su3_bench, a Micro-benchmark for Exploring Exascale Era Programming Models, Compilers and Runtimes

Douglas Doerfler, Lawrence Berkeley National Laboratory
Christopher Daley, Lawrence Berkeley National Laboratory
Thomas Applencourt, Argonne National Laboratory

P3HPC Forum
September 1st, 2020

The su3_bench benchmark

- su3_bench was developed to provide a means to explore different programming methodologies using a simple, but nontrivial, mathematical kernel
- Derived from the MILC Lattice QCD (LQCD) code
  - Matrix-matrix and matrix-vector SU(3) (special unitary group of degree 3) operations are a fundamental building block of LQCD applications
  - Most LQCD applications use domain specific implementations (libraries) written in machine specific languages and/or intrinsics …
  - Hence performance portable methodologies are of interest
- Kernel calculates an SU(3) matrix-matrix multiply of complex numbers
  - Benchmark operates over a lattice of dimension = L^4
- https://gitlab.com/NERSC/nersc-proxies/su3_bench
  - Released as open-source software under LBNL's modified BSD license

su3_bench data structures

- SU(3) matrix definition (72 bytes single, 144 bytes double)

```
typedef struct { std::complex<float> e[3][3]; } fsu3_matrix;
typedef struct { std::complex<double> e[3][3]; } dsu3_matrix;
#if (PRECISION==1)
  #define su3_matrix    fsu3_matrix
#else
  #define su3_matrix    dsu3_matrix
#endif
```

- Site definition
  - Based on MILC's lattice.h, but reduced to bare minimum of fields

```
typedef struct {
  su3_matrix link[4];  // the fundamental gauge field
  int x,y,z,t;         // coordinates of this site
  int index;           // my index in the array
  char parity;         // is it even or odd?
#if (PRECISION==1)
  int pad[2];          // pad out to 64 byte alignment
#else
  int pad[10];
#endif
} site __attribute__ ((aligned));
```

C = A * B

su3_bench performs a 3x3 complex matrix-matrix multiply for each gauge field in the 4 lattice dimensions

The kernel: C = A * B

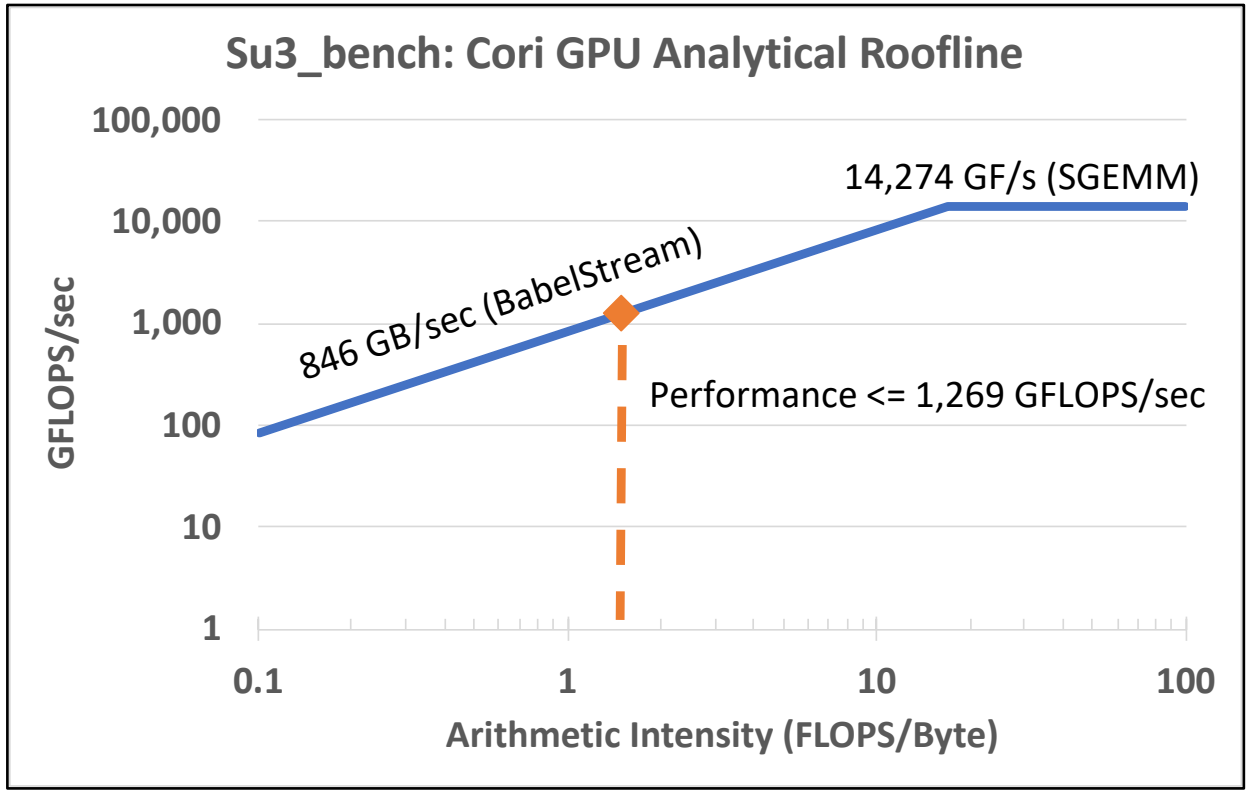
```
for (i=0;i<total_sites;++i)    // L^4 lattice sites
  for (j=0;j<4;++j)            // 4 links, SU(3) matrices, per site
    for(k=0;k<3;k++)           // 3x3 matrix elements per link
      for(l=0;l<3;l++) {
        cc = {0.0,0.0};
        for(m=0;m<3;m++)       // 3x1 dot product per matrix element
          cc += A[i].link[j].e[k][m] * B[j].e[m][l];
        C[i].link[j].e[k][l] = cc;
      }
```

Nominal GPU parallelization strategy:

- For each site, create 4*3*3=36 threads
- Each thread does a single 3x1 vector dot product
- Reduces the number of Sites/group and alleviates cache pressure

Analytical roofline model

- A & C are lattices of size L^4 sites
  - su3_matrix[4] → 288 bytes/site
  - A is read once per iteration
  - C is written once per iteration
- B is a single su3_matrix[4] array
  - Relatively small, should stay in cache
- Total Bytes = 576 Bytes (single-precision)
- Total FLOPS = 864 FLOPS/site
- Arithmetic Intensity (FLOPs/Byte)
  - AI = 864 / 576 = 1.5 single-precision
  - AI = 0.75 double-precision



Test beds used for this study

	NERSC: Cori GPU	OLCF: Lyra	ALCF: Iris
GPU architecture	Nvidia V100	AMD MI-60	Intel Gen9 NEO
# units/device	80 SM	64 CU	72 EU
FP32 cores/simd lanes	5120 = SMs*64	4096 = CUs*64	576 = EUs*2*4
FP64 cores/simd lanes	2560 = SMs*32	same	144 = EUs*1*2
L2 cache	6144 KB	4096 KB	1536 KB
L1 cache	6400 KB/SM (shared)	16 KB/CU	
TFLOP/s peak (nominal/boost clock)	13.4/15.7 single 6.72/7.83 double	9.83/14.8 single 4.92/7.37 double	1.32 single 0.331 double
TFLOP/s sustained	14.3(1) single 7.05(1) double	11.2(1) single 5.63(1) double	1.21(3) single 0.302(3) double
Gbyte/s	897(2) peak 847(2) sustained (94%)	1024(2) peak 816(2) sustained (80%)	25.6(3)

1. Using mt-dgemm benchmark
2. Using BabelStream benchmark
3. Using Empirical Roofline Toolkit, single-precision is derived from double-precision

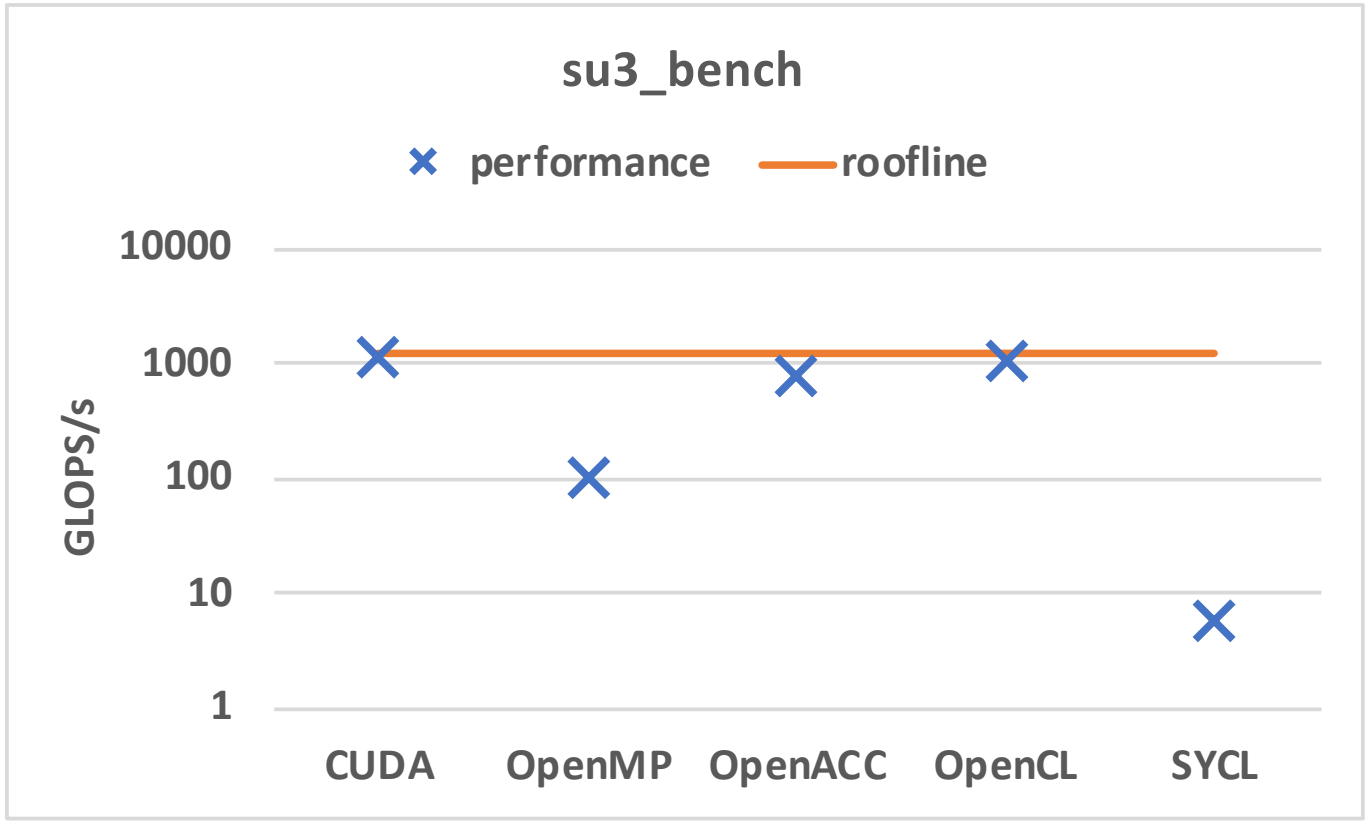
Cori-GPU Programming Environments

CUDA	HIP	OpenCL	OpenMP	OpenACC	SYCL	Intel DPCPP
CUDA 10.2.89	rocm-3.3.0	Version 1.2 • GCC • OpenCL in CUDA driver • POCL: based on llvm 9 w/SPIRV-LLVM translator; CUDA 9.2.148	llvm/10.0.0 • CUDA 10.1.243 PGI/19.20-alpha2 Cray PE	PGI/19.10 • Cori GPU module Cray PE	Codeplay ComputeCpp 1.3.0 • With POCL (see OpenCL) • Experimental PTX target hipSYCL • llvm/9.x • CUDA 10.0.130	*sycl* branch • With Codeplay developed NVPTX backend • CUDA 10.1.243

- Environments in bold where used for this study
- Environments in grey are available, but not explored here
  - I will note that POCL outperformed Nvidia's OpenCL driver by 22% on average

Early Results (Fall 2019)

	CUDA	OpenMP	OpenACC	OpenCL	SYCL
# threads/SM	128	36	N/A	128	128
GFLOPS/sec	1112	104	810	1095	5.8
analytical roofline	1269	1269	1269	1269	1269



- Note: Log scale!
- CUDA and OpenCL perform near roofline
- OpenACC is respectable
- OpenMP & SYCL have serious issues

OpenMP Workaround

Nominal Implementation: one thread/dot product

```
#pragma omp target teams distribute \
            thread_limit(threads_per_team)
for(int i=0; i<total_sites; ++i) {
  #pragma omp parallel for collapse(3)
  for (int j=0; j<4; ++j) {
    for(int k=0;k<3;k++) {
      for(int l=0;l<3;l++){
        Complx cc = {0.0, 0.0};
        for(int m=0;m<3;m++)
          cc += d_a[i].link[j].e[k][m]
                * d_b[j].e[m][l];
        d_c[i].link[j].e[k][l] = cc;
      }
    }
  }
}
```

= 104 GF/s

→ LLVM implementation: end of parallel region forces a flush after each iteration, resulting in excessive memory traffic[1]

Workaround: OpenCL like implementation[2], w/manual collapse

```
size_t num_work_items = total_sites *
threads_per_team;
#pragma omp target teams distribute parallel for
for (int id =0; id < num_work_items; id++) {
  int i = id/36;
  int j = (id%36)/9;
  int k = (id%9)/3;
  int l = id%3;
  Complx cc = {0.0, 0.0};
  for(int m=0;m<3;m++)
    cc += d_a[i].link[j].e[k][m]
          * d_b[j].e[m][l];
  d_c[i].link[j].e[k][l] = cc;
}
```

= 1028 GF/s !!!

1. Thanks to Chris Daley (LBL) for help with implementation and identifying the flush “feature”
2. Thanks to Xinmin Tian (Intel) for workaround and Intel compiler optimizations

SYCL Workaround

Nominal Implementation: array indexing

```
auto d_a = a_buf.get_access<cl::sycl::access::mode::read>(cgh);
auto d_b = b_buf.get_access<cl::sycl::access::mode::read>(cgh);
auto d_c = c_buf.get_access<cl::sycl::access::mode::discard_write>(cgh);

cgh.parallel_for<class k_mat_nn>(cl::sycl::nd_range<1> {total_wi, wgsize},
[=](cl::sycl::nd_item<1> item) {

  size_t myThread = item.get_global_id(0);
  size_t mySite = myThread/36;
  if (mySite < total_sites) {
    int j = (myThread%36)/9;
    int k = (myThread%9)/3;
    int l = myThread%3;
    Complx cc = {0.0, 0.0};
    for (int m=0;m<3;m++) {
      const auto aa = d_a[mySite].link[j].e[k][m];
      const auto bb = d_b[j].e[m][l];
      cc += aa * bb;
    }
    d_c[mySite].link[j].e[k][l] = cc;
  }
}
```

= 5.8 GF/s

SYCL 1.2.1 spec bug[2]:
For dataT operator[] using read only mode:
"Returns the value of the element stored within the SYCL buffer this SYCL accessor is accessing at the index specified by index."

Workaround: Pointer indexing[1]

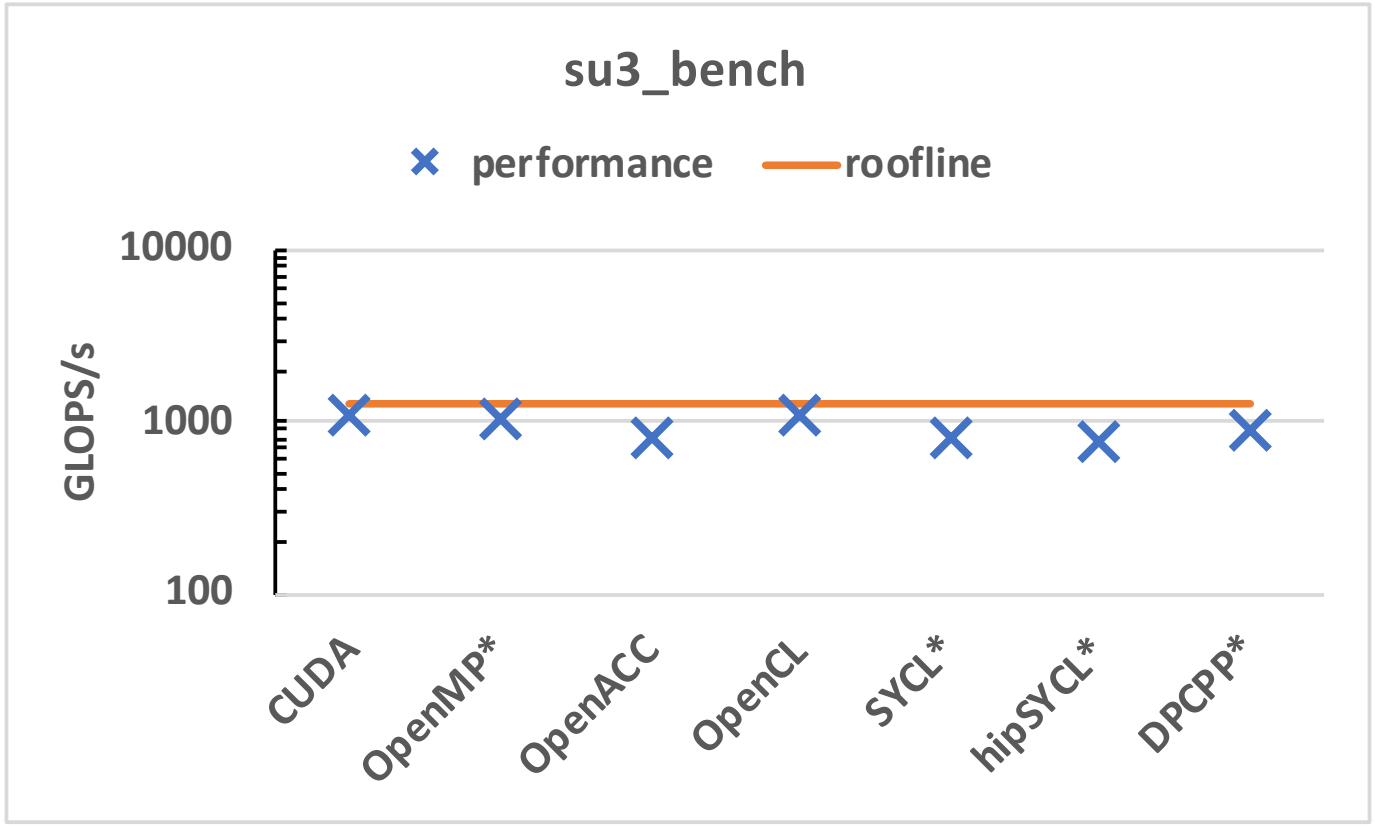
```
for (int m=0;m<3;m++) {
  const auto aa = (d_a.get_pointer() + mySite)->link[j].e[k][m];
  const auto bb = (d_b.get_pointer() + j)->e[m][l];
  cc += aa * bb;
}
d_c[mySite].link[j].e[k][l] = cc;
```

= 816 GF/s !!!

1. Thanks to Thomas Applencourt (ANL) for figuring out pointer reference performs well
2. Thanks to John Pennycook (Intel) for figuring out SYCL spec issue

Results after workarounds

	CUDA	OpenMP*	OpenACC	OpenCL	SYCL*	hipSYCL*	DPCPP*
# threads/SM	128	144	N/A	128	144	144	144
GFLOPS/sec	1111	1028	810	1095	816	767	880
analytical roofline	1269	1269	1269	1269	1269	1269	1269

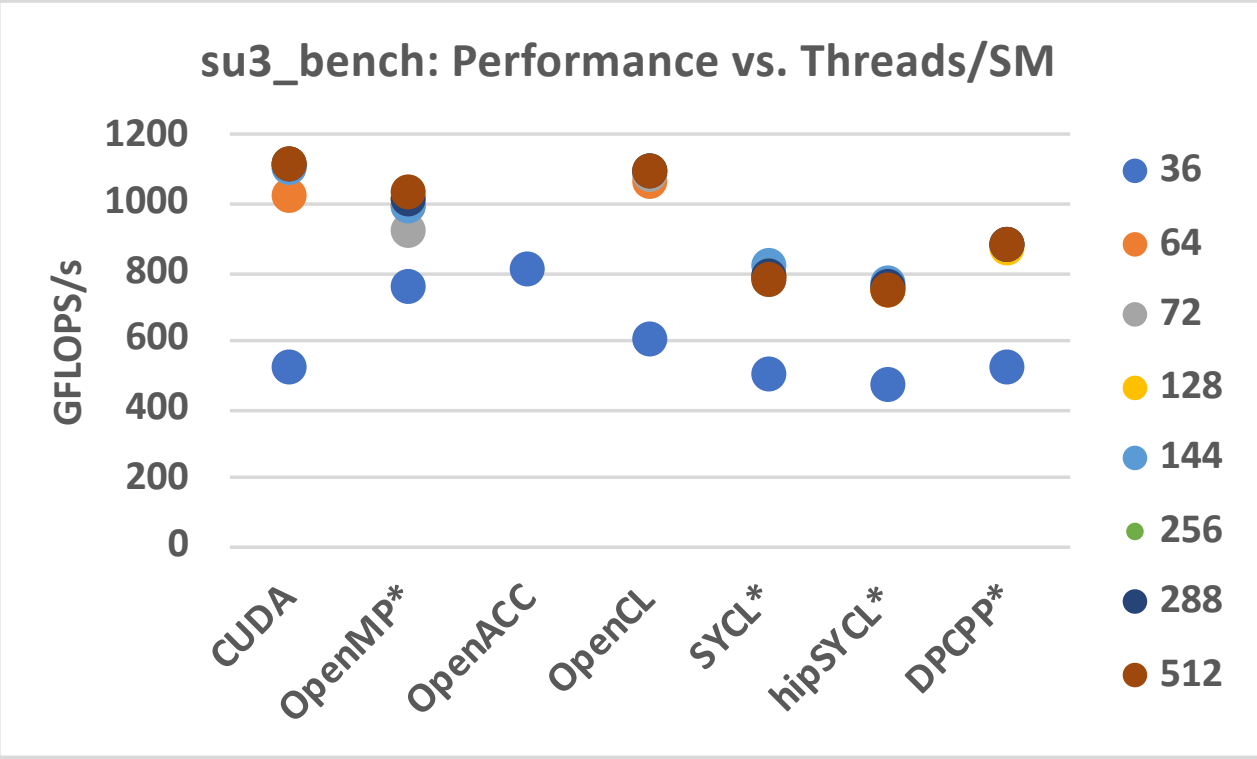


* result with workaround

- CUDA, OpenCL and OpenMP are near the roofline and are essentially BW bound
- OpenACC, and SYCL implementations are still seeing some form of compute bound behavior

Performance vs. Threads/Workgroup

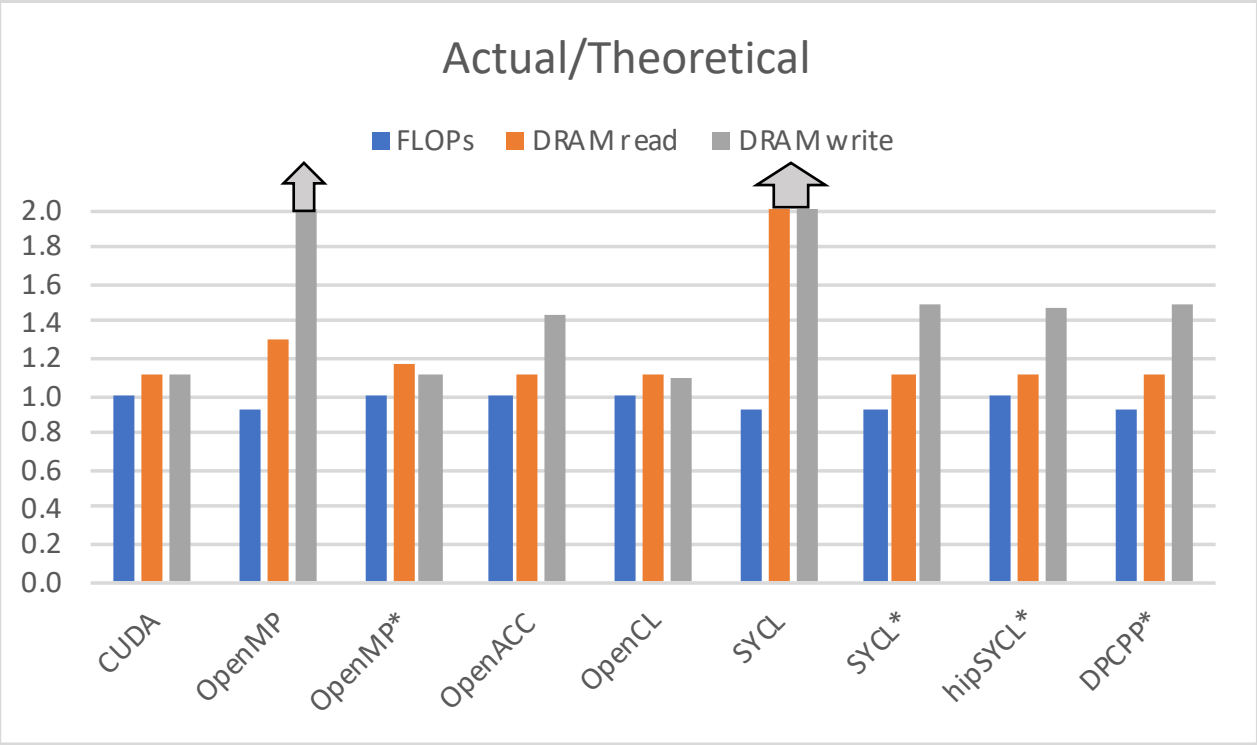
# of threads/SM	CUDA	OpenMP*	OpenACC	OpenCL	SYCL*	hipSYCL*	DPCPP*
36	521.9	757.1	810.0	599.4	498.6	466.7	520.3
64	1025.1	985.3		1056.7	780.6	741.4	878.1
72	1103.2	921.2		1083.7	774.1	758.2	879.3
128	1111.5	1005.5		1095.2	786.6	742.8	870.8
256	1108.0	1020.5		1092.4	806.1	756.8	872.0



- CUDA & OpenCL
  - Require at least 64 threads/block
  - Near roofline performance
- OpenMP, OpenACC, & SYCL
  - Still seem to be have computational inefficiencies

Measured Roofline (using nvprof)

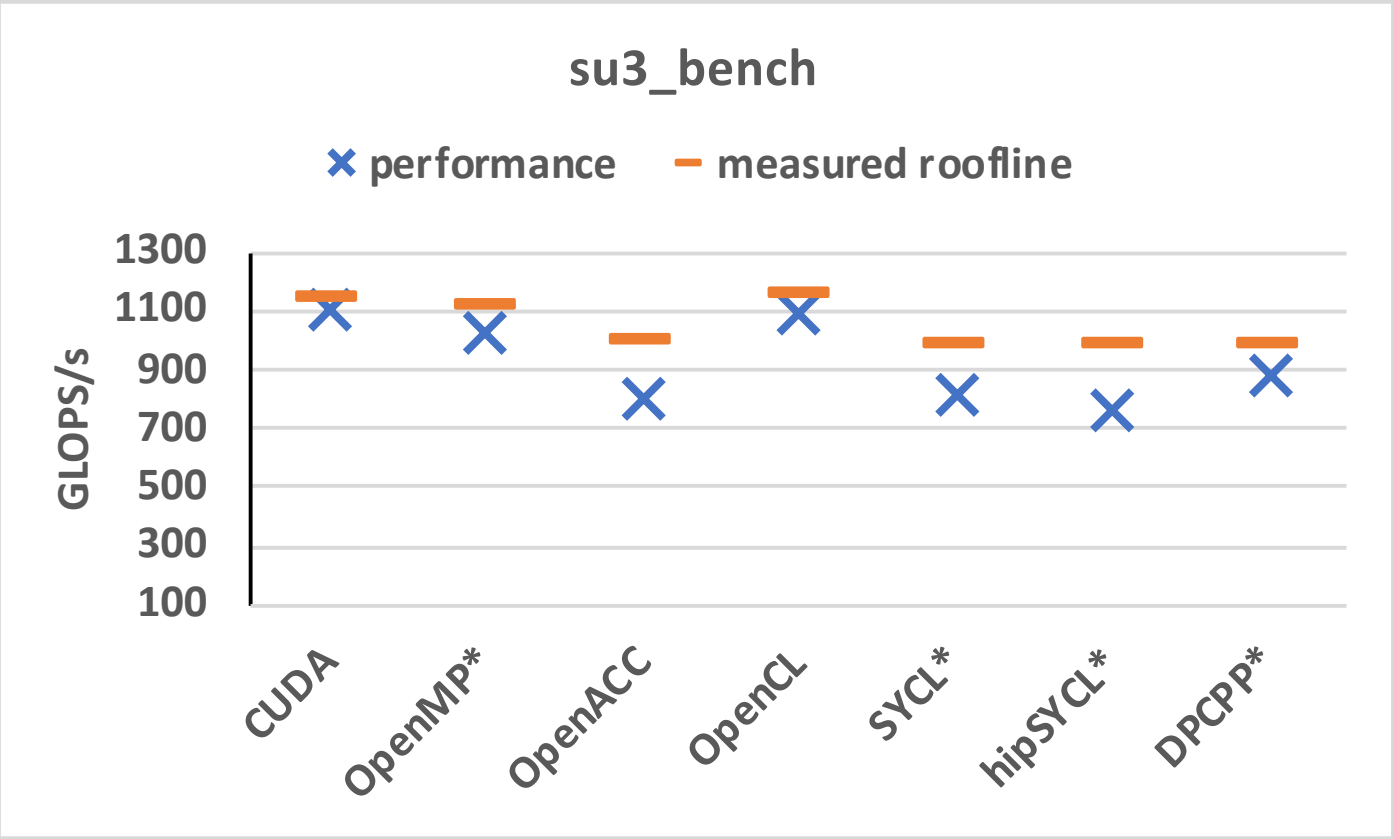
	CUDA	OpenMP	OpenMP*	OpenACC	OpenCL	SYCL	SYCL*	hipSYCL*	DPCPP*
FLOPs	1.00	0.92	1.00	1.00	1.00	0.92	0.92	1.00	0.92
DRAM read	1.11	1.30	1.17	1.12	1.11	41.09	1.11	1.11	1.11
DRAM write	1.11	4.07	1.12	1.44	1.09	243.89	1.48	1.47	1.48
measured Roofline	1144	473	1108	992	1152	9	978	985	978



- Pre-workaround, OpenMP and SYCL implementations were moving a lot of data!
  - SYCL still has high write ratio
- DRAM read ratio of 1.11 is ideal
  - Actual AI is 1.35 including other elements in the site structure,
    1.5 / 1.35 = 1.11
- FLOP counts depend on the compiler
  - C += A * B; for 3x1 vectors
  - 1.00 – All ops are FMA
  - 0.92 – 1st accumulation of 3x1 vector-vector multiply is an assignment

Performance vs. Measured Roofline

	CUDA	OpenMP*	OpenACC	OpenCL	SYCL*	hipSYCL*	DPCPP*
# threads/SM	128	144		128	144	144	144
GFLOPS/sec	1111	1028	810	1095	816	767	880
measured roofline	1144	1108	992	1152	978	985	978

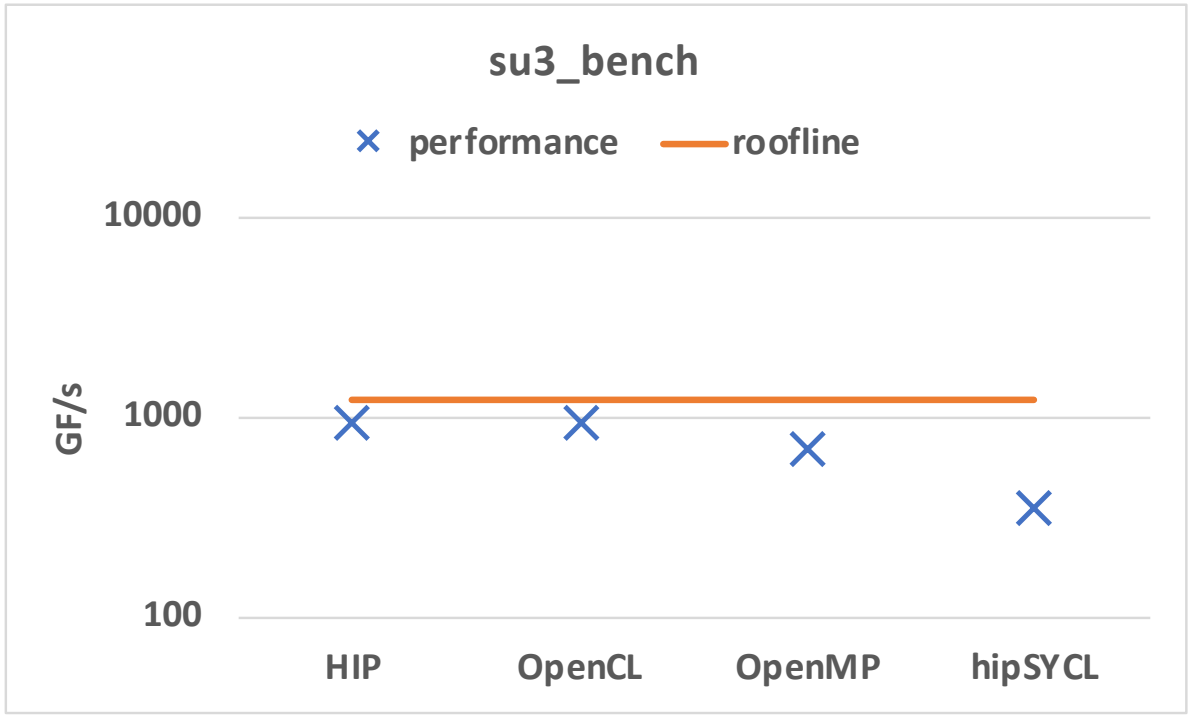


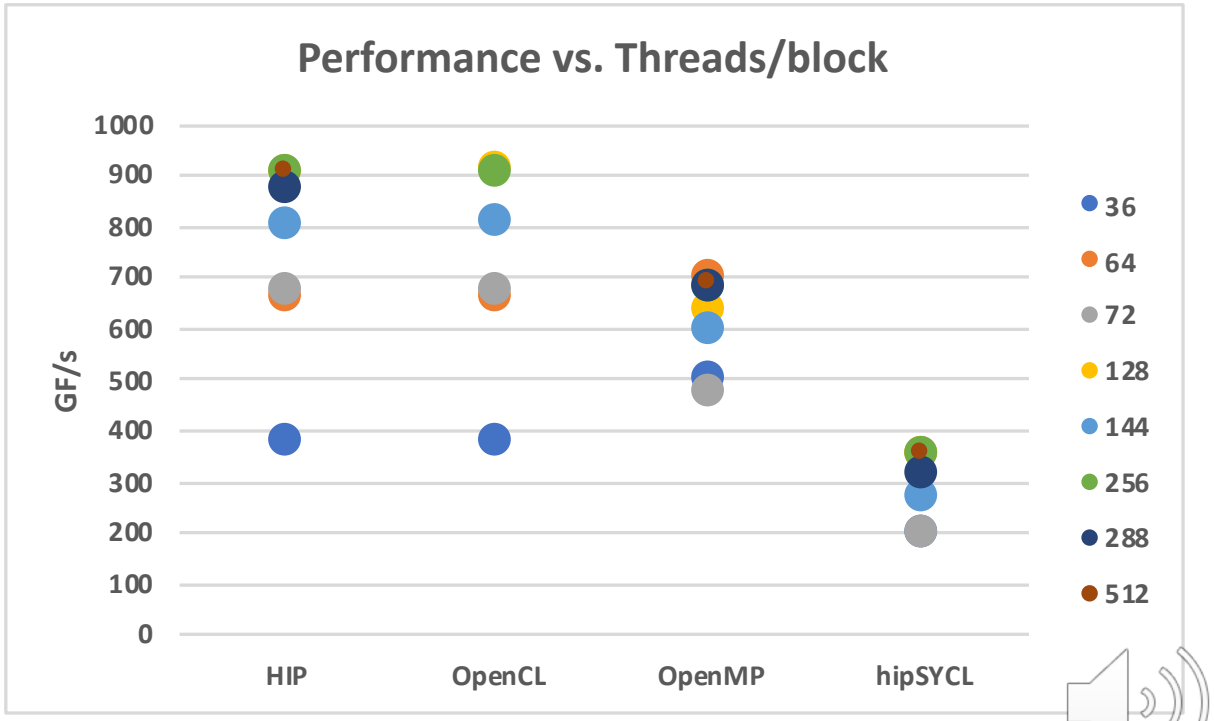
- CUDA, OpenMP and OpenCL are near the roofline and are essentially BW bound
- OpenACC, and SYCL implementations are moving more data and have a lower roofline, in particular writes

Results for AMD Vega 20: OLCF Lyra test bed

	HIP	OpenCL	OpenMP	hipSYCL
# threads/SM	512	128	64	512
GFLOPS/sec	908.1	912.6	703.5	356.2
roofline	1215.8	1215.8	1215.8	1215.8

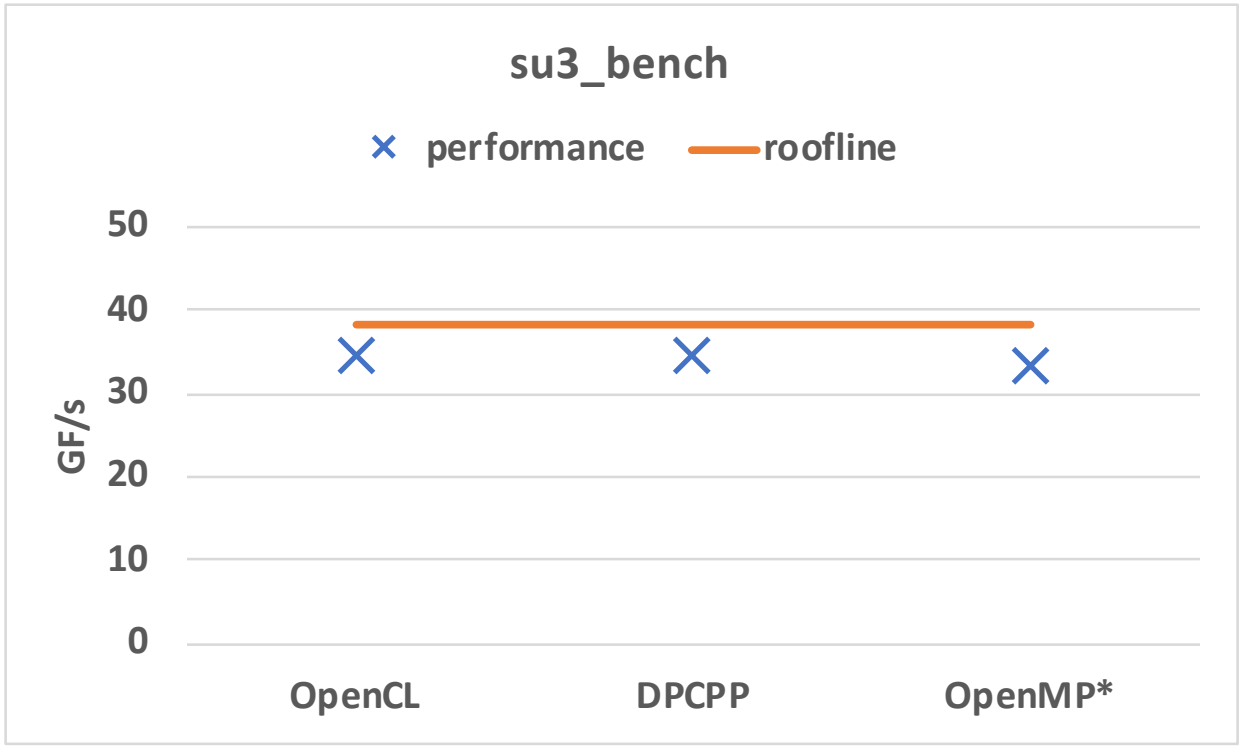
- HIP and OpenCL perform well, but not as good as CUDA on Nvidia's Volta
  - Same 4 stacks of HBM as Volta
- hipSYCL limitation?

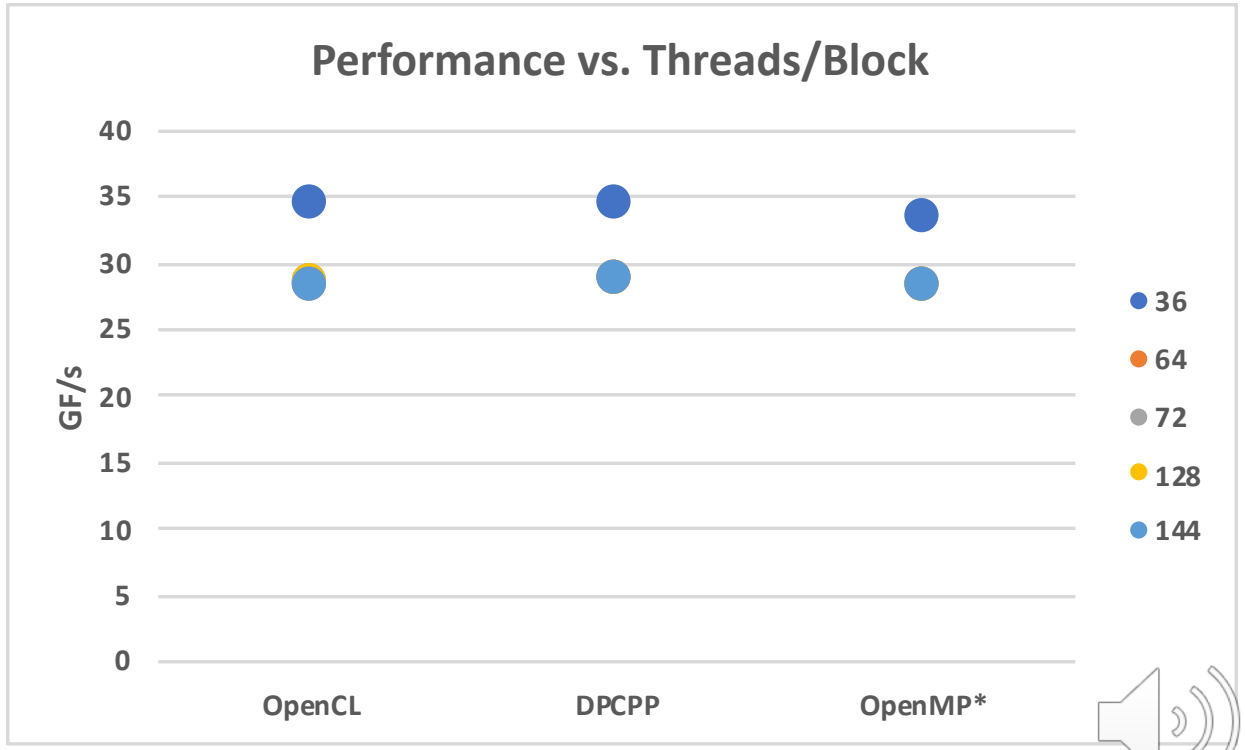




Results for Intel Gen9/NEO: ALCF Iris test bed

	OpenCL	DPCPP	OpenMP
# threads/SM	36	36	36
GFLOPS/sec	34.6	34.5	33.4
roofline	38.4	38.4	38.4





Programming Model vs. Architecture

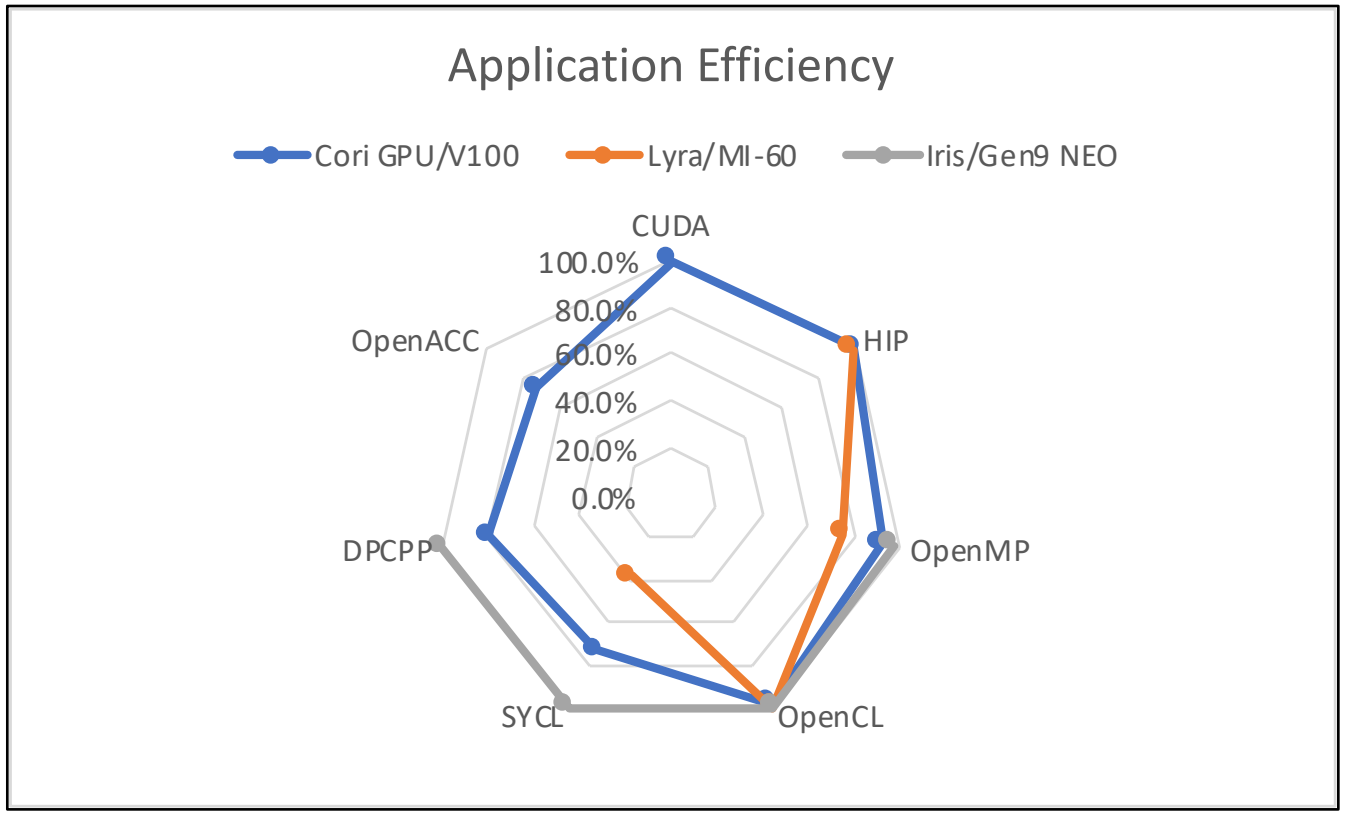
	CUDA	HIP	OpenCL	OpenMP	OpenACC	SYCL	DPCPP
Nvidia	X	X	X	X	X	X(1)	X(2)
AMD		X	X	X		X(3)	
Intel			X	X		X	X

1. ComputeCPP with POCL, which is experimental/unsupported; ComputeCPP also supports a NVPTX backend, but it's deemed experimental and had performance issues with su3_bench
2. This study used DPCPP as a SYCL compiler, SYCL extensions are untested
3. hipSYCL only at this point in time; ComputeCpp doesn't support GCN backend, perhaps POCL works?

Performance Portability[1]

	Programming Model Efficiency
Cori GPU	86.2%
Lyra	66.9%
Iris	99.0%

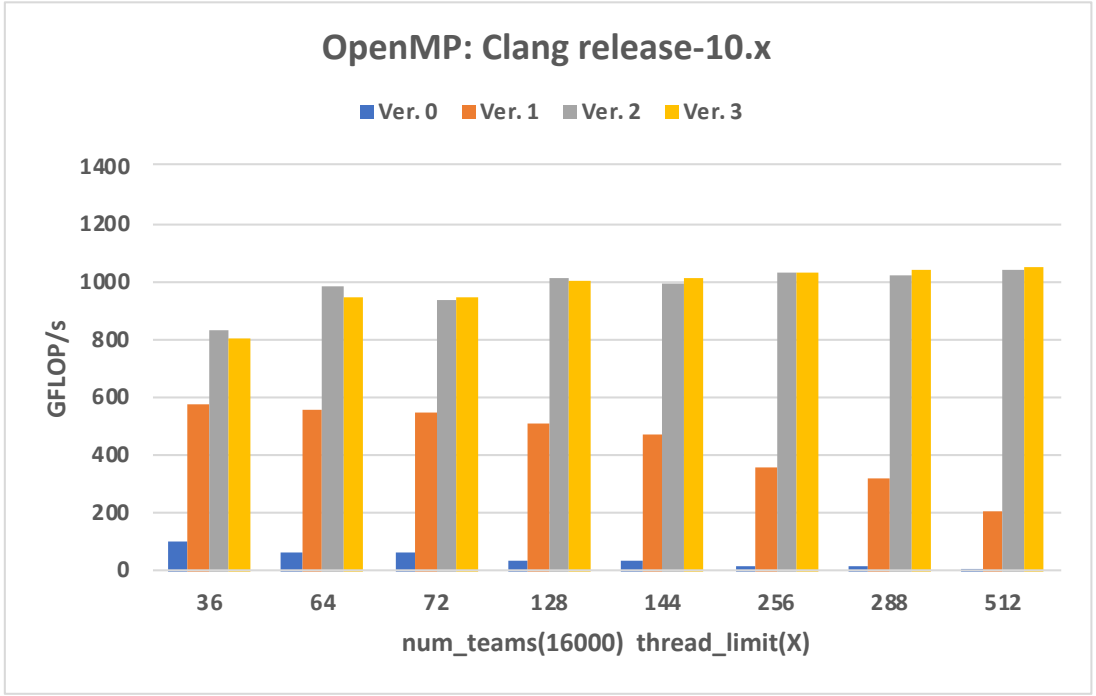
	Cross Platform Efficiency
OpenMP	86.9%
OpenCL	99.4%
SYCL	59.9%



1) S. J. Pennycook, J. D. Sewall, V. W. Lee, "*A Metric for Performance Portability*", Proceedings of the 7th International Workshop in Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, Nov. 2016

Getting good OpenMP performance can be a challenge

- Su3_bench includes four (4) different OpenMP implementations
  - All 4 seem to be reasonable solutions
  - Drastically varied performance
  - Still necessary to tune with num_teams() and thread_limit() directives
- We have explored Clang, Cray CCE, NVIDIA/PGI and Intel compilers and runtimes
  - Using su3_bench to explore OpenMP compilers and runtimes is a presentation in itself!



Clang release-10.x results

Version 0: Nominal version (see OpenMP issue slide)
Version 1: Manually distribute sites across teams
Version 2: Work item version (see OpenMP issue slide)
Version 3: Uses collapse(4) over outer loop

Summary and conclusions

- Su3_bench is an open benchmark developed to explore exascale era languages, compilers and runtimes
  - https://gitlab.com/NERSC/nersc-proxies/su3_bench
- Roofline analysis shows that the benchmark is memory bound, however it is more than just another STREAM benchmark
  - A non-trivial complex matrix-matrix multiply kernel with multiple loop nests
  - Initial analysis discovered serious compiler issues that significantly limited performance
  - Even after workarounds and optimizations, performance varies up to 30% across the different programming environments
- Analysis has been performed across NVIDIA, AMD and Intel GPUs
  - Performance portability is good across architectures
  - All languages can target the NVIDIA GPU, not a surprising conclusion given its longevity in the market
- There has been extensive use of su3_bench in evaluating OpenMP compilers and runtimes, results of which are beyond the time allowed by this venue
  - However, if you're interested we'd be happy to work with you

Future Work

- Need to incorporate more realistic memory access patterns
  - Although the SU(3) multiplications represent LQCD codes, the lattice site access patterns of su3_bench do not
  - Higher level Dslash stencil operation proxy-application is desirable
- Need to incorporate Lattice QCD methods that allow effective use of SIMD for CPU targets?
  - Typically incorporates a data reordering technique to allow adjacent sites to have better spatial locality and hence better utilization of long SIMD lengths