OvO

OpenMP Versus Offload

Thomas Applencourt - apl@anl.gov

Argonne Leadership Computing Facility
Argonne National Laboratory
9700 S. Cass Ave
Argonne, IL 60349

Table of contents

Introduction

What is OvO?

- OpenMP versus Offload
- Not a conformance test suite!
- An OpenMP Offload Mathematical and Hierarchical Parallelism Test Suite

What is OvO? More technically

- 2700+ OpenMP 5.0 tests (C++ and FOR$_{\text{mula}}$TRAN$_{\text{slation}}$)[1]
- Scripts to compile, run, and check correctness
  - make
  - Bourne-Again SHell >3.2
  - Optionally $_{\text{monthy}}$Python3[2]
- *`https://github.com/TApplencourt/OvO`*

[1] More precisely C++11 and F90 standards. Some math function are defined in C++17 and C++20 standards

[2] Used to generated summaries, or re-generate tests source-code

Why OvO?

1. OpenMP specification is extensive
2. Compilers only support a subset of it

For application developers:

- Check if a required feature is supported by a majority of compilers (else don't use it)

For compiler developers:

- Check if a required feature is supported by our compiler (else implement it)

How to Use OvO?

Help:

```
./ovo.sh
Usage:
  ovo.sh gen
  ovo.sh run [<test_folder>...] [--no_long] [--no_loop]
  ovo.sh display [--detailed | --failed | --passed ] [--no_long] [--no_loop] [<result_folder>...]
  ovo.sh report  [--no_long] [--no_loop]  [<result_folder>]
  ovo.sh clean
```

How to use OvO? “Live” Demo

```
$ git clone --quiet https://github.com/TApplencourt/OvO

$ CXXFLAGS='-std=c++2a' FC='gfortran' ./ovo.sh run
Running test_src/cpp/complex_cpp11 | Saving in ${long_path}
g++ -std=c++2a log_complex_float_complex_float.cpp -o
↪  log_complex_float_complex_float.exe
[...]
timeout -k 5s 10s ./log_complex_float_complex_float.exe
[...]

$ ./ovo.sh report
>> test_result/2020-05-12_21-02_iris15.ftm.alcf.anl.gov
> cpp math and complex: 259 / 259 ( 100% ) pass
    [failures: 0 compilation, 0 offload, 0 incorrect results]
> cpp hierarchical parallelism: 642 / 642 ( 100% ) pass
    [failures: 0 compilation, 0 offload, 0 incorrect results]
> fortran math: 79 / 79 ( 100% ) pass
    [failures: 0 compilation, 0 offload, 0 incorrect results]
> fortran hierarchical parallelism: 642 / 642 ( 100% ) pass
    [failures: 0 compilation, 0 offload, 0 incorrect results]
> Summary: 1622 / 1622 ( 100% ) pass
    [failures: 0 compilation, 0 offload, 0 incorrect results]
```

Mathematical Functions

How We Test?

- Floating-Point Arithmetic is hard. I'm Not an IEEE 754 expert
- Take a best-effort approach
- Compare GPU value to assumed gold CPU[3]
- Inputs are "goods" inputs:
    - No nan, no infinities, no subnormal, ...
    - Should not trigger any invalid operation, overflow, ...

[3] OpenMP specification doesn't define a required precision. OpenCL and CUDA® does. We choose to use a tolerance of 4 ulp, who correspond to Low Accuracy (LA) mode of Intel Math Kernel Library Vector Mathematics

What functions are we testing?

C++

`cmath.h` (391 tests for C++11, C++17, C++20), *`complex.h`* (51 tests) and one GNU extension (*`sincos`*)

FORTRAN

FORTRAN 77 Language Reference, section "Arithmetic and Mathematical Functions"[4] (67 real and 17 complex tests)

Mathematical Result (in no particular order)[6]

	Hardware	C++	FORTRAN
Cray-llvm 9.1.3	V100	58%	100%
Cray-Classic 9.1.3	V100	49%	100%
clang-AOMP 11.0.1	Radeon VII	70%	44%
gcc 10.0.0	P100	5%[5]	59%
clang 11.0.0	V100	70%	
xlC 16.1.1	V100	76%	100%

[5] I guess / hope it's a problem with our installation...
[6] `--no_loop` used

Hierarchical Parallelism

Goals

- One loop-nest
- All the combinations of OpenMP constructs:
  - team, distribute, parallel, for, loop, simd
- All the permutation of Type:
  - Real, Complex[7]
  - Single, Double precision
- Two big categories of tests:
  1. memory copy
  2. fold (reduction, atomic)

```
$find ./test_src/*/hierarchical_parallelism -type f | wc -l
2664
```

[7] OvO assume implicit mapping of `std::complex`. Not part of the C++ OpenMP standard

Quaternion of Extreme: One Line Complex Reduction

```
#pragma omp declare reduction(+: complex<float>: omp_out += omp_in)
std::complex<float> counter{};
#pragma omp target teams distribute parallel for simd
↪  reduction(+:counter) map(tofrom:counter)
for (int i = 0 ; i < N*M*L ; i++)
    counter += std::complex<float> { 1.0f };
```

OpenMP features

- Custom reduction[8]
- In the combined statement, the *map* close and *reduction* close share a variable (OpenMP 5.0)

OpenMP “features”

Non-trivial type `complex<float>` is mapped

[8] Doesn’t required in FORTRAN. Just saying that maybe one language is superior to another...

Quaternion of Extreme: exploded atomic

```
REAL counter = 0.
!$OMP TARGET map(tofrom:counter)
!$OMP TEAMS
!$OMP LOOP
DO i = 1 , L
    !$OMP PARALLEL
    !$OMP LOOP
    DO j = 1 , N*M
        !$OMP ATOMIC UPDATE
        counter = counter +  1.
    END DO
    !$OMP END LOOP
    !$OMP END PARALLEL
END DO
!$OMP END LOOP
!$OMP END TEAMS
!$OMP END TARGET
```

OpenMP Restriction

- No `!$OMP ATOMIC` in `!$OMP SIMD`
- No `!$OMP ATOMIC` with complex

Quaternion of Extreme: CUDA®-like reduction+atomic

```
double counter{};
#pragma omp target map(tofrom:counter)
#pragma omp teams
{
    const int num_teams = omp_get_num_teams();
    double partial_counter{};
    #pragma omp parallel reduction(+: partial_counter)
    {
        const int num_threads = omp_get_num_threads();
        partial_counter += double { 1.0f/(num_teams*num_threads) } ;
    }
    #pragma omp atomic update
    counter += partial_counter;
}
```

Trivia: What is the final value of `counter`?

Argonne NATIONAL LABORATORY

Quaternion of Extreme: host hierarchical parallelism

```
REAL counter = 0.
!$OMP PARALLEL DO REDUCTION(+: counter)
DO i = 1 , L
  !$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO MAP(TOFROM: counter)
  DO j = 1 , N*M
    !$OMP ATOMIC UPDATE
    counter = counter +  1.
  END DO
  !$OMP END TARGET TEAMS DISTRIBUTE PARALLEL DO
END DO
!$OMP END PARALLEL DO
```

Hierarchical Parallelism Result (in no particular order)[9]

	Hardware	C++	FORTRAN
Cray-llvm 9.1.3	V100	70%	99%
Cray-Classic 9.1.3	V100	77%	99%
clang-AOMP 11.0.1	Radeon VII	50%	47%
gcc 10.0.0	P100	62%	81%
clang 11.0.0	V100	85%	
xlC 16.1.1	V100	73%	100%

[9] `--no_loop` used

Argonne NATIONAL LABORATORY

Conclusion

Conclusion

- OvO (*https://github.com/TApplencourt/OvO*)
- Nobody is perfect
- but FORTRAN is more perfect than others [10]

[10] Cray and xlC have near a 100% pass rate. And FORTRAN have native complex support