2020 PERFORMANCE, PORTABILITY,
AND PRODUCTIVITY IN HPC FORUM

Argonne NATIONAL LABORATORY

PERFORMANCE PORTABLE IMPLEMENTATIONS FOR A GEOMETRIC MULTIGRID KERNEL

JAEHYUK KWACK
ALCF Perf. Engr. Group
Argonne National Laboratory

September 1, 2020

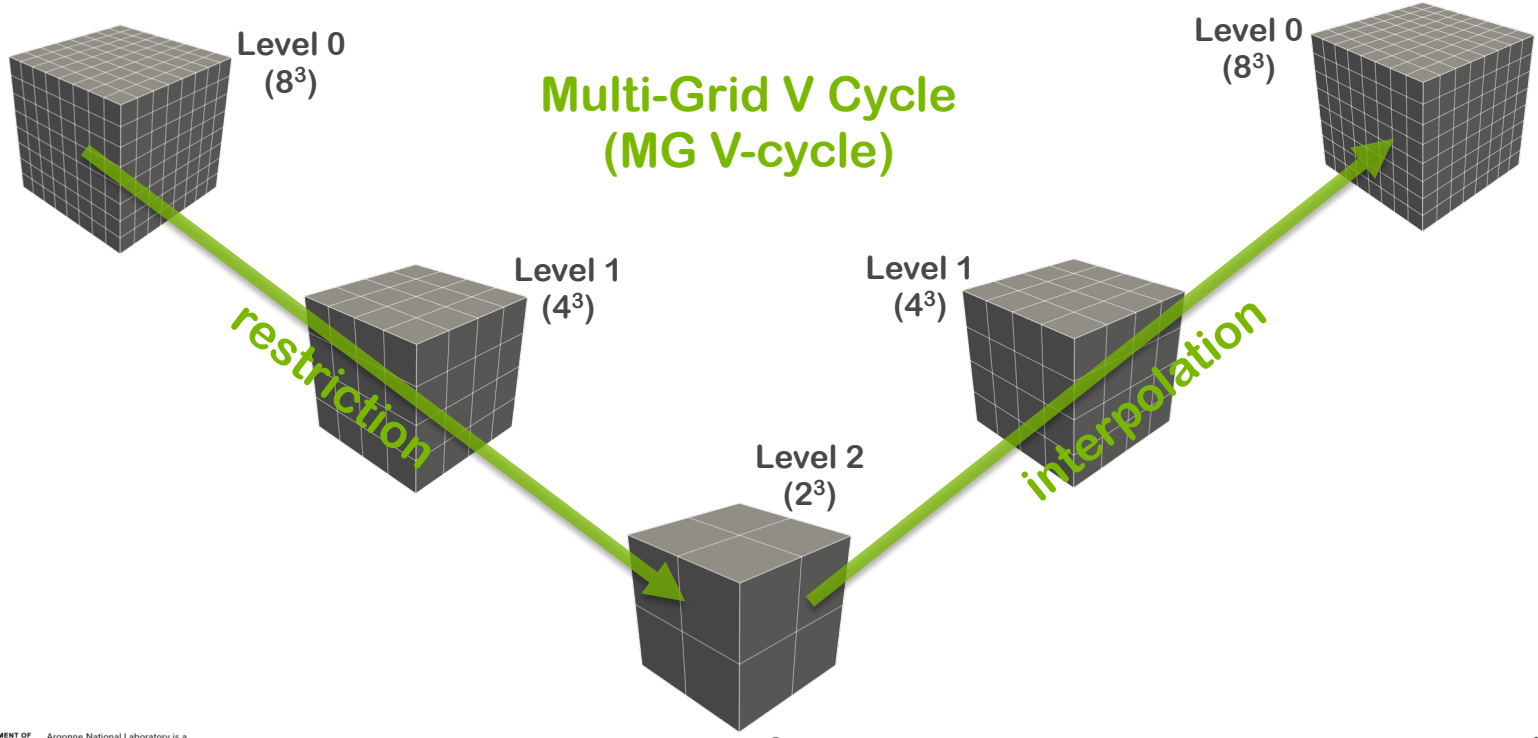
GMG METHODS AND HPGMG [1]

- Geometric Multigrid solves partial differential equations (Lx = f) using a hierarchical approach
  - Coarse grids are constructed from fine grids by agglomeration of fine grid elements/cells
  - Provides O(N) computational complexity where N is number of unknowns.

- HPGMG: a HPC benchmark for full multigrid (FMG) algorithms
  - HPGMG-FE(Finite Element): compute-intensive and cache-intensive
  - HPGMG-FV(Finite Volume): memory bandwidth-intensive
    - Solving an elliptic problem on isotropic Cartesian grids with 4th order accuracy
    - 4× FP ops, 3× MPI messages, 2× MPI message size w/o DRAM data movement compared to 2th order HPGMG-FV
    - Employing the Full Multi-grid (FMG) F-cycle
    - A series of progressively deeper geometric multi-grid V-cycles

MG V-CYCLE

Multi-Grid V Cycle



MG V-CYCLE

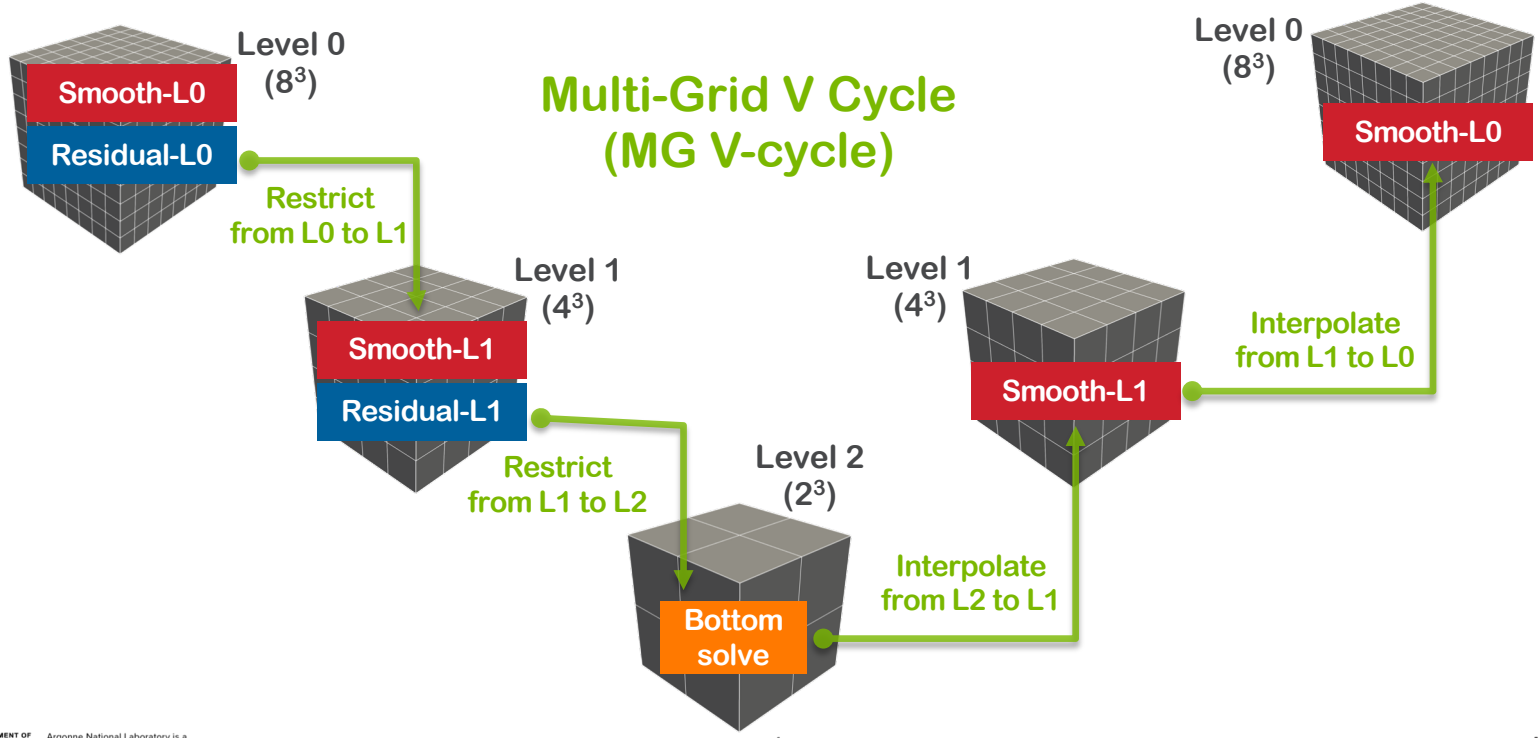
Multi-Grid V Cycle

Restrict from L1 to L2: $f^{L2} = restrict(r^{L1})$

Interpolate from L2 to L1: $u^{L1} += interpolate(u^{L2})$

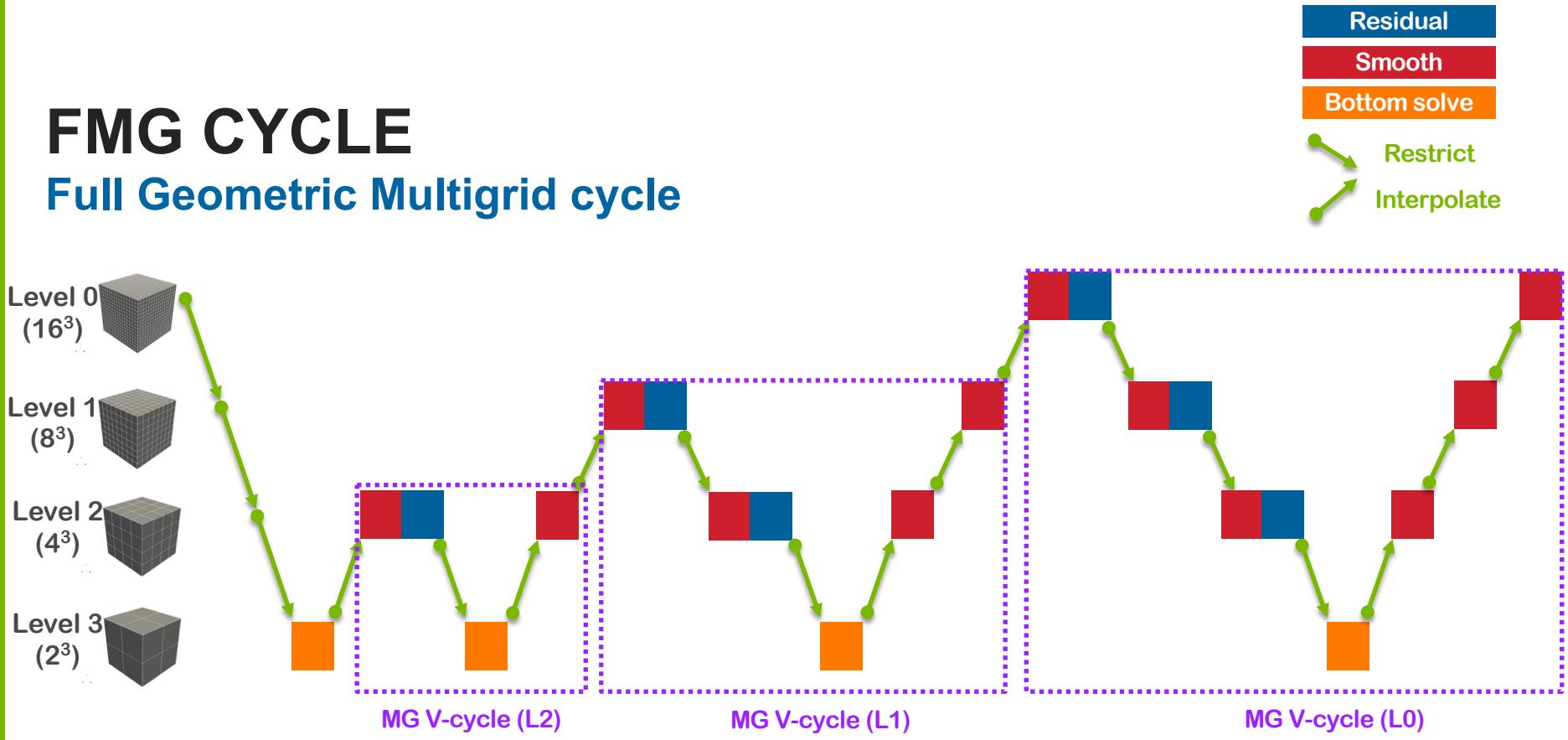
Smooth at L1: $Lu^{L1} = f^{L1}$

Residual at L1: $r^{L1} = f^{L1} - Lu^{L1}$



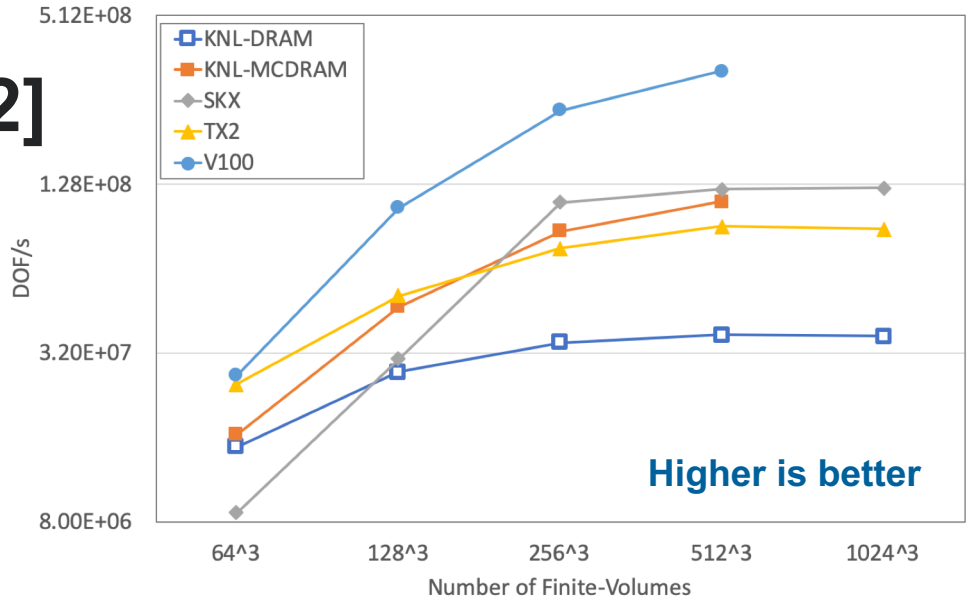
FMG CYCLE

Full Geometric Multigrid cycle



HPGMG-FV RESULTS [2]

- Source
  - MPI+OpenMP version for CPUs
  - MPI+CUDA version for NVIDIA GPUs
- Compilers
  - KNL / SKX : Intel 19.0.3.199
  - TX2: Arm Compiler version 19.0
  - V100: CUDA V10.0.130



- Inputs

Number of Finite-Volumes	Multi-grid Levels	Degrees-of-Freedom	Numerical Errors
64^3	6	2.62E+05	6.93E-05
128^3	7	2.10E+06	7.45E-06
256^3	8	1.68E+07	5.14E-07
512^3	9	1.34E+08	4.15E-08
1024^3	10	1.07E+09	5.15E-09

- Runtime configurations

Processor	Number of MPI ranks	Number of Threads per MPI rank	Total Threads
KNL	64	1	64
SKX	16	7	112
TX2	16	7	112
V100	1	7	all GPU cores

A STAND-ALONE MINI-APP WITH SMOOTH KERNEL

HPGMG TIMING BREAKDOWN

Smooth kernel time from HPGMG (log2BoxSize = 9)

```
===== Timing Breakdown ==========================================================
```

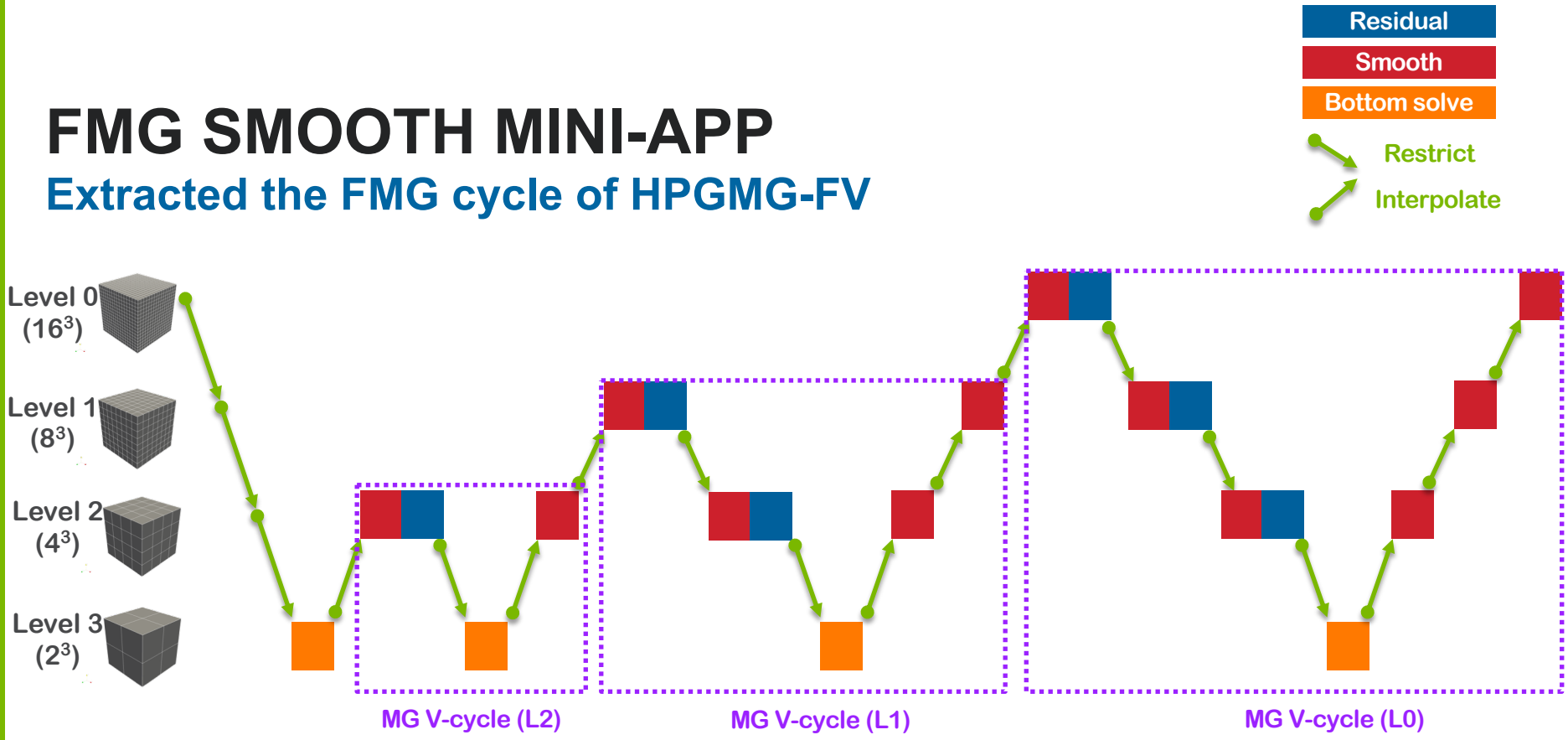
level	0	1	2	3	4	5	6	7	8	
level dimension	512^3	256^3	128^3	64^3	32^3	16^3	8^3	4^3	2^3	
box dimension	512^3	256^3	128^3	64^3	32^3	16^3	8^3	4^3	2^3	total
smooth	17.945751	4.274007	0.769077	0.125126	0.019930	0.003058	0.000491	0.000090	0.000000	23.137532
max	17.945751	4.274007	0.769077	0.125126	0.019930	0.003058	0.000491	0.000090	0.000000	23.137532
min	17.945751	4.274007	0.769077	0.125126	0.019930	0.003058	0.000491	0.000090	0.000000	23.137532
residual	4.780243	0.598737	0.112966	0.018437	0.002898	0.000434	0.000067	0.000014	0.000011	5.513808
applyOp	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000019	0.000019
BLAS1	0.730518	0.026407	0.005993	0.000983	0.000182	0.000052	0.000016	0.000007	0.000111	0.764268
BLAS3	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
Boundary Conditions	0.286837	0.143824	0.037334	0.005992	0.001699	0.000582	0.000309	0.000202	0.000061	0.476841
Restriction	0.213965	0.046598	0.007108	0.000644	0.000093	0.000024	0.000011	0.000009	0.000000	0.268451
local restriction	0.213965	0.046597	0.007107	0.000643	0.000092	0.000023	0.000010	0.000009	0.000000	0.268445
Interpolation	1.228388	0.195196	0.028880	0.003800	0.000543	0.000082	0.000019	0.000012	0.000000	1.456920
local interpolation	1.228388	0.195196	0.028879	0.003799	0.000543	0.000081	0.000018	0.000011	0.000000	1.456914
Ghost Zone Exchange	0.000000	0.000003	0.000005	0.000003	0.000003	0.000002	0.000003	0.000003	0.000002	0.000024
local exchange	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
Total by level	25.040127	5.256366	0.954373	0.153983	0.025155	0.004193	0.000913	0.000345	0.000195	31.435649

```
   Total time in MGBuild    27.591388 seconds
   Total time in MGSolve    31.435667 seconds
      number of v-cycles            1
Bottom solver iterations           16
```

Smooth kernel time / Total = 73.6%

FMG SMOOTH MINI-APP

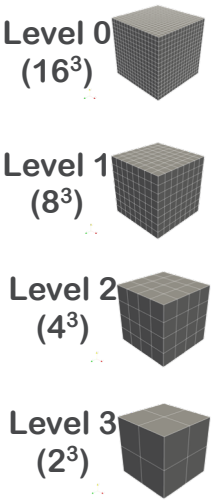
Extracted the FMG cycle of HPGMG-FV

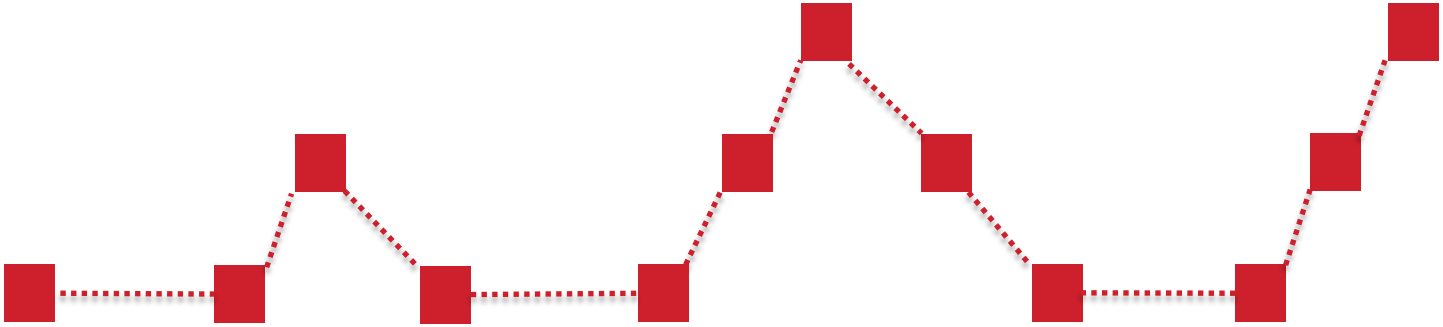


Smooth

FMG SMOOTH MINI-APP

Extracted the FMG cycle of HPGMG-FV





FMG smooth mini-app

Initialization
(e.g., grids for levels, memory allocations, a series of levels for smooth in a FMG cycle, and so on)

Loop for iterations (to measure sustained performance)

Loop for a series of levels for smooth in a FMG cycle (e.g., L2-L2-L1-L2-L2-L1-L0-L1-L2-L2-L1-L0 for 4 levels)

Loop for GSRB smooth (i.e. RBRBRB)

Finalization
(e.g., reporting smooth kernel time per level, memory deallocation, and so on)

```
int block,s;
for(s=0;s<2*NUM_SMOOTHS;s++){ // there are two sweeps per GSRB smooth
   double _timeStart = getTime();

   // loop over all block/tiles this process owns...
   for(block=0;block<level->num_my_blocks;block++){
      const int ilo = ilo_levels[ilevel][block];
      const int jlo = jlo_levels[ilevel][block];
      const int klo = klo_levels[ilevel][block];
      const int ihi = ihi_levels[ilevel][block];
      const int jhi = jhi_levels[ilevel][block];
      const int khi = khi_levels[ilevel][block];

      int i,j,k;
      const double h2inv = h2inv_levels[ilevel];
      const int jStride = jStride_levels[ilevel];
      const int kStride = kStride_levels[ilevel];
      // is element 000 red or black on *THIS* sweep
      const int color000 = (lowi_levels[ilevel]^lowj_levels[ilevel]^lowk_levels[ilevel]^s)&1;

      const double * __restrict__ rhs   = vectors[       rhs_id] + ghosts*(l+jStride+kStride);
      const double * __restrict__ beta_i= vectors[VECTOR_BETA_I] + ghosts*(l+jStride+kStride);
      const double * __restrict__ beta_j= vectors[VECTOR_BETA_J] + ghosts*(l+jStride+kStride);
      const double * __restrict__ beta_k= vectors[VECTOR_BETA_K] + ghosts*(l+jStride+kStride);
      const double * __restrict__ Dinv  = vectors[VECTOR_DINV  ] + ghosts*(l+jStride+kStride);
      const double * __restrict__ x_n;
      double * __restrict__ x_npl;
      if((s&1)==0){x_n      = vectors[         x_id] + ghosts*(l+jStride+kStride);
                   x_npl    = vectors[VECTOR_TEMP  ] + ghosts*(l+jStride+kStride);}
      else{        x_n      = vectors[VECTOR_TEMP  ] + ghosts*(l+jStride+kStride);
                   x_npl    = vectors[         x_id] + ghosts*(l+jStride+kStride);}

      for(k=klo;k<khi;k++){
      for(j=jlo;j<jhi;j++){
      // out-of-place must copy old value...
         for(i=ilo;i<ihi;i++){
            int ijk = i + j*jStride + k*kStride;
            x_npl[ijk] = x_n[ijk];
         }
         for(i=ilo+((ilo^j^k^color000)&1);i<ihi;i+=2){ // stride-2 GSRB
            int ijk = i + j*jStride + k*kStride;
            double Ax     = apply_op_ijk(x_n);
            x_npl[ijk] = x_n[ijk] + Dinv[ijk]*(rhs[ijk]-Ax);
         }
      }}

   } // boxes

   level->timers.smooth += (double)(getTime()-_timeStart);

} // s-loop
```

Loop for GSRB smooth (i.e. s from 0 to 6)

Loop for blocks (i.e. block from 0 to num_my_blocks)

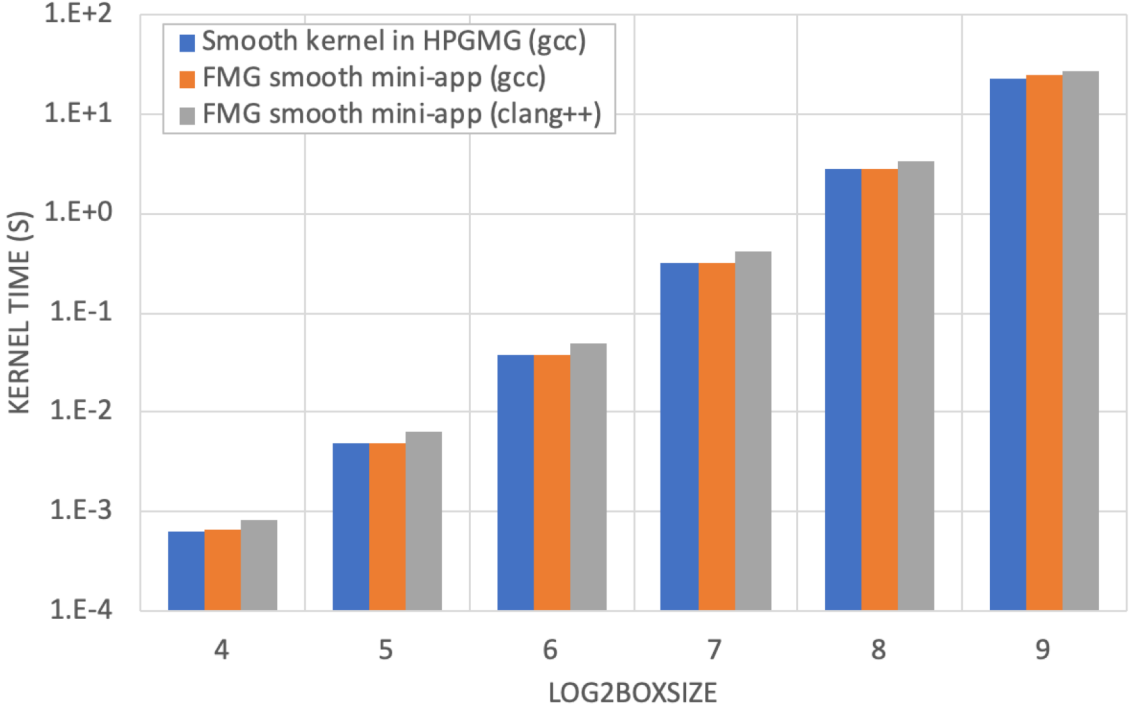
Loop for cells in the block (i.e. k,j,i from lo to hi)

```
#define apply_op_ijk(x)                                                                                                                            \
(                                                                                                                                                  \
 -b*h2inv*(                                                                                                                                        \
    STENCIL_TWELFTH*(                                                                                                                              \
      + beta_i[ijk        ]*( 15.0*(x[ijk-1      ]-x[ijk]) - (x[ijk-2        ]-x[ijk+1      ]) )                                                   \
      + beta_i[ijk+1      ]*( 15.0*(x[ijk+1      ]-x[ijk]) - (x[ijk+2        ]-x[ijk-1      ]) )                                                   \
      + beta_j[ijk        ]*( 15.0*(x[ijk-jStride]-x[ijk]) - (x[ijk-2*jStride]-x[ijk+jStride]) )                                                   \
      + beta_j[ijk+jStride]*( 15.0*(x[ijk+jStride]-x[ijk]) - (x[ijk+2*jStride]-x[ijk-jStride]) )                                                   \
      + beta_k[ijk        ]*( 15.0*(x[ijk-kStride]-x[ijk]) - (x[ijk-2*kStride]-x[ijk+kStride]) )                                                   \
      + beta_k[ijk+kStride]*( 15.0*(x[ijk+kStride]-x[ijk]) - (x[ijk+2*kStride]-x[ijk-kStride]) )                                                   \
    )                                                                                                                                              \
    + 0.25*STENCIL_TWELFTH*(                                                                                                                       \
      + (beta_i[ijk        +jStride]-beta_i[ijk        -jStride]) * (x[ijk-1      +jStride]-x[ijk+jStride]-x[ijk-1      -jStride]+x[ijk-jStride])  \
      + (beta_i[ijk        +kStride]-beta_i[ijk        -kStride]) * (x[ijk-1      +kStride]-x[ijk+kStride]-x[ijk-1      -kStride]+x[ijk-kStride])  \
      + (beta_j[ijk        +1      ]-beta_j[ijk        -1      ]) * (x[ijk-jStride+1      ]-x[ijk+1      ]-x[ijk-jStride-1      ]+x[ijk-1      ])  \
      + (beta_j[ijk        +kStride]-beta_j[ijk        -kStride]) * (x[ijk-jStride+kStride]-x[ijk+kStride]-x[ijk-jStride-kStride]+x[ijk-kStride])  \
      + (beta_k[ijk        +1      ]-beta_k[ijk        -1      ]) * (x[ijk-kStride+1      ]-x[ijk+1      ]-x[ijk-kStride-1      ]+x[ijk-1      ])  \
      + (beta_k[ijk        +jStride]-beta_k[ijk        -jStride]) * (x[ijk-kStride+jStride]-x[ijk+jStride]-x[ijk-kStride-jStride]+x[ijk-jStride])  \
                                                                                                                                                   \
      + (beta_i[ijk+1      +jStride]-beta_i[ijk+1      -jStride]) * (x[ijk+1      +jStride]-x[ijk+jStride]-x[ijk+1      -jStride]+x[ijk-jStride])  \
      + (beta_i[ijk+1      +kStride]-beta_i[ijk+1      -kStride]) * (x[ijk+1      +kStride]-x[ijk+kStride]-x[ijk+1      -kStride]+x[ijk-kStride])  \
      + (beta_j[ijk+jStride+1      ]-beta_j[ijk+jStride-1      ]) * (x[ijk+jStride+1      ]-x[ijk+1      ]-x[ijk+jStride-1      ]+x[ijk-1      ])  \
      + (beta_j[ijk+jStride+kStride]-beta_j[ijk+jStride-kStride]) * (x[ijk+jStride+kStride]-x[ijk+kStride]-x[ijk+jStride-kStride]+x[ijk-kStride])  \
      + (beta_k[ijk+kStride+1      ]-beta_k[ijk+kStride-1      ]) * (x[ijk+kStride+1      ]-x[ijk+1      ]-x[ijk+kStride-1      ]+x[ijk-1      ])  \
      + (beta_k[ijk+kStride+jStride]-beta_k[ijk+kStride-jStride]) * (x[ijk+kStride+jStride]-x[ijk+jStride]-x[ijk+kStride-jStride]+x[ijk-jStride])  \
    )                                                                                                                                              \
  )                                                                                                                                                \
)
```

BASELINE PERFORMANCE

Smooth kernel in HPGMG vs. FMG smooth mini-app

- Test configuration
  - Processor
    - Intel Skylake 8180M
  - Compilers
    - gcc/8.2.0
    - clang++/12.0.0: Intel public SYCL compiler



SYCL IMPLEMENTATION

Creating a queue & setting up buffers

Initialization
(e.g., grids for levels, memory allocations, a series of levels for smooth in a FMG cycle, and so on)

Scope for SYCL execution

Create a queue and set up buffers

Loop for iterations (to measure sustained performance)

Loop for a series of levels for smooth in a FMG cycle (e.g., L2-L2-L1-L2-L2-L1-L0-L1-L2-L2-L1-L0 for 4 levels)

Loop for GSRB smooth (i.e. RBRBRB)

Submit the queue and add accessors to the queue

Finalization
(e.g., reporting smooth kernel time per level, memory deallocation, and so on)

Creating a queue

```
// SYCL
queue myQueue;
auto dev = myQueue.get_device();
auto ctxt = myQueue.get_context();
```

Setting up buffers

```
//
// Set up buffers for SYCL
//
buffer<double, 1> fp_base_all_buf(fp_base_all, range<1>(fp_base_all_size));
buffer<double, 1> b_buf(&b, range<1>(1));
buffer<double, 1> h2inv_levels_buf(h2inv_levels, range<1>(num_levels));
buffer<int, 1> jStride_levels_buf(jStride_levels, range<1>(num_levels));
buffer<int, 1> kStride_levels_buf(kStride_levels, range<1>(num_levels));
buffer<int, 1> lowi_levels_buf(lowi_levels, range<1>(num_levels));
buffer<int, 1> lowj_levels_buf(lowj_levels, range<1>(num_levels));
buffer<int, 1> lowk_levels_buf(lowk_levels, range<1>(num_levels));
buffer<int, 1> ilo_base_buf(ilo_base, range<1>(num_levels_blocks));
buffer<int, 1> jlo_base_buf(jlo_base, range<1>(num_levels_blocks));
buffer<int, 1> klo_base_buf(klo_base, range<1>(num_levels_blocks));
buffer<int, 1> ihi_base_buf(ihi_base, range<1>(num_levels_blocks));
buffer<int, 1> jhi_base_buf(jhi_base, range<1>(num_levels_blocks));
buffer<int, 1> khi_base_buf(khi_base, range<1>(num_levels_blocks));
buffer<int, 1> levels_ijk_lohi_base_id0_buf(levels_ijk_lohi_base_id0,\
                                            range<1>(num_levels));
buffer<uint64_t, 1> levels_vectors_id0_buf(levels_vectors_id0, \
                                            range<1>(num_levels));
buffer<int, 1> levels_box_volume_buf(levels_box_volume, range<1>(num_levels));
```

Submitting the queue & adding accessors to the queue

Initialization
(e.g., grids for levels, memory allocations, a series of levels for smooth in a FMG cycle, and so on)

Scope for SYCL execution

Create a queue and set up buffers

Loop for iterations (to measure sustained performance)

Loop for a series of levels for smooth in a FMG cycle (e.g., L2-L2-L1-L2-L2-L1-L0-L1-L2-L2-L1-L0 for 4 levels)

Loop for GSRB smooth (i.e. RBRBRB)

Submit the queue and add accessors to the queue

Finalization
(e.g., reporting smooth kernel time per level, memory deallocation, and so on)

```
int block,s;
for(s=0;s<2*NUM_SMOOTHS;s++){ // there are two sweeps per GSRB smooth
  TIC_smooth = getTime();

  myQueue.submit([&](handler& h) {  // start of the submit

    auto fp_base_all_d = fp_base_all_buf.get_access<access::mode::read_write>(h);
    auto b_d = b_buf.get_access<access::mode::read>(h);
    auto h2inv_d = h2inv_levels_buf.get_access<access::mode::read>(h);
    auto jStride_d = jStride_levels_buf.get_access<access::mode::read>(h);
    auto kStride_d = kStride_levels_buf.get_access<access::mode::read>(h);
    auto lowi_d = lowi_levels_buf.get_access<access::mode::read>(h);
    auto lowj_d = lowj_levels_buf.get_access<access::mode::read>(h);
    auto lowk_d = lowk_levels_buf.get_access<access::mode::read>(h);
    auto ilo_base_d = ilo_base_buf.get_access<access::mode::read>(h);
    auto jlo_base_d = jlo_base_buf.get_access<access::mode::read>(h);
    auto klo_base_d = klo_base_buf.get_access<access::mode::read>(h);
    auto ihi_base_d = ihi_base_buf.get_access<access::mode::read>(h);
    auto jhi_base_d = jhi_base_buf.get_access<access::mode::read>(h);
    auto khi_base_d = khi_base_buf.get_access<access::mode::read>(h);
    auto levels_ijk_lohi_base_id0_d = \
            levels_ijk_lohi_base_id0_buf.get_access<access::mode::read>(h);
    auto levels_vectors_id0_d = levels_vectors_id0_buf.get_access<access::mode::read>(h);
    auto levels_box_volume_d = levels_box_volume_buf.get_access<access::mode::read>(h);
```

Submitting queue & adding accessors

```
    h.parallel_for<class smooth_a_block>(range<3>(1,1,level->num_my_blocks), // global range
        id<3>(ilevel,s,0),   // offset
        [=](item<3> item) {
      int ilevel = item[0];
      int s = item[1];
      int block = item[2];
      int numVectors = VECTORS_RESERVED;
      int box_volume = levels_box_volume_d[ilevel];
      double b = b_d[0];

      const int ilo = ilo_base_d[ levels_ijk_lohi_base_id0_d[ilevel]+block ];
      const int jlo = jlo_base_d[ levels_ijk_lohi_base_id0_d[ilevel]+block ];
      const int klo = klo_base_d[ levels_ijk_lohi_base_id0_d[ilevel]+block ];
      const int ihi = ihi_base_d[ levels_ijk_lohi_base_id0_d[ilevel]+block ];
      const int jhi = jhi_base_d[ levels_ijk_lohi_base_id0_d[ilevel]+block ];
      const int khi = khi_base_d[ levels_ijk_lohi_base_id0_d[ilevel]+block ];

      int i,j,k;
      const double h2inv = h2inv_d[ilevel];
      const int jStride = jStride_d[ilevel];
      const int kStride = kStride_d[ilevel];
      // is element 000 red or black on *THIS* sweep
      const int color000 = (lowi_d[ilevel]^lowj_d[ilevel]^lowk_d[ilevel]^s)&1;

      double * vectors[VECTORS_RESERVED];
      for(int v=0;v<numVectors;v++){vectors[v] = &fp_base_all_d[0] +\
          levels_vectors_id0_d[ilevel] + (uint64_t)box_volume*v;} // setup vector pointers
```

parallel_for in the queue

```
// continue to the next page
```

Submitting the queue & adding accessors to the queue

Initialization
(e.g., grids for levels, memory allocations, a series of levels for smooth in a FMG cycle, and so on)

Scope for SYCL execution

Create a queue and set up buffers

Loop for iterations (to measure sustained performance)

Loop for a series of levels for smooth in a FMG cycle (e.g., L2-L2-L1-L2-L2-L1-L0-L1-L2-L2-L1-L0 for 4 levels)

Loop for GSRB smooth (i.e. RBRBRB)

Submit the queue and add accessors to the queue

Finalization
(e.g., reporting smooth kernel time per level, memory deallocation, and so on)

```
// continue from the previous page

    const double * __restrict__ rhs   = vectors[VECTOR_R     ]+ghosts*(1+jStride+kStride);
    const double * __restrict__ beta_i= vectors[VECTOR_BETA_I]+ghosts*(1+jStride+kStride);
    const double * __restrict__ beta_j= vectors[VECTOR_BETA_J]+ghosts*(1+jStride+kStride);
    const double * __restrict__ beta_k= vectors[VECTOR_BETA_K]+ghosts*(1+jStride+kStride);
    const double * __restrict__ Dinv  = vectors[VECTOR_DINV  ]+ghosts*(1+jStride+kStride);
    const double * __restrict__ x_n;
          double * __restrict__ x_np1;
                   if((s&1)==0){x_n   = vectors[VECTOR_U     ]+ghosts*(1+jStride+kStride);
                                x_np1 = vectors[VECTOR_TEMP  ]+ghosts*(1+jStride+kStride);}
                         else{x_n   = vectors[VECTOR_TEMP  ]+ghosts*(1+jStride+kStride);
                                x_np1 = vectors[VECTOR_U     ]+ghosts*(1+jStride+kStride);}
    for(k=klo;k<khi;k++){
    for(j=jlo;j<jhi;j++){
      // out-of-place must copy old value...
      for(i=ilo;i<ihi;i++){
        int ijk = i + j*jStride + k*kStride;
        x_np1[ijk] = x_n[ijk];
      }
      for(i=ilo+((ilo^j^k^color000)&1);i<ihi;i+=2){ // stride-2 GSRB
        int ijk = i + j*jStride + k*kStride;
        double Ax     = apply_op_ijk(x_n);
        x_np1[ijk] = x_n[ijk] + Dinv[ijk]*(rhs[ijk]-Ax);
      }
    }}
                                                          End of parallel_for in the queue
  });    // End of smooth_a_block kernel

 });  // end of the submit
                                                          End of the queue submission
 TOC_smooth = getTime();
 timer = TOC_smooth - TIC_smooth;
 level->timers.smooth += (double)(timer.count());

} // s-loop
```

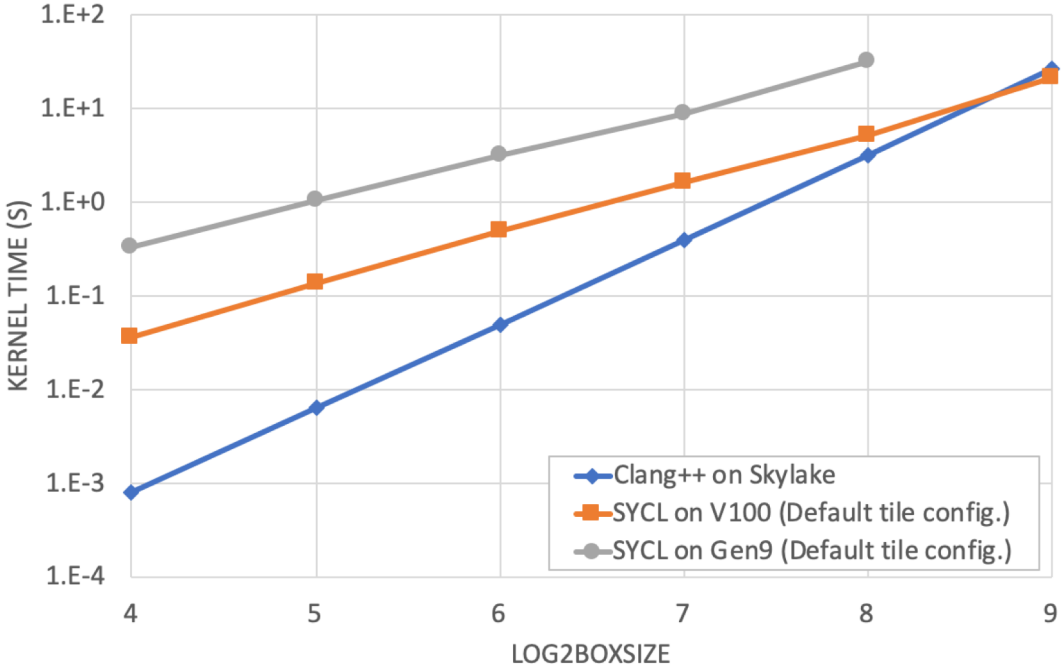
TEST BED SYSTEMS IN ARGONNE JLSE [3]

Intel Gen9 integrated GPU vs. NVIDIA V100 discrete GPU

JLSE Nodes	GPU_V100_SMX2	IRIS
HOST	Intel Xeon Gold 6152	Intel Xeon E3-1585 v5
GPU	NVIDIA V100	Intel Gen9 GT4e
GPU DP peak	7.8 TFLOPS	0.33 TFLOPS
GPU BW peak	900 GB/s	34.1 GB/s
SYCL Compiler	Public LLVM SYCL-CUDA	Public LLVM SYCL-Level0

SYCL PERFORMANCE

w/ default tile configuration (i.e., 10000x8x8)



Level	Number of blocks
1	1
2	1
3	1
4	4
5	16
6	64
7	256
8	1024
9	4096

Not sufficient items for parallel_for

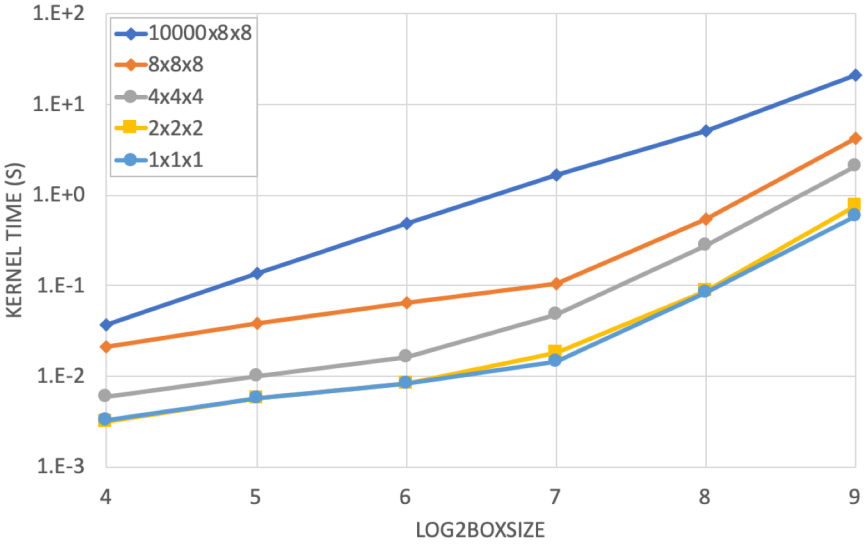
TILE CONFIGURATION VS. # OF BLOCKS

Level	Tile configurations				
	10000x8x8	8x8x8	4x4x4	2x2x2	1x1x1
1	1	1	1	1	8
2	1	1	1	8	64
3	1	1	8	64	512
4	4	8	64	512	4,096
5	16	64	512	4,096	32,768
6	64	512	4,096	32,768	262,144
7	256	4,096	32,768	262,144	2,097,152
8	1,024	32,768	262,144	2,097,152	16,777,216
9	4,096	262,144	2,097,152	16,777,216	134,217,728

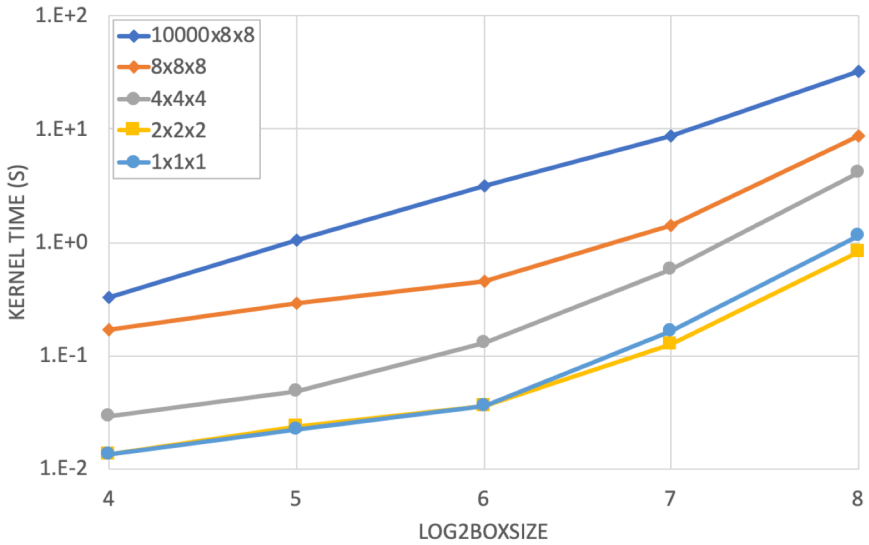
More items for parallel_for

SYCL PERFORMANCE

with various tile configurations



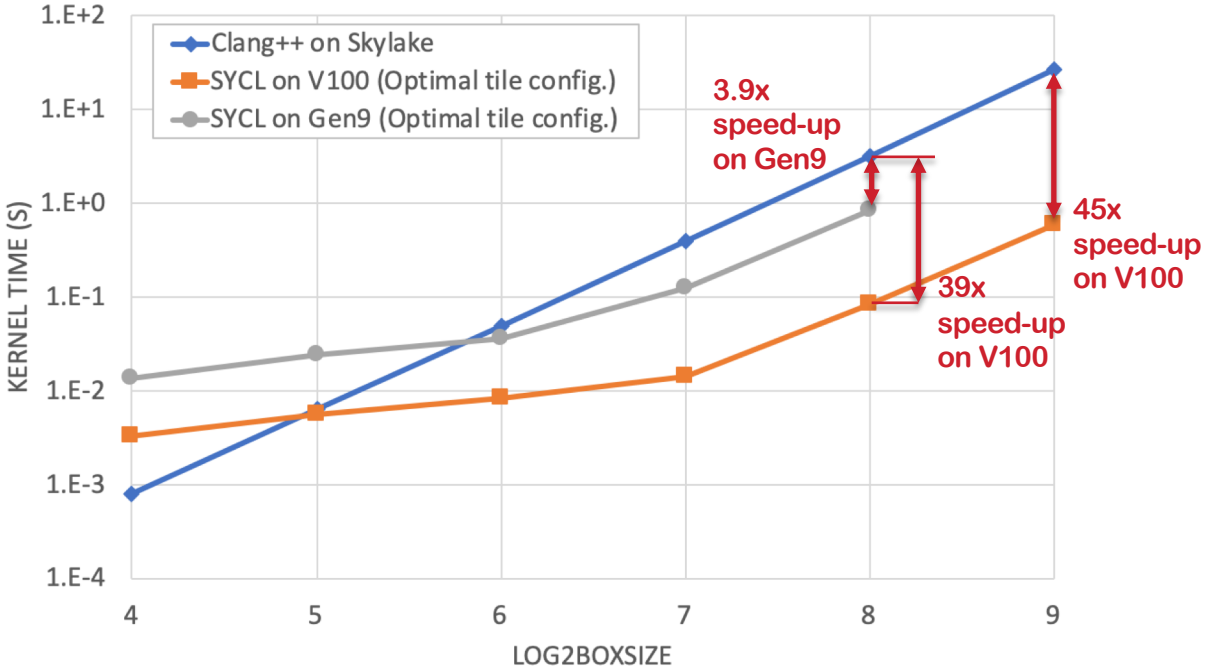
On NVIDIA V100 GPU



On Intel Gen9 GPU

SYCL PERFORMANCE COMPARISON

w/ best tile configurations for GPUs



OPENMP TARGET OFFLOADING IMPLEMENTATION

omp target enter/exit data map

Initialization
(e.g., grids for levels, memory allocations, a series of levels for smooth in a FMG cycle, and so on)

Transfer data from host to device

```
#pragma omp target enter data map(to: fp_base_all[0:fp_base_all_size])
#pragma omp target enter data map(to: h2inv_levels[0:num_levels])
#pragma omp target enter data map(to: jStride_levels[0:num_levels])
#pragma omp target enter data map(to: kStride_levels[0:num_levels])
#pragma omp target enter data map(to: lowi_levels[0:num_levels])
#pragma omp target enter data map(to: lowj_levels[0:num_levels])
#pragma omp target enter data map(to: lowk_levels[0:num_levels])
#pragma omp target enter data map(to: ilo_base[0:num_levels_blocks])
#pragma omp target enter data map(to: jlo_base[0:num_levels_blocks])
#pragma omp target enter data map(to: klo_base[0:num_levels_blocks])
#pragma omp target enter data map(to: ihi_base[0:num_levels_blocks])
#pragma omp target enter data map(to: jhi_base[0:num_levels_blocks])
#pragma omp target enter data map(to: khi_base[0:num_levels_blocks])
#pragma omp target enter data map(to: levels_ijk_lohi_base_id0[0:num_levels])
#pragma omp target enter data map(to: levels_vectors_id0[0:num_levels])
#pragma omp target enter data map(to: levels_box_volume[0:num_levels])
```

Loop for iterations (to measure sustained performance)

Loop for a series of levels for smooth in a FMG cycle (e.g., L2-L2-L1-L2-L2-L1-L0-L1-L2-L2-L1-L0 for 4 levels)

Loop for GSRB smooth (i.e. RBRBRB)

Add omp teams distribute parallel for

Transfer data from device to host

```
#pragma omp target exit data map(release: fp_base_all[0:fp_base_all_size])
#pragma omp target exit data map(release: h2inv_levels[0:num_levels])
#pragma omp target exit data map(release: jStride_levels[0:num_levels])
#pragma omp target exit data map(release: kStride_levels[0:num_levels])
#pragma omp target exit data map(release: lowi_levels[0:num_levels])
#pragma omp target exit data map(release: lowj_levels[0:num_levels])
#pragma omp target exit data map(release: lowk_levels[0:num_levels])
#pragma omp target exit data map(release: ilo_base[0:num_levels_blocks])
#pragma omp target exit data map(release: jlo_base[0:num_levels_blocks])
#pragma omp target exit data map(release: klo_base[0:num_levels_blocks])
#pragma omp target exit data map(release: ihi_base[0:num_levels_blocks])
#pragma omp target exit data map(release: jhi_base[0:num_levels_blocks])
#pragma omp target exit data map(release: khi_base[0:num_levels_blocks])
#pragma omp target exit data map(release: levels_ijk_lohi_base_id0[0:num_levels])
#pragma omp target exit data map(release: levels_vectors_id0[0:num_levels])
#pragma omp target exit data map(release: levels_box_volume[0:num_levels])
```

Finalization
(e.g., reporting smooth kernel time per level, memory deallocation, and so on)

omp target teams distribute parallel for

Initialization
(e.g., grids for levels, memory allocations, a series of levels for smooth in a FMG cycle, and so on)

Copy data from host to device

Loop for iterations (to measure sustained performance)

Loop for a series of levels for smooth in a FMG cycle (e.g., L2-L2-L1-L2-L2-L1-L0-L1-L2-L2-L1-L0 for 4 levels)

Loop for GSRB smooth (i.e. RBRBRB)

Add omp teams distribute parallel for

Copy data from host to device

Finalization
(e.g., reporting smooth kernel time per level, memory deallocation, and so on)

```
int block,s;
for(s=0;s<2*NUM_SMOOTHS;s++){ // there are two sweeps per GSRB smooth
   double _timeStart = getTime();

   // loop over all block/tiles this process owns...
   #pragma omp target teams distribute parallel for
   for(block=0;block<level->num_my_blocks;block++){
      const int ilo = ilo_levels[ilevel][block];
      const int jlo = jlo_levels[ilevel][block];
      const int klo = klo_levels[ilevel][block];
      const int ihi = ihi_levels[ilevel][block];
      const int jhi = jhi_levels[ilevel][block];
      const int khi = khi_levels[ilevel][block];

      int i,j,k;
      const double h2inv = h2inv_levels[ilevel];
      const int jStride = jStride_levels[ilevel];
      const int kStride = kStride_levels[ilevel];
      // is element 000 red or black on *THIS* sweep
      const int color000 = (lowi_levels[ilevel]^lowj_levels[ilevel]^lowk_levels[ilevel]^s)&1;

      const double * __restrict__ rhs   = vectors[       rhs_id] + ghosts*(1+jStride+kStride);
      const double * __restrict__ beta_i= vectors[VECTOR_BETA_I] + ghosts*(1+jStride+kStride);
      const double * __restrict__ beta_j= vectors[VECTOR_BETA_J] + ghosts*(1+jStride+kStride);
      const double * __restrict__ beta_k= vectors[VECTOR_BETA_K] + ghosts*(1+jStride+kStride);
      const double * __restrict__ Dinv  = vectors[VECTOR_DINV  ] + ghosts*(1+jStride+kStride);
      const double * __restrict__ x_n;
      double * __restrict__ x_npl;
      if((s&1)==0){x_n      = vectors[         x_id] + ghosts*(1+jStride+kStride);
                   x_npl    = vectors[VECTOR_TEMP  ] + ghosts*(1+jStride+kStride);}
      else{        x_n      = vectors[VECTOR_TEMP  ] + ghosts*(1+jStride+kStride);
                   x_npl    = vectors[         x_id] + ghosts*(1+jStride+kStride);}

      for(k=klo;k<khi;k++){
      for(j=jlo;j<jhi;j++){
      // out-of-place must copy old value...
         for(i=ilo;i<ihi;i++){
            int ijk = i + j*jStride + k*kStride;
            x_npl[ijk] = x_n[ijk];
         }
         for(i=ilo+((ilo^j^k^color000)&1);i<ihi;i+=2){ // stride-2 GSRB
            int ijk = i + j*jStride + k*kStride;
            double Ax     = apply_op_ijk(x_n);
            x_npl[ijk] = x_n[ijk] + Dinv[ijk]*(rhs[ijk]-Ax);
         }
      }}

   } // boxes

   level->timers.smooth += (double)(getTime()-_timeStart);

} // s-loop
```

BUILT/TESTED IT ON GEN9 AND V100

Compiler	Test bed	Results
Aurora SDK	Intel Gen9 GPU	Got good results, but requires approval for public release
LLVM/ clang-11.0.0	NVIDIA V100 GPU	Libomptarget fatal error
IBM/ xlc-16.1.1	NVIDIA V100 GPU	illegal memory access issue

Possibly using pointers inside of the target regions could be problematic with some compilers, so the following code (on the next page) was tested.

Test code with a pointer in a target region

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <omp.h>
#define n_base 100

int main(int argc, char *argv[]){

  double * base;
  unsigned long int n_base2=n_base*n_base;
  base = (double *)malloc(n_base2*sizeof(double));

  for (int i=0;i<n_base2;i++) {base[i] = 0.0;}            // Initialization

  #pragma omp target enter data map(to:base[0:n_base2])   // Target enter data

  // loop over all block/tiles this process owns...
  #pragma omp target teams distribute parallel for
  for(int i=0;i<n_base;i++){
    double * vectors[n_base];
    for(int j=0;j<n_base;j++) {vectors[j] = &base[0] + j*n_base;};
    double * __restrict__ base_tmp = vectors[i];
    for(int j=0;j<n_base;j++) {base_tmp[j] = (double)(i*n_base+j);}
  }

  // Target exit data
  #pragma omp target exit data map(from:base[0:n_base2])

  // Print out the results, deallocate base, and return
  for(int i=n_base2-10;i<n_base2;i++) {printf("base[%d]=%lf\n",i,base[i]);}
  free(base);  return(0);
}
```

Compiler	GPU	Max array size
xlc	NVIDIA V100	1102^2
clang	NVIDIA V100	22116^2
Aurora SDK	Intel Gen9	Better than clang

With the same code, compilers have different capacities to handle an array with pointers in the device.

CONCLUDING REMARKS

CONCLUDING REMARKS

- FMG Smooth mini-app has been extracted from HPGMG-FV.
- FMG Smooth mini-app has been implemented with SYCL
  - Tested with public LLVM SYCL compilers on NVIDIA V100 and Intel Gen9 GPUs
  - With a large number of blocks, the SYCL version shows meaningful speed-up on NVIDIA V100 and Intel Gen9 GPUs
- FMG Smooth mini-app has been implemented with OpenMP Target Offloading.
  - Successfully, built and tested it on Intel Gen9
  - On NVIDIA V100, LLVM clang and IBM xlc compilers returned errors.
  - Via a test code, it turns out that compilers show different capacity to handle an array with pointers in a target region.

NEXT STEP

- SYCL implementation
  - Performance optimization with roofline analysis
    - Intel Advisor roofline feature on Intel Gen9 GPU
    - Via NVProf/NSight on NVIDIA V100
  - Include more kernels from HPGMG-FV
- OpenMP Target offloading implementation
  - Investigating the issue with clang and xlc on V100
- Kokkos implementation (Working-in-progress)

ACKNOWLEDGEMENT

- This work was supported by
  - the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357,
  - and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration).
- We also gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

REFERENCES

1. “HPGMG webpage,” https://hpgmg.org, 2020.
2. J. Kwack, T. Applencourt, C. Bertoni, Y. Ghadar, H. Zheng, C. Knight, and S. Parker, “Roofline-based performance efficiency of HPC benchmarks and applications on current generation of processor architectures”, Cray User Group (CUG) meeting, Montreal, Canada, 2019
3. “JLSE webpage”, https://www.jlse.anl.gov, 2020.

THANKS!