GPU IMPLEMENTATION OF HPGMG-FV

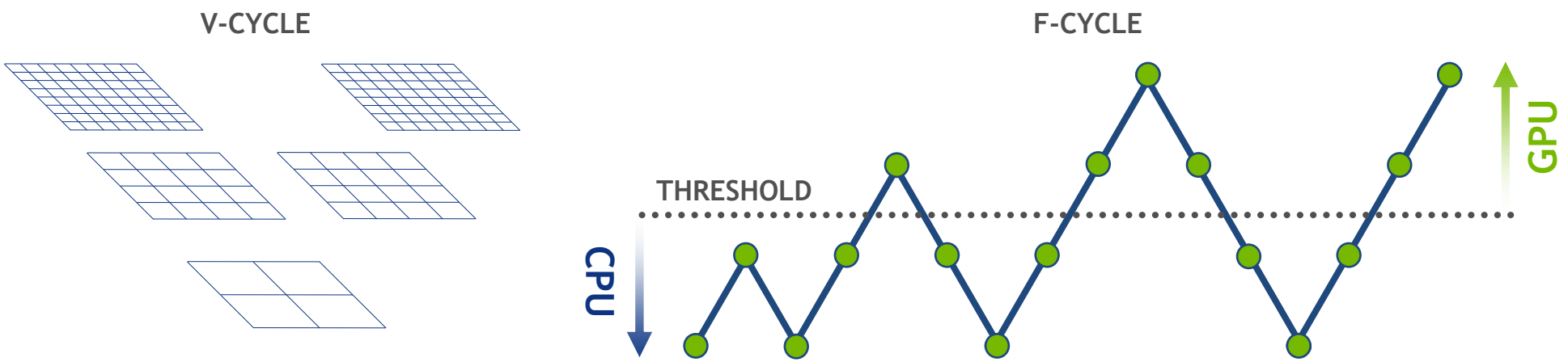
Nikolay Sakharnykh, Simon Layton, Kate Clark

NVIDIA

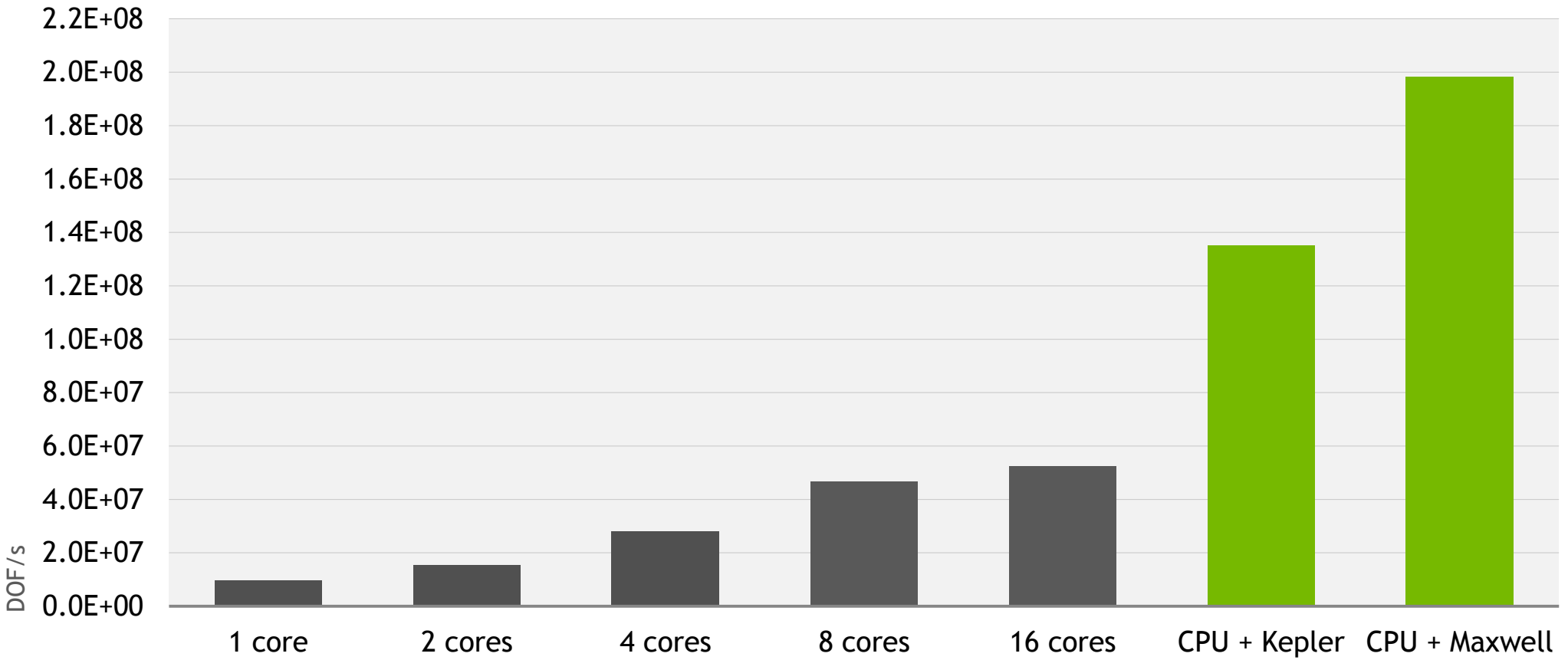
HYBRID MULTIGRID

Heterogeneous implementation using CUDA + OpenMP

Fine grids are offloaded to GPU (TOC), **coarse** grids are processed on CPU (LOC)



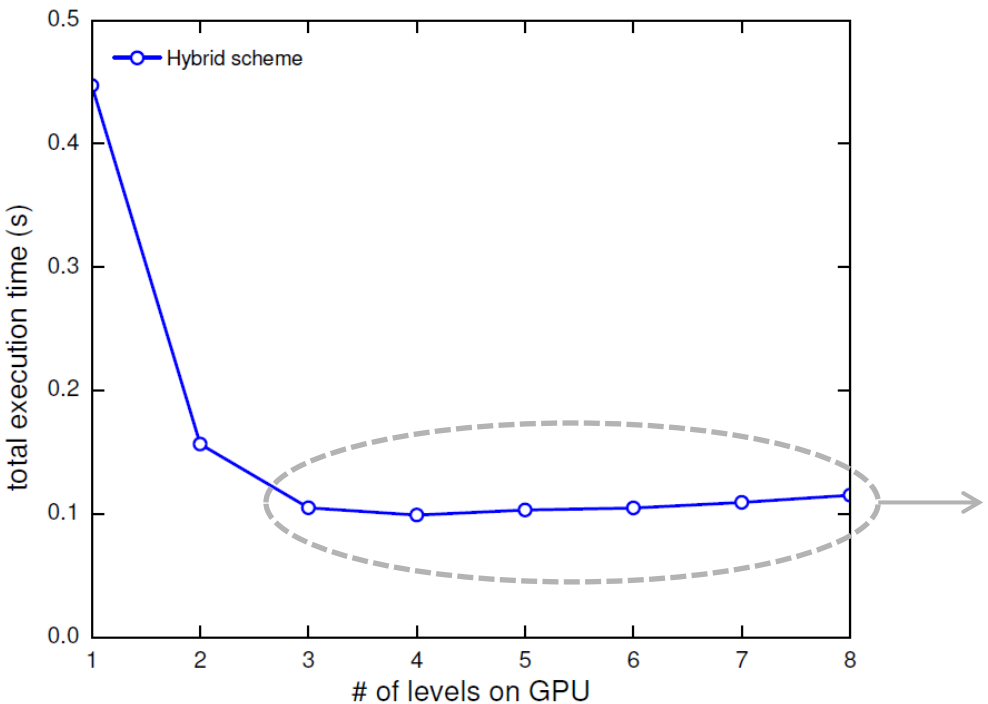
HYBRID MULTIGRID

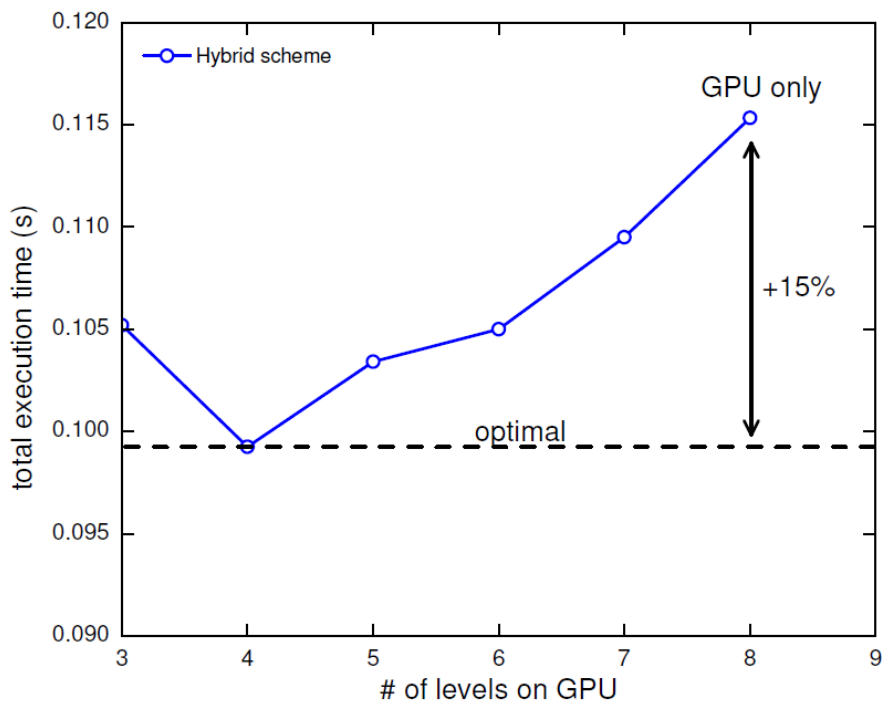


CPU-only: *Haswell Xeon E5-2698 v3, 16-core, single socket*
CPU + Kepler GPU: *Ivy Bridge Xeon E5-2690 v2, 10-core, single socket + NVIDIA Tesla K40 with ECC on*
CPU + Maxwell GPU: *Sandy Bridge i7-3930K, 6-core + NVIDIA GeForce Titan X*

HYBRID MULTIGRID

Threshold benchmark

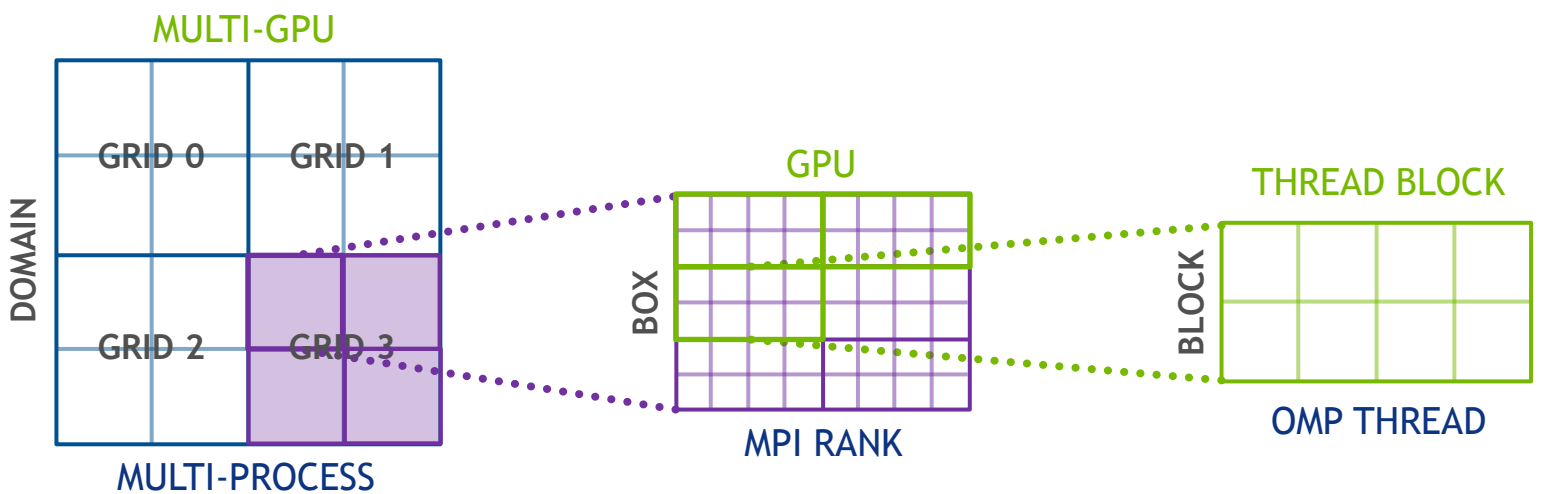




GPU: NVIDIA Quadro M6000, **CPU:** Intel Ivy Bridge Xeon E5-2690 v2

DATA STRUCTURES

HPGMG-FV entities naturally map to GPU hierarchy



UNIFIED MEMORY

No changes to data structures

No explicit data movements

Single pointer for CPU and GPU data

```
void  smooth(level_type *level,...){
...
if(level->use_cuda) {
   // run on GPU
   cuda_cheby_smooth(level,...);
}
else {
   // run on CPU
   #pragma omp parallel for
   for(block = 0; block < num_blocks; block++)
      ...
}}
```

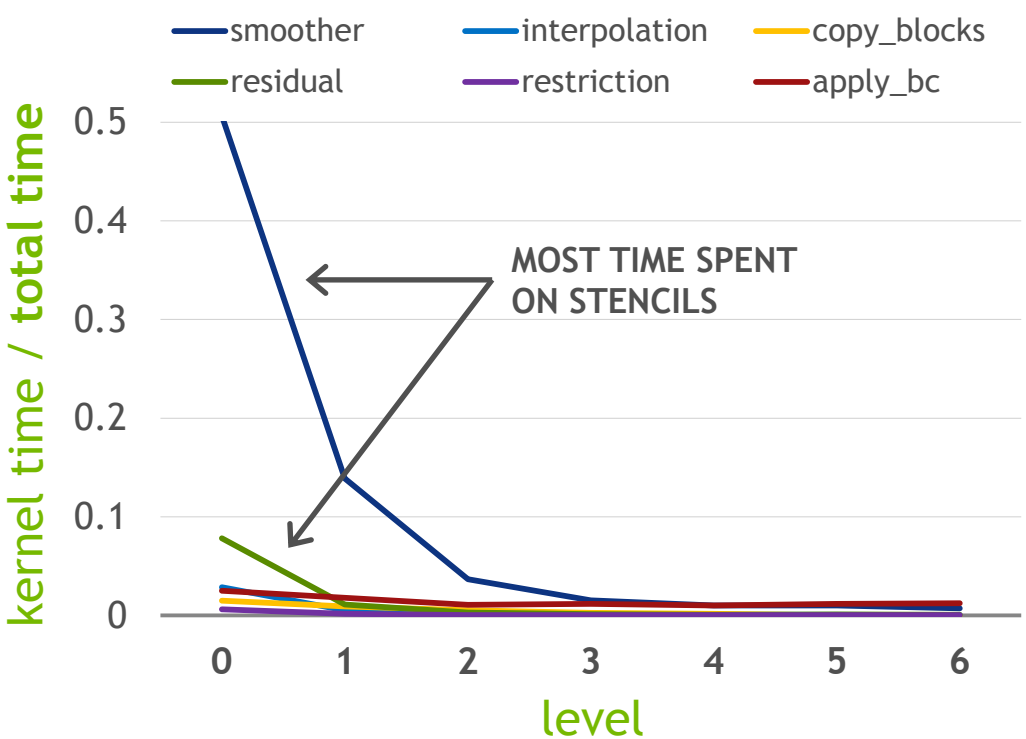
Minimal modifications to the original code:

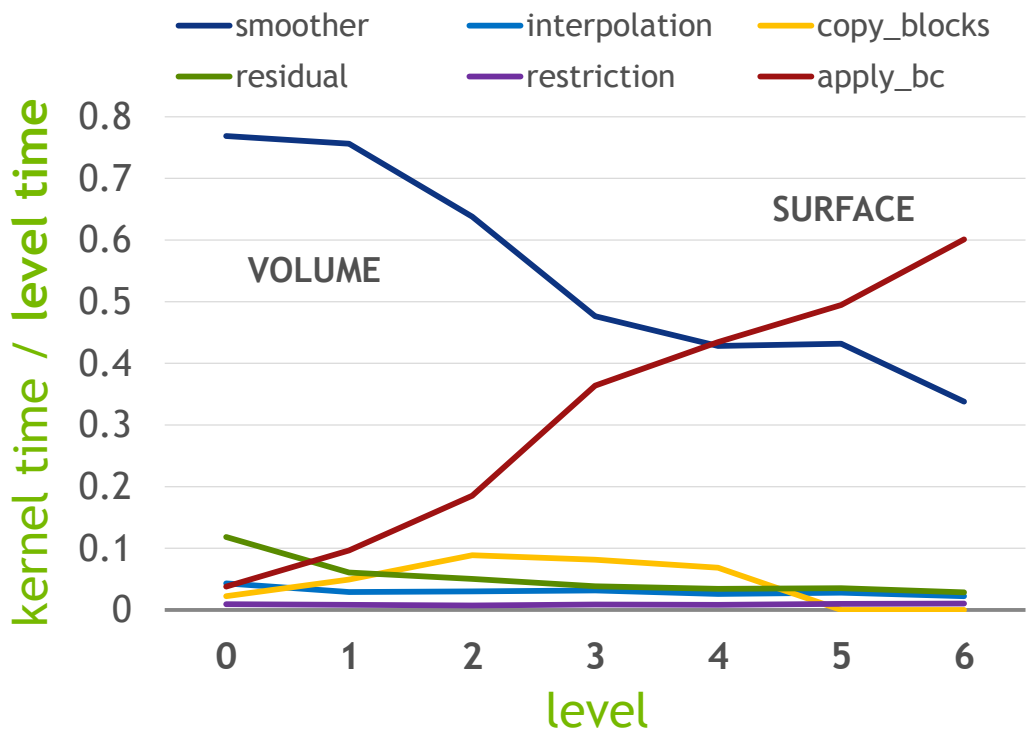
(1) malloc replaced with cudaMallocManaged for levels accessed by GPU

(2) Invoke CUDA kernel if level size is greater than threshold

COMPUTE BOTTLENECK

Cost of operations



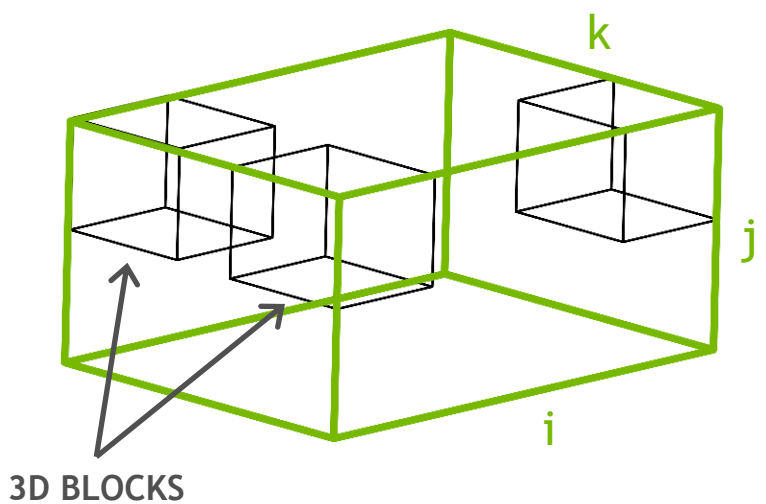


STENCILS ON GPU

Parallelization strategies

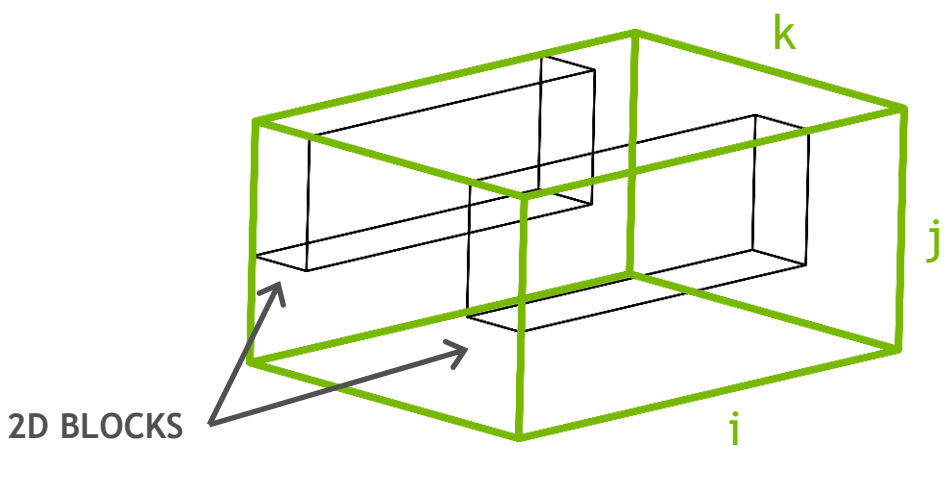
3D thread blocks

Provides more than enough parallelism



2D thread blocks

Parallelize over XY plane and march along Z



STENCILS ON GPU

One kernel for many smoothers

THREAD BLOCK MARCHES FROM 0 TO DIMZ

```
void smooth_kernel(level_type level,...) {
// prologue
const double * __restrict__ rhs = ...;
...

for(k=0; k<dimz; k++) {
   // apply operator
   const double Ax = apply_op_ijk();

   // smoother
   #ifdef CHEBY
   xo[ijk] = x[ijk] + ... + c2*lambda*(rhs[ijk]-Ax);
   #elif JACOBI
   xo[ijk] = x[ijk] + (2.0/3.0)*lambda*(rhs[ijk]-Ax);
   #elif GSRB
   xo[ijk] = x[ijk] + RB[ij]*lambda*(rhs[ijk]-Ax);
   #endif
}}
```

APPLY STENCIL OPERATION POISSON, HELMHOLTZ (FV2,FV4)

CHEBYSHEV POLYNOMIALS

JACOBI

GAUSS SEIDEL RED-BLACK

STENCILS ON GPU

Optimization: register caching

38 REGS IN KERNEL WITHOUT STENCIL

```
// load k and k-1 planes into registers
double xc0 = x[ijk - kStride];
double xc1 = x[ijk]; ...

for(k=0; k<dimz; k++) {
   // load k+1 plane into registers
   xc2 = x[ijk + kStride]; ...

   // apply operator
   const double Ax = apply_op_ijk();

   // smoother
   xo[ijk] = xc1 + ...;

   // update k and k-1 planes in registers
   xc0 = xc1; xc1 = xc2; ...
}}
```

TOTAL REG USAGE: 56 FOR FV2 AND 128 FOR FV4

7-POINT STENCIL, 18 REGS

```
const double Ax  =
-b*h2inv*(
STENCIL_TWELFTH*(
+ bir1 * (xr1 - xc1)
+ bic1 * (zl1 - xc1)
+ bju1 * (zu1 - xc1)
+ bjc1 * (zd1 - xc1)
+ bkc2 * (xc2 - xc1)
+ bkc1 * (xc0 - xc1)
);
```

up to 1.5x speed-up!

4TH ORDER STENCIL, 90 REGS

```
const double Ax  =
-b*h2inv*(
STENCIL_TWELFTH*(
+ bic1 * ( 15.0*(xl1-xc1) - (xll-xr1) )
+ bir1 * ( 15.0*(xr1-xc1) - (xrr-xl1) )
+ bjc1 * ( 15.0*(xu1-xc1) - (xuu-xd1) )
+ bjd1 * ( 15.0*(xd1-xc1) - (xdd-xu1) )
+ bkc1 * ( 15.0*(xc0-xc1) - (xbb-xc2) )
+ bkc2 * ( 15.0*(xc2-xc1) - (xff-xc0) ) )

+ 0.25*STENCIL_TWELFTH*(
+ (bid  - biu ) * (xld - xd1 - xlu + xu1)
+ (bic2 - bic0) * (xl2 - xc2 - xl0 + xc0)
+ (bjr  - bjl ) * (xru - xr1 - xlu + xl1)
+ (bjc2 - bjc0) * (xu2 - xc2 - xu0 + xc0)
+ (bkr1 - bkl1) * (xr0 - xr1 - xl0 + xl1)
+ (bkd1 - bku1) * (xd0 - xd1 - xu0 + xu1)

+ (bird - biru) * (xrd - xd1 - xru + xu1)
+ (bir2 - bir0) * (xr2 - xc2 - xr0 + xc0)
+ (bjrd - bjld) * (xrd - xr1 - xld + xl1)
+ (bjd2 - bjd0) * (xd2 - xc2 - xd0 + xc0)
+ (bkr2 - bkl2) * (xr2 - xr1 - xl2 + xl1)
+ (bkd2 - bku2) * (xd2 - xd1 - xu2 + xu1)
));
```

STENCILS ON GPU

Optimization: read-only loads

MACROS FOR GRIDPOINTS AND COEFFICIENTS

POISSON AND HELMHOLTZ WITH 7-POINT STENCIL

```
// macros
#ifdef USE_TEX
  #define  X(i)   ( __ldg(&x[i]) )
  ...
#else
  #define  X(i)   ( x[i] )
  ...
#endif

// operator
#define apply_op_ijk() (                                  \
H0 - b*h2inv*(                                            \
+ BI(ijk+1      )*( X(ijk+1      ) - X(ijk) ) \
+ BI(ijk        )*( X(ijk-1      ) - X(ijk) ) \
+ BJ(ijk+jStride)*( X(ijk+jStride) - X(ijk) ) \
+ BJ(ijk        )*( X(ijk-jStride) - X(ijk) ) \
+ BK(ijk+kStride)*( X(ijk+kStride) - X(ijk) ) \
+ BK(ijk        )*( X(ijk-kStride) - X(ijk) ))\
)
```

LESS INTRUSIVE THAN SMEM, BUT BETTER PERF

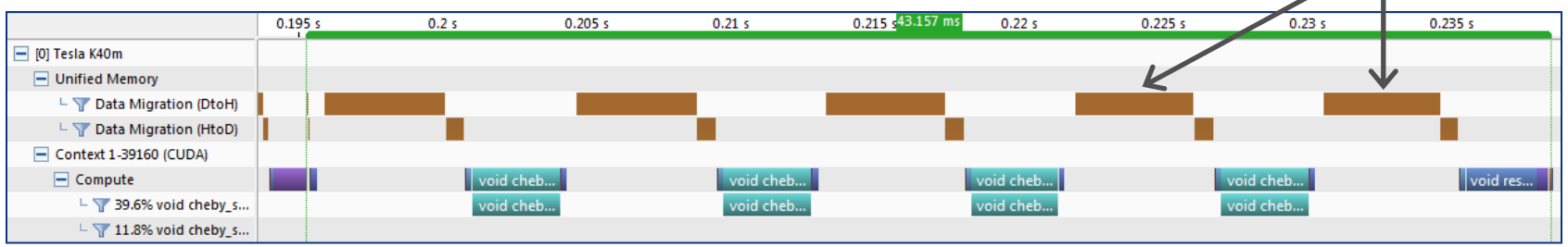
up to 1.3x speed-up!

MULTI-GPU

CUDA-aware MPI

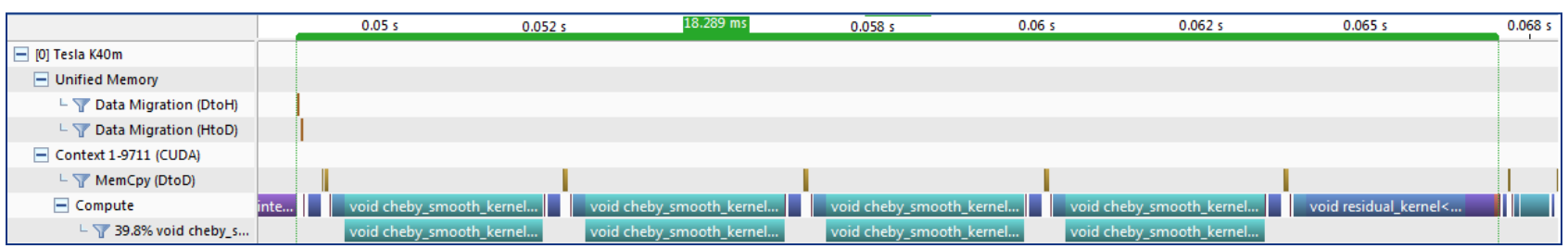
DATA MIGRATIONS (50%)

MPI



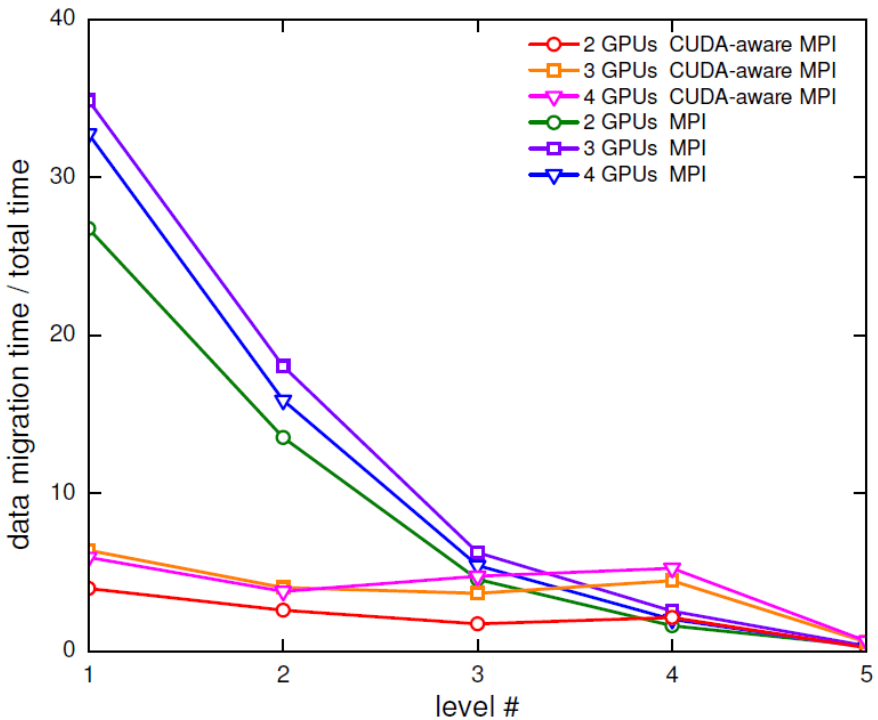
SINGLE NODE WITH P2P BETWEEN TWO K40s

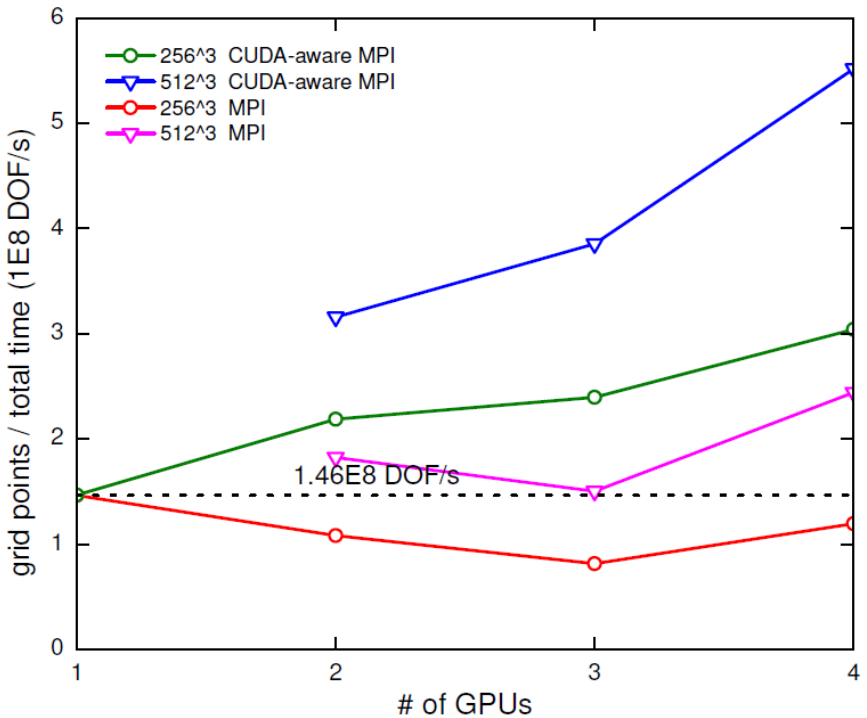
CUDA-AWARE MPI



MULTI-GPU SCALING

Single node with P2P

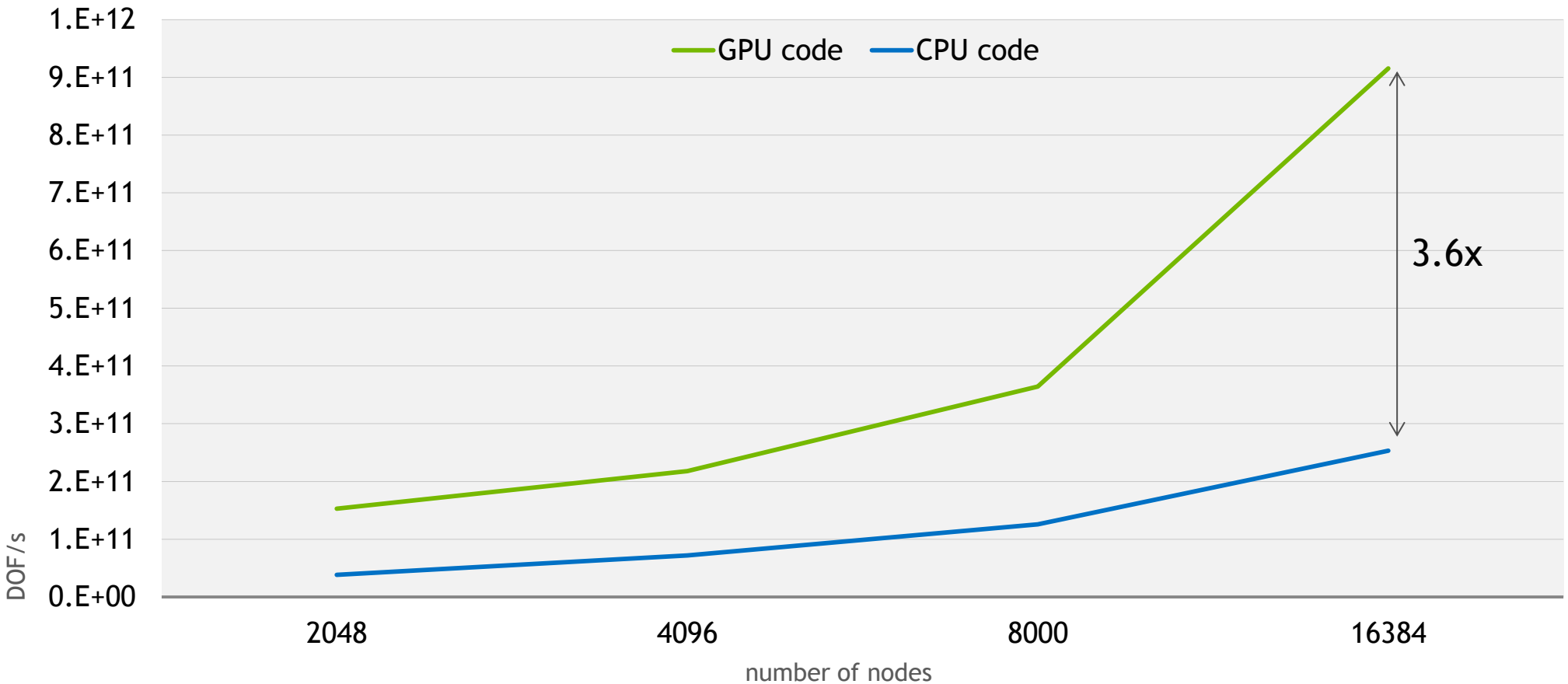




GPU: NVIDIA Tesla K40, ECC on

MULTI-GPU SCALING

ORNL Titan performance



Results acquired by Sam Williams using the latest code : https://bitbucket.org/nsakharnykh/hpgmg-cuda

HPGMG RANKING LIST

ORNL Titan (GPU) – **9.156E+11** DOF/s, 32M DOF per GPU

HPGMG-FV Rank	System	Site	DOF/s	Fraction of System	Parallelization MPI	OMP	DOF per Process	Top500 Rank
1	Mira	Argonne	7.21E+11	100%	49152	64	16M	5
2	K	RIKEN	7.12E+11	73%	64000	8	2M	4
3	Edison	NERSC	3.85E+11	100%	131072	1	4M	18
4	Titan (CPU-only)	Oak Ridge	2.53E+11	88%	32768	8	16M	2
5	Stampede (CPU-only)	TACC	1.49E+11	64%	8192	8	2M	7
6	Hopper	NERSC	1.21E+11	86%	21952	6	2M	34
7	Piz Daint (CPU-only)	CSCS	1.02E+11	78%	4096	8	18M	6
8	SuperMUC	LRZ	7.13E+10	15%	2744	8	16M	12
9	BiFrost	NSC	4.67E+10	98%	1260	16	176M	-
10	Stampede (MIC-only)	TACC	2.16E+10	8%	512	180	16M	7
11	Peregrine (IVB-only)	NREL	1.08E+10	18%	512	12	2M	-
12	Carver	NERSC	1.35E+09	5%	125	4	2M	-
13	Babbage (MIC-only)	NERSC	8.24E+08	30%	27	180	16M	-

https://hpgmg.org/2015/02/06/updated-hpgmg-fv-ranking/

WORK IN PROGRESS

2nd order code is publically available on bitbucket:

https://bitbucket.org/nsakharnykh/hpgmg-cuda

Initial 4th order implementation is ready and will be uploaded soon!

Preliminary results show about 2-3x slower performance compared to 2nd order

Future work:

Exploring finer-grained parallelization for intermediate/coarse levels

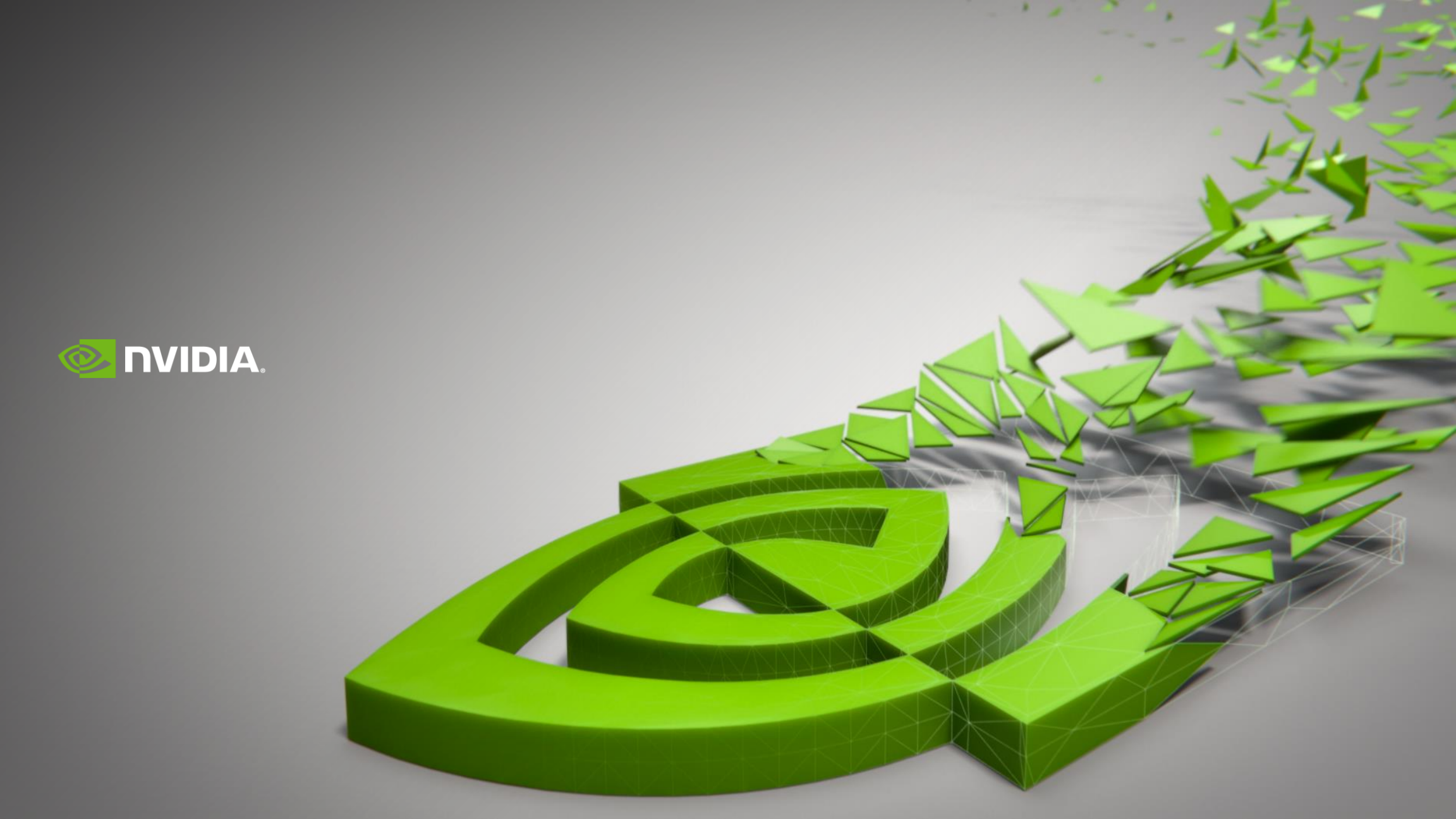
CONCLUSIONS

Both **LOC** and **TOC** architectures are utilized for best performance of HPGMG

Unified Memory greatly simplifies code porting to GPU architecture

Kernel optimizations are not intrusive and should apply to **future architectures**

The code scales well up to **16K GPUs** on large supercomputing clusters


NVIDIA