suffers when code is built with unified memory

6x slowdown on code running only on CPU with unified memory. All allocations are in initialization so this isn't a problem with overhead in cudaMallocManaged()

3.3 Can't refer to variables passed by reference in a target region

When I tried to use a variable that was passed by reference in a target region the compiler crashed. This is apparently a known problem.
Workaround: Create a temporary copy and refer to the copy instead. (ToDo: was it a copy or an alias?)

3.4 Can't refer to C++ this pointer in target region

Compiler segfaults when attempting to refer to the this pointer in a target region. This includes implicit references such as accessing a member variable.
Workaround: Use a copy

3.5 Can't map *this or this[0:1]

All of our attempts to write a map() member function that would map the object produced incorrect results. Is mapping this disallowed by the standard or is this a compiler issue? Or were we just using the wrong syntax? Other developers report there is a workaround involving mapping a copy.

3.6 Difficulty using declare target selectively for class member functions

This is the issue that declare target is only effective at file or namespace scope. Tom Scogland and I have discussed this at length.
Issues include:

- IBM workaround doesn't work for overloaded functions
- IBM warnings are inconsistent. Warns when declaration doesn't have declare target but definition does. Silently ignores missing declare target on definition.
- Why is the placement of declare target limited in the first place. CUDA has no problem decorating a member function with __device

3.7 Can't put critical section in target code

This crashed either at compile or run time. I can't recall which. It might be a bad idea for performance to put a critical section in GPU code, but it should work.

3.8 Triple nested parallel for give wrong results on GPU

Function f1() contains a parallel loop that calls f2() that contains a parallel loop that calls f3() that does work in a parallel loop. This works fine on CPU. Answer is incorrect on GPU. Performance is a separate issue from correctness. Sample code available.

3.9 Murkiness in the standard related to our data mapping strategy

Our strategy for mapping variables relies on the runtime automatically translating certain pointers. (I'm not describing this well. See the discussion at the end of section 4.4). From discussions with Tom it apparently isn't clear (at least to him) what the standard actually requires. Since we rely on this behavior we want a clear standard that we can hold compilers/runtimes to correctly implement.

4 Issues Identified (Hypre)

4.1 Static library support in Clang

the problem with the MPI executables that need mpirun and the lack of static library support in Clang. Want executables built with mpi compiler wrappers to be directly executable without mpirun. Device code not included in static libraries.

4.2 Offload Issues

- We will need some mechanism to look at what is happening within the offload code.
- Ideally this would be a detailed source line level report on what the compiler did to generate the offload code
- At the minimum it has to be a description of hotspots in the generated code and their source locations

4.3 Non-aliased maps compiler option

- We will need fine grain control of the non-aliased maps compiler option in Clang probably as a clause to omp target(standards/compiler)

4.4 If clause degrades performance

Using the if clause seemed to degrade performance on the host code due to the use of teams distribute parallel for(compiler)

5 Issues Identified (Miranda)

5.1 Seg-faults in xlflang compiler

Seg-faulting with the xlflang compiler, that according to one of the IBM guys on my team — as far as I understand it — was due to not mapping scalars firstprivate to the device by default, which is the usual behavior for most other compilers, like xlf95. This may be just be a case of growing pains for a new compiler that's wasn't quite ready for prime time — or unfounded assumptions on my part.

5.2 Non-performant default settings

The default settings for number of teams and threads using the xlf95 compiler were dreadful in terms of performance, and needed to be set and tuned by hand for the specific hardware. This may be par for the course, but this lazy (whiny) novice would like something more portable where the compiler chooses more sensible default values given the target device and/or there is dynamic optimization of these parameters at runtime.

5.3 Workshare construct unavailable (?)

There was, unfortunately, no discussion about the `workshare` construct for Fortran, and indeed I was under the (perhaps mistaken) impression it wasn't

supported in the xl OpenMP compilers we were using. Because `workshare` is used with whole array expressions and intrinsics, forall loops, etc., which are a mainstay of concise modern Fortran programming, it is a really useful feature of OpenMP that I hope is fully supported in future versions.