outthink
limits

Performance Analysis and Optimizations for Lambda-based Applications in OpenMP 4.5

Compiler and Application Teams at IBM
Various People at LLNL

David Truby, Carlo Bertolli, Kevin O’Brien, Kathryn O’Brien
david.truby@ibm.com, {cbertol,caomhin,kmob}@us.ibm.com
IBM T. J. Watson Research Center

Scope of Work

Compiler optimization perspective on (>=) C++11 frameworks

- Lambda-based frameworks make performance portability possible: no other compiler-free known solution
- State-of-Art: plotting performance differences when using C++11 features and OpenMP with various compilers
- **Unclear what compilers actually do**
  - On host and device!

In this presentation

- Using special branch of Clang: https://github.com/clang-ykt
  - ..and Lightweight OpenMP Library
- Experiments on LULESH v2.0 and RAJA
- Reporting performance **then** go figure out why - looking at generated code
- Porting LULESH presents various alternatives
  - Experiment on many different loops to get a full-application view

OpenMP and Lambdas on Host

Capture by copy

```
template <typename LOOP_BODY>
inline void forall_omp(int begin, int end,
                       LOOP_BODY loop_body) {
 #pragma omp parallel for proc_bind(spread)
  for (int ii = 0 ; ii < end ; ++ii ) {
    loop_body( ii );
  }
}

int main() {
 double *a, *b, *c;
 // init a, b, and c

 forall_omp(0, n, [=] (int i) {
     a[i] += b[i] + c[i];
   } );
}
```

Capture all variables undefined in region by copy

```
struct anon {
  double *a, *b, *c;
}

int main() {
  struct anon args;
  args.a = a;
  args.b = b;
  args.c = c;
  fork_call(outlined_region, .., args)
}
```

Captures are retrieved before the loop and re-used within it

```
void outlined_region(.., struct anon args) {
  double *a, *b, *c;
  a = args.a;
  b = args.b;
  c = args.c;
  for (int i = 0 ; i < n ; i++) {..}
}
```

OpenMP and Lambdas on Host

Capture by reference

```
template <typename LOOP_BODY>
inline void forall_omp(int begin, int end,
                                    LOOP_BODY loop_body) {
  #pragma omp parallel for proc_bind(spread)
    for (int ii = 0 ; ii < end ; ++ii ) {
      loop_body( ii );
    }
}

int main() {
  double *a, *b, *c;
  // init a, b, and c

  forall_omp(0, n, [&] (int i) {
      a[i] += b[i] + c[i];
    } );
}
```

Capture all variables undefined in region by reference

```
struct anon {
  double **a, **b, **c;
}

int main() {
  double **a, **b, **c;
  struct anon args;
  args.a = a;
  args.b = b;
  args.c = c;
  fork_call(outlined_region, .., args)
}
```

Captures are now retrieved from within loop body

```
void outlined_region(.., struct anon args) {
  double **a, **b, **c;
  for (int i = 0 ; i < n ; i++) {
    a = args.a;
    b = args.b;
    c = args.c;
    a_val = load a[i];
    b_val = load b[i];
    c_val = load c[i];
    // …
  }
}
```

OpenMP and Lambdas on Device

```
template <typename LOOP_BODY>
inline void forall_omp(int begin, int end,
                                   LOOP_BODY loop_body) {
  #pragma omp target teams distribute \
    parallel for
    for (int ii = 0 ; ii < end ; ++ii )
      loop_body( ii );
}

int main() {
  double *a, *b, *c;
  // init a, b, and c
  #pragma omp target enter data map(to: a[:n], b[:n], c[:n])
  forall_omp(0, n, [=] (int i) {
      a[i] += b[i] + c[i];
    } );
 #pragma omp target exit data map(from: a[:n]) \
    map(release:b[:n], c[:n])
}
```

What the compiler does for you:

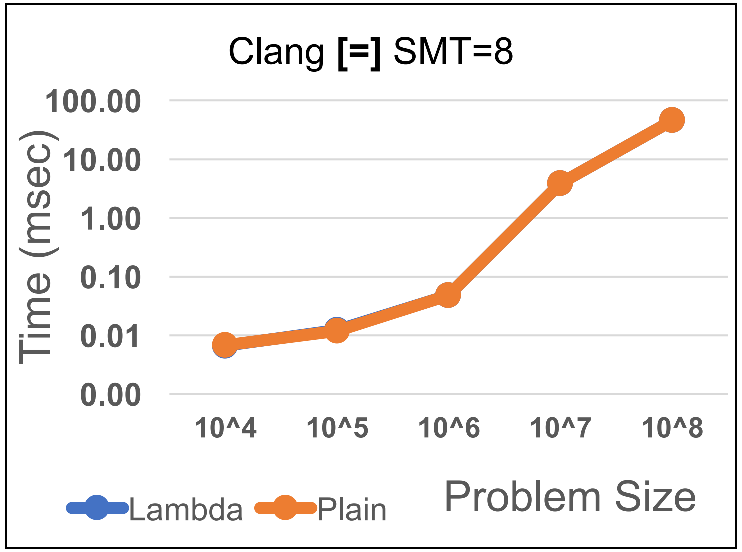
1. Implicit map(tofrom) of lambda struct (can be optimized to map(to))
2. **Instruct the runtime to translate pointers in struct anon from host to device**

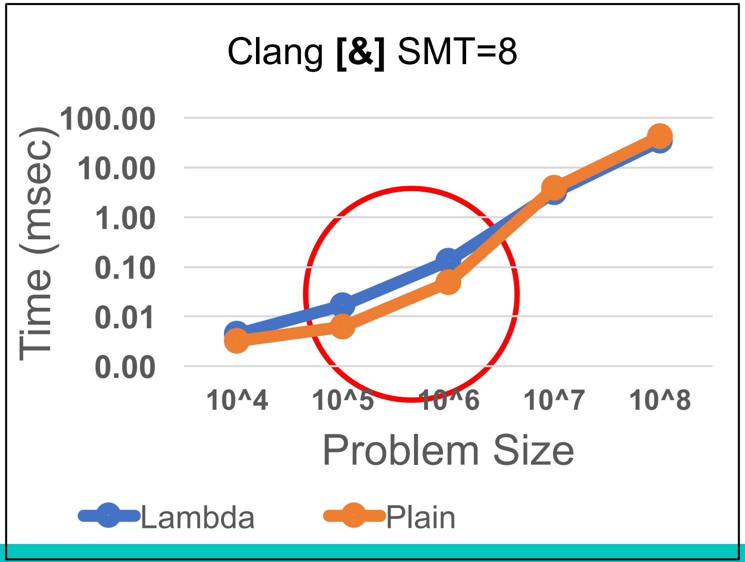
```
struct anon {
  double *a, *b, *c;
}

int main() {
  double *a, *b, *c;
  struct anon args;
  args.a = a;
  args.b = b;
  args.c = c;
  tgt_target_teams(outlined_region, .., args)
}
```

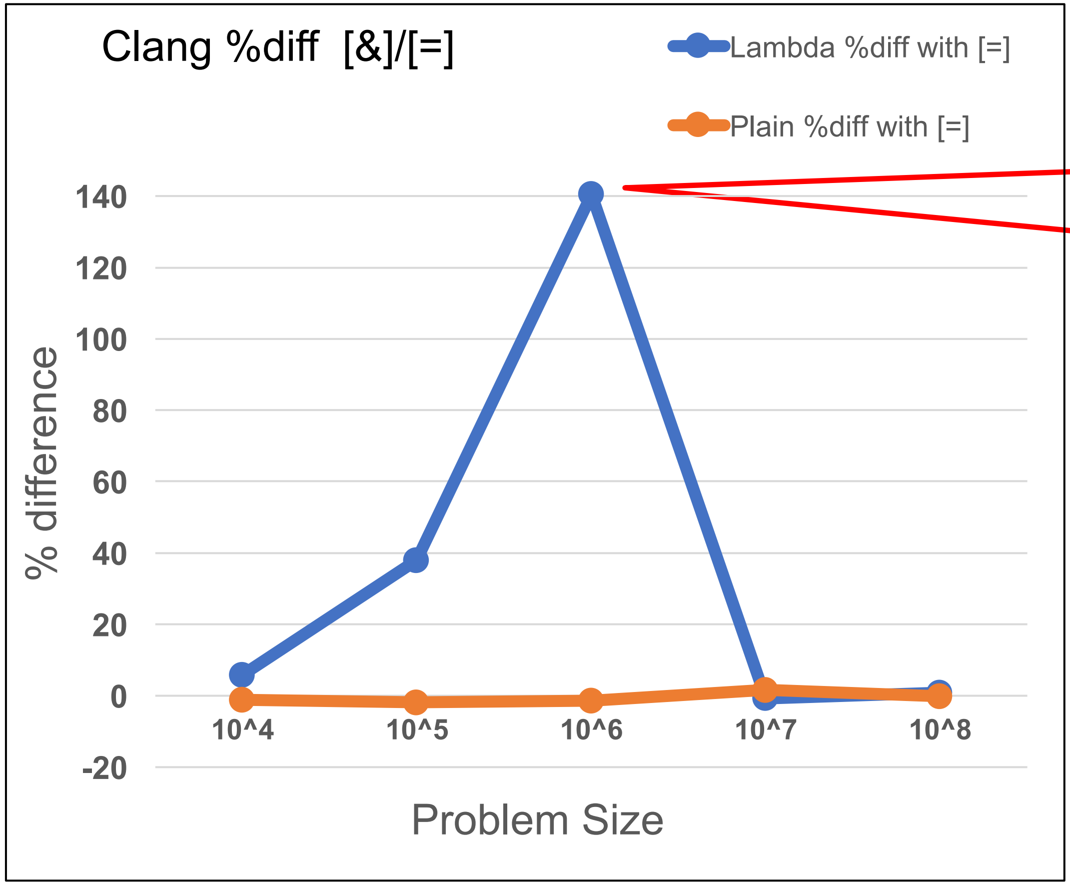
a, b, and c will be translated by runtime

Very Simple Tests – Vector Add

Compare #parallel for with and without lambda, different captures



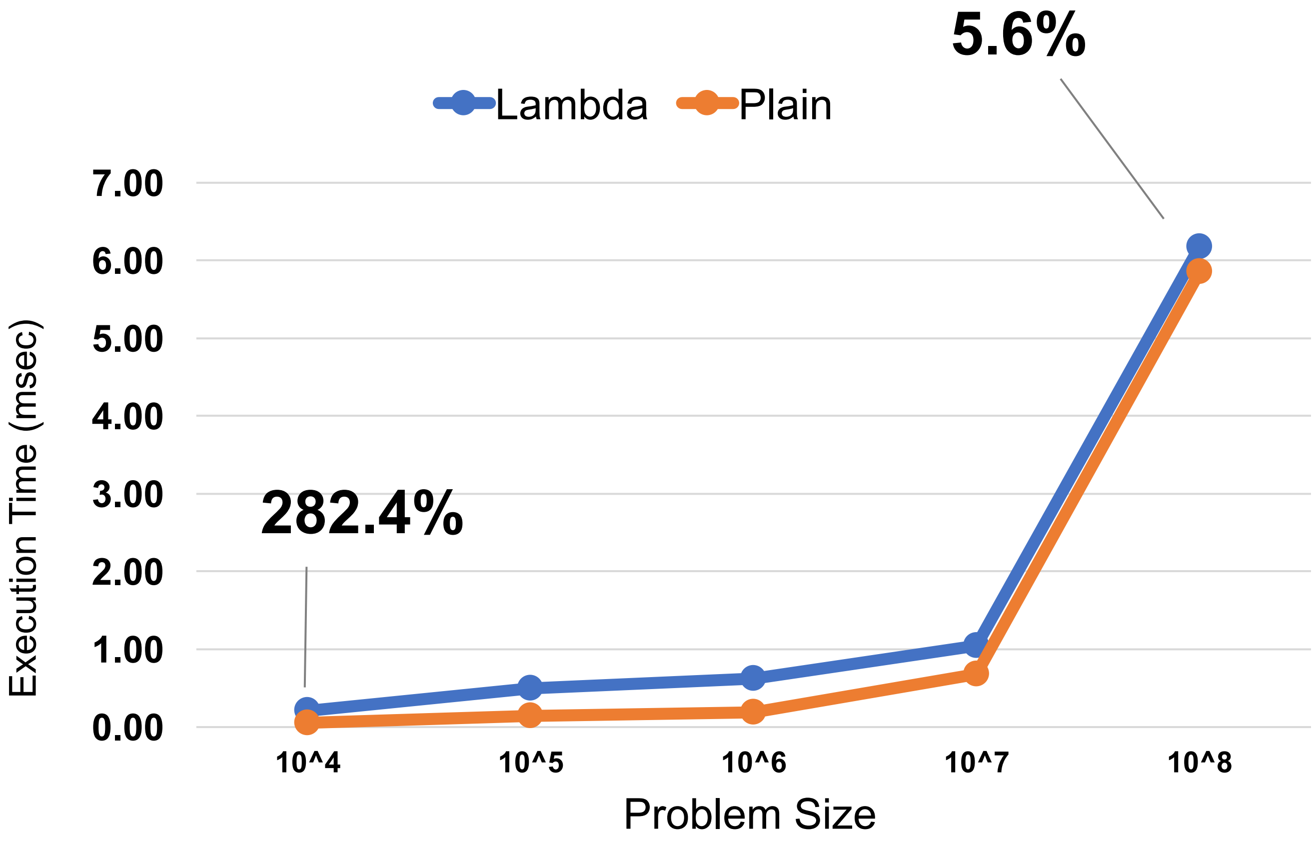




Clang does not vectorize lambda body with [&] capture

remark: loop not vectorized: cannot identify array bounds

Very Simple Test – Vector Add with Target



Difference only at smaller sizes, up to one order of magnitude

Disappears with large iteration space size

Code generated is identical, except lambda version has to retrieve pointers from struct

LULESH 2.0 – Performance Analysis

Partial study of LULESH 2.0 using Raja

- Using RAJA OpenMP 4.5. backend plus our special compiler branch
- Four version of code:
  - Host: plain OpenMP parallel for, RAJA with domain, RAJA with direct array access
  - Device: plain OpenMP target region, RAJA with array capturing

Experiments

- On Power8 S822LC ("Minsky") server, including Pascal GPU (Tesla P100-SXM2-16GB)
- Options and env: -O3, -fopenmp-implicit-declare-target, -ffp-contract=fast, explicitly pinning threads to cores

Kernel	Description	Instructions
CalcLagrangeElements	Elements, small kernel with few operations	4 fadd, 6 fsub, 2 fdiv
CalcSoundSpeedForElems	Variable iteration space, small kernel with switch	1 fadd, 4 fmul, 1 fdiv, 1 sqrt
CalcMonotonicQGradientsForElems	Elements, large kernel without control flow	118 fadd, 27 fsub, 64 fmul, 4 fdiv, 2 sqrt
CalcMonotonicQRegionForElems	Variable iteration space, large kernel with switch	10 fadd, 7 fsub, 35 fmul, 4 fdiv,

LULESH – OpenMP Target Implementation

We modified LULESH to access domain arrays from within capture expression

```
RAJA::forall<elem_exec_policy>(0, numElem,
    [=] (int k) {
      // calc strain rate and apply as constraint
      // (only done in FB element)
      Real_t vdov = domain.dxx(k) + domain.dyy(k)
         + domain.dzz(k) ;
      Real_t vdovthird = vdov/Real_t(3.0) ;
      // make the rate of deformation tensor deviatoric
      domain.vdov(k) = vdov ;
      domain.dxx(k) -= vdovthird ;
      domain.dyy(k) -= vdovthird ;
      domain.dzz(k) -= vdovthird ;
    }
);
```

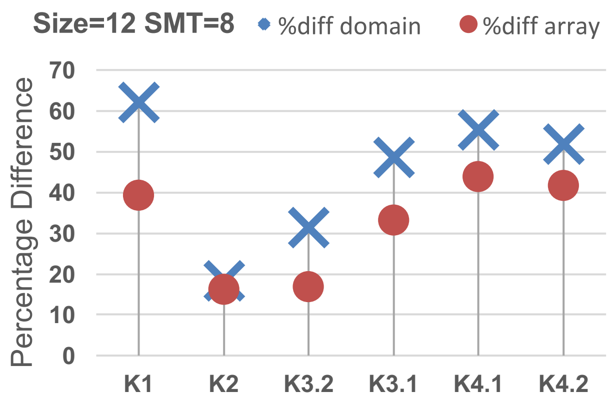
```
RAJA::forall<target_exec_policy>(0, numElem,
    [=, dxx=&domain.dxx(0),
      dyy=&domain.dyy(0),
      dzz=&domain.dzz(0),
      vdov_v=&domain.vdov(0)] (int k) {
    // calc strain rate and apply as constraint
    // (only done in FB element)
    Real_t vdov = dxx[k] + dyy[k] + dzz[k];
    Real_t vdovthird = vdov/Real_t(3.0) ;
    // make the rate of deformation tensor deviatoric
    vdov_v[k] = vdov ;
    dxx[k] -= vdovthird ;
    dyy[k] -= vdovthird ;
    dzz[k] -= vdovthird ;
  }
);
```

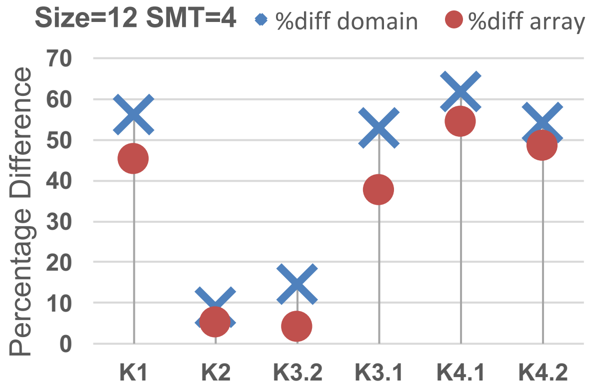
Host Performance - Impact of Lambdas

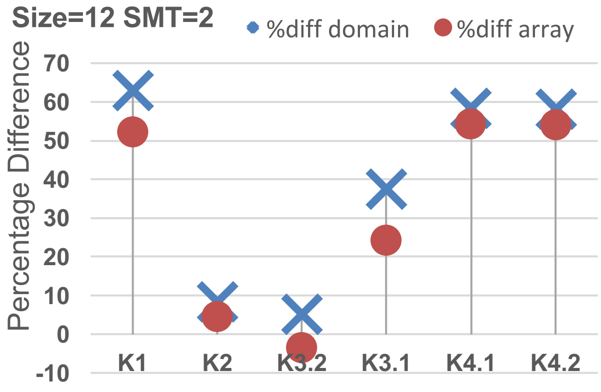
✕ **%difference between RAJA version with domain object and plain version**

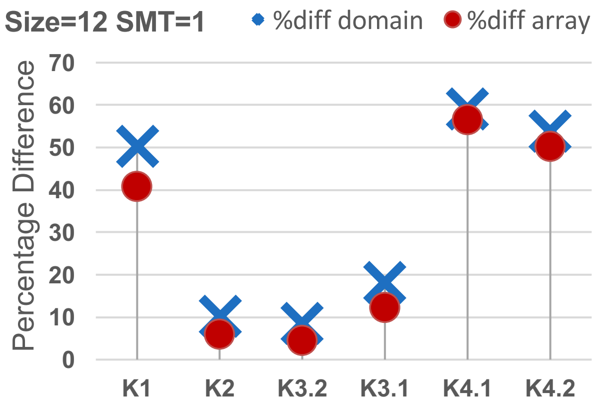
● **%difference between RAJA version using arrays and plain version**

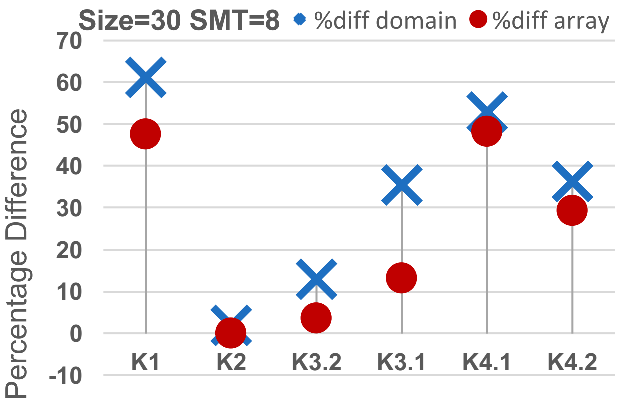
K1 = CalcLagrangeElements
K2 = CalcMonotonicQGradientsForElem
K3.1,3.2 = CalcMonotonicQRegionForElem
K4.1,4.2 = CalcSoundSpeedForElem

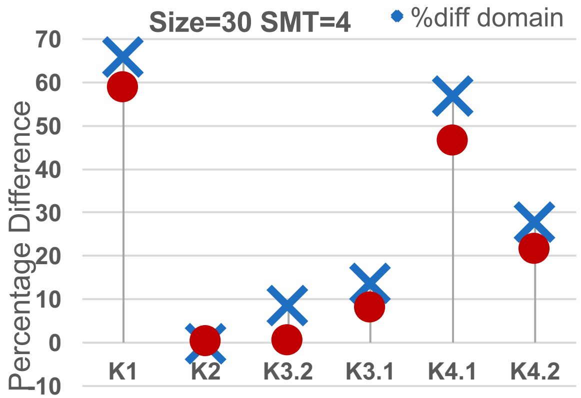

Size=12 SMT=8

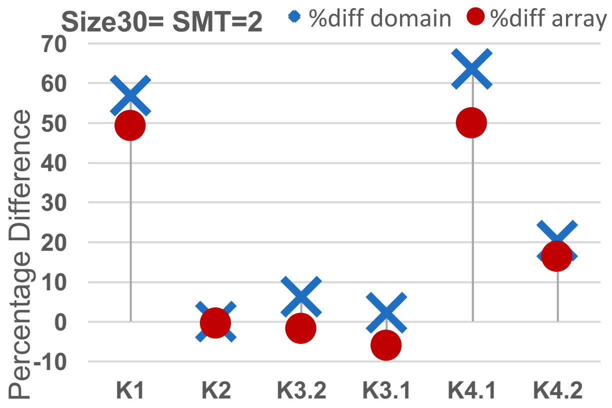

Size=12 SMT=4

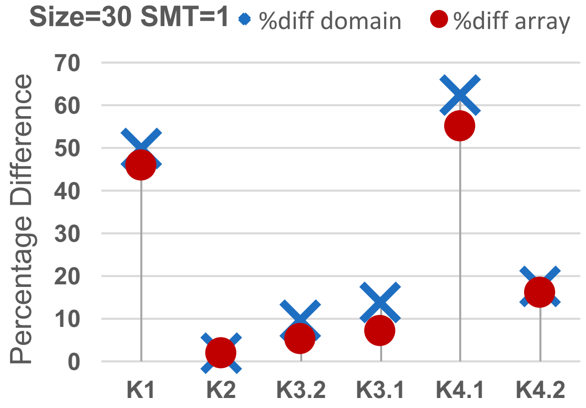

Size=12 SMT=2


Size=12 SMT=1

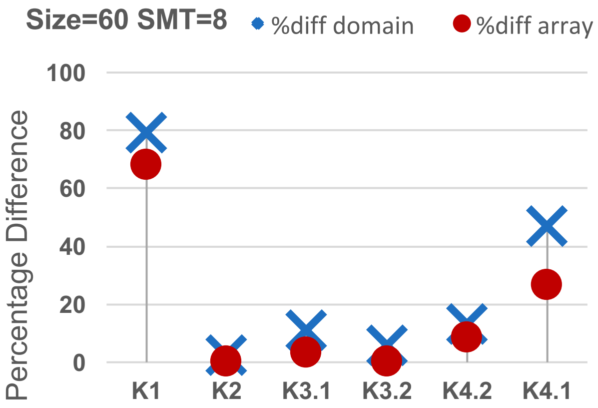

Size=30 SMT=8

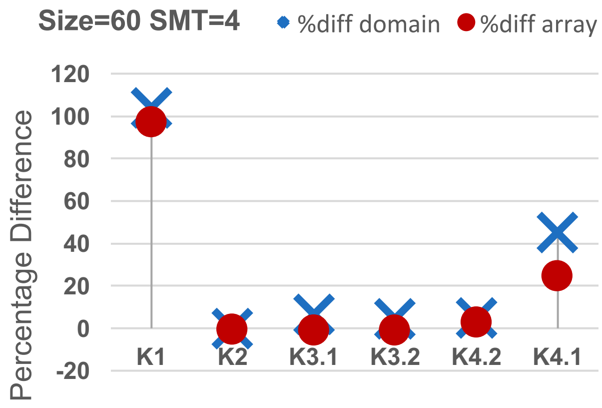

Size=30 SMT=4

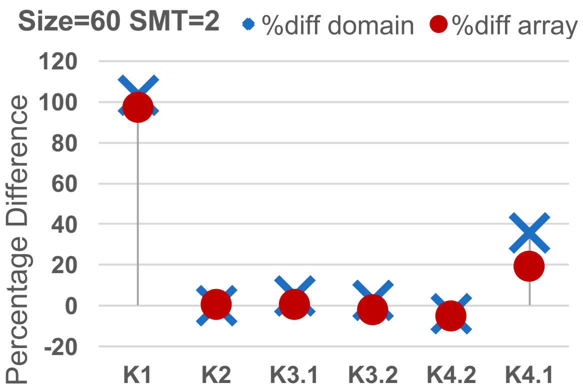

Size30= SMT=2

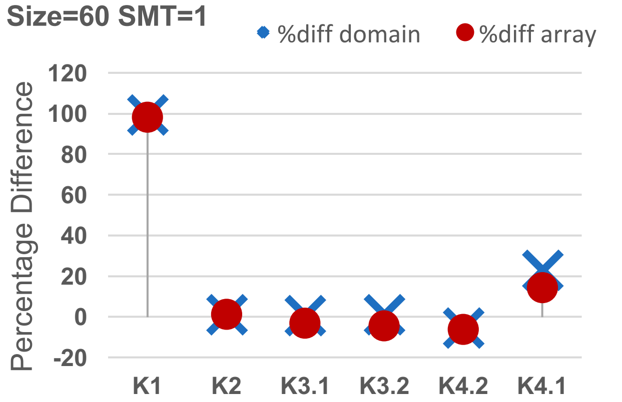

Size=30 SMT=1

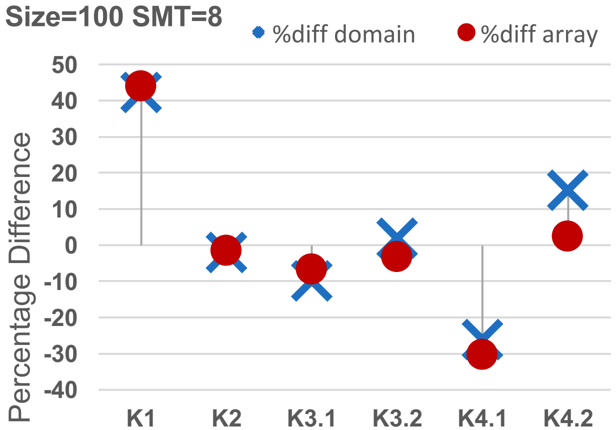
Host Performance - Impact of Lambdas

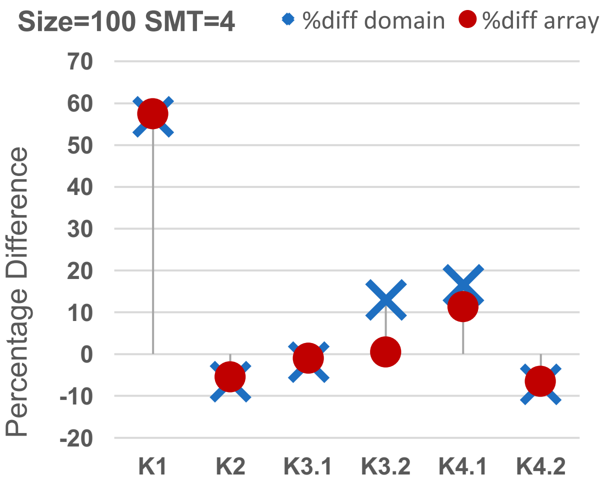


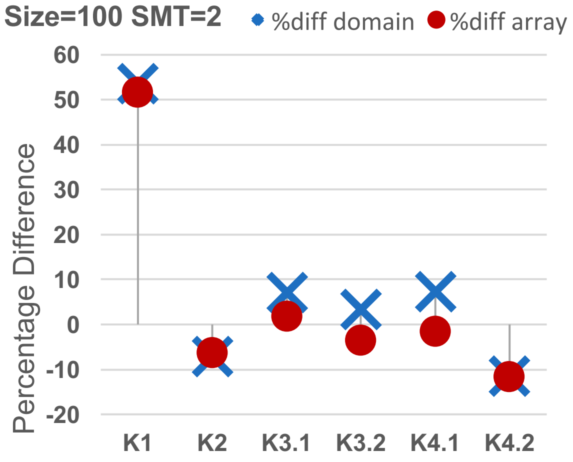


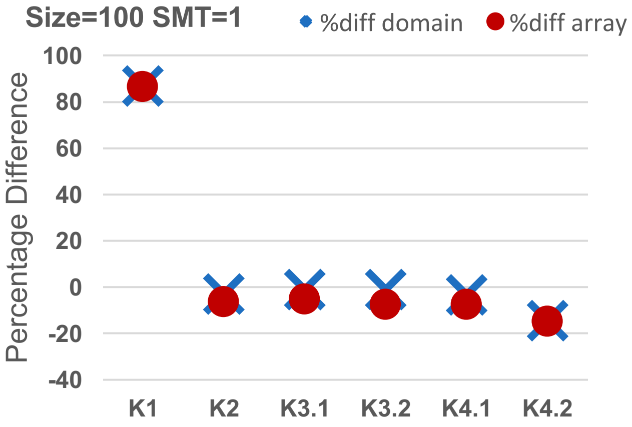












LULESH – Host Results

At small iteration sizes, missing vectorization shows significant slow downs

At large iteration sizes, difference within 10% for most kernels

- In some cases, using lambda results in better performance!
- Missing vectorization becomes irrelevant

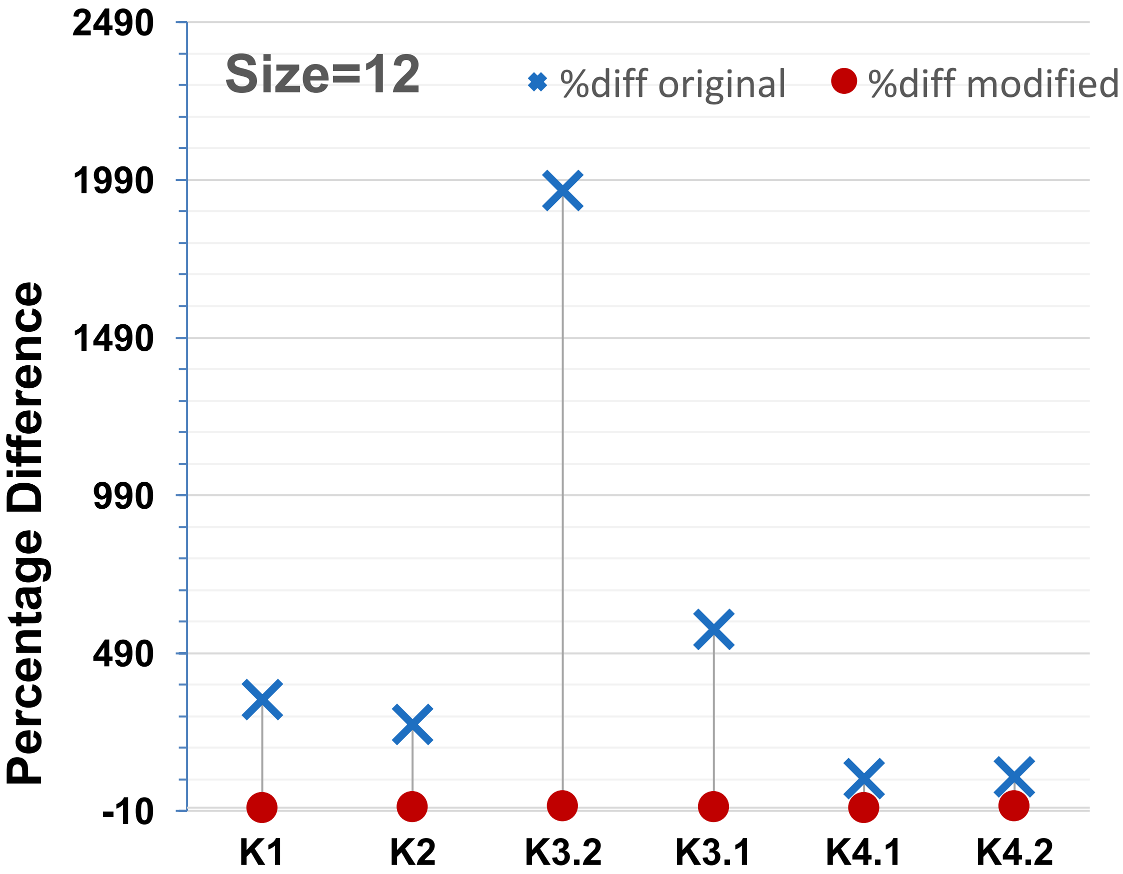
The only kernel that is not performing is CalcLagrangeElements

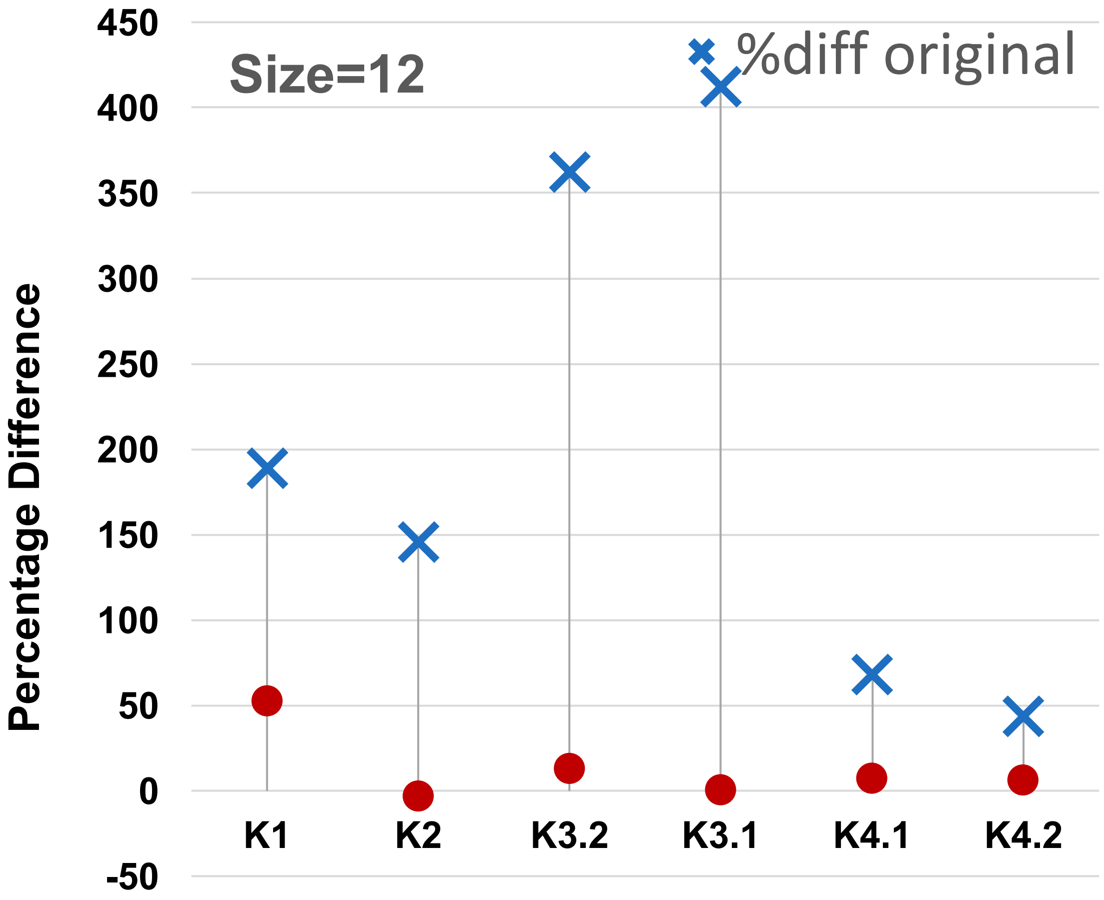
- Very small number of instructions and loads/stores
- Compute-limited: missing vectorization impacts heavily on performance
- Vectorizer report: cannot identify loop bounds
- Likely because of use of **std iteration spaces** to represent loop bounds
- Use/implement a different RAJA parallel for will fix the issue

Improved vectorization in Clang likely to impact multiple architectures

- Comparison with gcc shows that in simple examples it can be done

Lulesh Device Performance Numbers





Lulesh Device Performance Analysis

Latest version of compiler fails at eliding OpenMP runtime because

- Target region with function call to lambda (loop body)

Modified compiler version obtained by forcing runtime elision

- Also improves plain OpenMP target version – analyzing why

Lambda arguments retrieved from within loop body

- Similar to what happened for host vector add when capturing by reference **[&]**

Similar register allocation figures

Kernel	#regs plain target	#regs RAJA target
CalcLagrangeElements	30	32
CalcMonotonicQRegionForElems	64	112
CalcSoundSpeedForElems	32	32
CalcMonotonicQGradientsForElems	254	238

Conclusion

Huge Potential for improvements with some effort

- Improve vectorization capability on host
  - Modify RAJA and/or improve optimizer
- Improve runtime elision detection code for lambda-based target regions
  - Fix compiler, but might require slightly simpler RAJA implementation
- Move lambda argument loads out of loop body on device

Major focus for next few months

Thank you!

ibm.com/systems/hpc

FALLBACK

OpenMP and Lambdas on Device

Limitations

Analysis of LULESH presents challenges

```
class Domain {
  double *m_x, *m_y, *m_z;
};

int main() {
  Domain domain;
  // init m_x, m_y, and m_z in domain object
 #pragma omp target enter data map(to: domain, \
    domain.a[:n], domain.b[:n], domain.c[:n])
  forall_omp(0, n, [=] (int i) {
      domain.a[i] += domain.b[i] + domain.c[i];
    } );
}
```

Capture of Domain object:
The compiler would need to map:

1. domain object – one map entry
2. **all pointers** within domain as any could be used in the target region – one map entry per Domain field

Too many maps per target region: 46 in LULESH 2.0

Capture of
Original LULESH 2.0 version even uses domain pointer: domain→a[i]
Map:

1. Domain pointer
2. Domain pointee – object
3. All pointers within domain object

deep copy?