Developing NEPTUNE for U.S. Naval Weather Prediction

John Michalakes
University Corporation for Atmospheric Research/CPAESS
Boulder, Colorado USA

Dr. Alex Reinecke
U.S. Naval Research Laboratory Marine Meteorology Division

8th NCAR Multicore Workshop, 2018
Acknowledgements

– Martin Berzins, Ouermi Judicael, Brad Peterson: U. Utah
– Sameer Shende, Nick Chaimov: U. Oregon/Paratools
– Christian Trott: Sandia NL (Kokkos)
– Doug Doerfler: Lawrence Berkeley NL (Roofline)
– Vendors: Cavium, Intel, NEC, NVIDIA, Portland Group
NEPTUNE/NUMA

- Navy’s Next Generation Prediction System
- Spectral element dynamics on a cubed sphere
  - Based on NUMA (Frank Giraldo, NPS)
  - Higher-order continuous Galerkin
  - Cubed sphere grid

- Computationally dense but highly scalable
  - Constant width-one halo communication
  - Good locality for next generation HPC

NEPTUNE 72-h forecast (5 km resolution) of accumulated precipitation for Hurr. Sandy

Example of Adaptive Grid tracking a severe event courtesy: Frank Giraldo, NPS

NEPTUNE: Navy Environmental Prediction System Utilizing the NUMA2 core
NUMA: Nonhydrostatic Unified Model of the Atmosphere (Giraldo et. al. 2013)
NEPTUNE/NUMA

- Navy’s Next Generation Prediction System
  - Interoperable physics under NUOPC
  - Data assimilation development under JEDI framework
  - Coupling using ESMF framework
  - Conducting tests with real forecast data
  - Designing, testing and optimizing for next-gen HPC

NEPTUNE Roadmap

- One month of real-data forecasts initialized with GFS analysis fields
- 35-km horizontal grid spacing
Performance and Portability requirements

- **Performance** has lagged badly: good scaling but poor node speed
  - Insufficient fine-grain (vector) utilization
  - Low locality increases mem. latency
  - Excessive data movement lowers C.I.

- **Portability** limited by parallel programming model (MPI/OpenMP/vector) and code structure
  - Intel Xeon (Broadwell, Skylake, Knights Landing)
  - ARM64 (Cavium ThunderX2)
  - NEC VE
  - GPU (Nvidia) (NPS has a NUMA port using OCCA†)

- Solution likely to require major refactoring
  - Minimize one-time and recurring costs
  - Maximize performance benefit over time and range of architectures

**Crucial:** performance analysis and testing starting with kernels


‡https://www.weather.gov/media/sti/nggps/AVEC%20Level%201%20Benchmarking%20Report%2008%2020150602.pdf
Diffusion kernel: create_laplacian

Purpose: Damp energy that cascades to frequencies higher than model can resolve

- Local laplacian computed and applied on each 3D element in **CGD layout**
  + Computationally **dense**, **element-local**, **thread safe**
- Global solution computed on **CGC layout** using Direct Stiffness Summation (DSS) on points shared by neighboring elements
  - Copying from CGC to CGD to accumulate face values requires transposition and non-unit strides that trash **data locality**
  - Potential data races impede **thread parallelism**
- Hot spot routine in **NEPTUNE**
  - Original implementation only stored CGC layout and copied into and out of local CGC arrays for every subroutine in dycore
  - **Initial optimization**: Pick a layout and stick with it
Diffusion kernel: create_laplacian

Purpose:
- Damp energy that cascades to frequencies higher than model can resolve

- Local laplacian computed and applied on each 3D element in CGD layout
  - Computationally dense, element-local, thread safe

- Global solution computed on CGC layout using Direct Stiffness Summation (DSS) on points shared by neighboring elements
  - Copying face values requires transposition and non-unit strides that trash data locality
  - Potential data races impede thread parallelism

- Hot spot routine in NEPTUNE
  - Original implementation only stored CGC layout and copied in and out of local CGC arrays for every subroutine in dycore
  - Key optimization: Pick a layout and stick with it

Original NEPTUNE diffusion code
- Stores data as CGC points
- Copies in/out to CGD local arrays

Optimized Versions
- Refactored to Store Data as Elements (CGD)
- Only copy in and out to CGC for DSS

<table>
<thead>
<tr>
<th>Platform</th>
<th>Architecture</th>
<th>Frequency</th>
<th>Cores</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orig. Gold 6140 2.3 GHz 36c/72t</td>
<td>PX Gold 6140 2.3 GHz 36c/72t</td>
<td>Kokkos V100 (atomic)</td>
<td>Kokkos Gold 6140 2.3 GHz 36c/72t</td>
<td>EPX OpenACC V100 (atomic)</td>
</tr>
<tr>
<td>Time in Seconds (lower is better)</td>
<td>□ dss</td>
<td>□ create_laplacian</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What can we control? Data layout and loops

Element Loop Nesting

- **Outer/Outer**
  - Cache-local elements, good locality
  - Vector dimension is limited to short, often non unit-stride accesses

- **Inner/Outer**
  - Fine-grain dimension is unit-stride, dependency-free, and arbitrary length
  - Having fine-grain innermost requires array temporaries (cache, mem. pressure)

- **Outer/Inner**
  - Fine-grain dimension is unit-stride, dependency-free, and arbitrary length
  - Coalesced accesses to memory by successive GPU threads

Memory Layout

- Inner/Inner
- Inner/Outer
- Outer/Inner
- Outer/Outer

**element-inner** arrays

- dimension(ne,np,nv)

**element-outer** arrays

- dimension(np,nv,ne)

**element-inner loops**

- do v ← 1, nv
- do p ← 1, np
- do e ← 1, ne

**element-outer loops**

- do e ← 1, ne
- do v ← 1, nv
- do p ← 1, np
PX Optimization (element-outer)

- NEPTUNE Prototype
- Ported to
  - Xeon
  - ARM
  - NEC VE

**Memory Layout**

- **Inner/Outer**
  - Inner (null)

- **Outer/Outer**
  - Cache-local elements, good locality
  - Vector dimension is limited to short, often non unit-stride accesses

- **Inner/Inner**
  - Fine-grain dimension is unit-stride, dependency-free, and arbitrary length
  - Having fine-grain innermost requires array temporaries (cache, mem. pressure)

- **Outer/Inner**
  - Fine-grain dimension is unit-stride, dependency-free, and arbitrary length
  - Coalesced accesses to memory by successive GPU threads

**Element Loop Nesting**

```
do e ← 1, ne
  do v ← 1, nv
    do p ← 1, np
```

**element-outer loops**

**element-outer arrays**

dimension(np, nv, ne)
PX Optimization (element-outer)

- NEPTUNE Prototype
- Ported to:
  - Xeon
  - ARM
  - NEC VE

Element Loop Nesting
- Original version of hot-spot diffusion kernel on Skylake
- "PX" optimized version of hot-spot diffusion kernel on Skylake

<table>
<thead>
<tr>
<th>Platform</th>
<th>Time in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orig. Gold 6140 2.3 GHz 36c/72t</td>
<td>0.364</td>
</tr>
<tr>
<td>PX Gold 6140 2.3 GHz 36c/72t</td>
<td>0.079</td>
</tr>
<tr>
<td>Kokkos V100 (atomic)</td>
<td>0.175</td>
</tr>
<tr>
<td>Kokkos Gold 6140 2.3 GHz 36c/72t (atomic)</td>
<td>0.180</td>
</tr>
<tr>
<td>EPX OpenACC V100</td>
<td>0.200</td>
</tr>
<tr>
<td>EPX OpenMP Gold 6140 2.3 GHz 36c/72t</td>
<td>0.205</td>
</tr>
<tr>
<td>EPX OpenMP THX2</td>
<td>0.210</td>
</tr>
</tbody>
</table>
EPX (element-inner) Optimization

Element Loop Nesting

- **Element-ininner loops**
  - do v ← 1,nv
  - do p ← 1,np
  - do e ← 1,ne

- **Element-outer loops**
  - do e ← 1,ne
  - do p ← 1,np

Memory Layout

- **Inner/Outer**
  - (null)
  - Outer/Outer
    - Cache-local elements, good locality
    - Vector dimension is limited to short, often non unit-stride accesses

- **Inner/Inner**
  - Fine-grain dimension is unit-stride, dependency-free, and arbitrary length
  - Having fine-grain innermost requires array temporaries (cache, mem. pressure)

- **Outer/Inner**
  - Fine-grain dimension is unit-stride, dependency-free, and arbitrary length
  - Coalesced accesses to memory by successive GPU threads

- **Outer/Outer**
  - Cache-local elements, good locality
  - Vector dimension is limited to short, often non unit-stride accesses

- **Element-ininner arrays**
  - dimension(ne,np,nv)

**Human readable text**

- Xeon, ARM & NEC VE
  - OpenMP with vectorization
- Nvidia: OpenACC
  - (Thanks Dave Norton, PGI)
EPX (element-inner) Optimization – GPU

Element Loop Nesting

- Xeon, ARM & NEC VE
  - OpenMP with vectorization
- Nvidia: OpenACC
  (Thanks Dave Norton, PGI)

```fortran
do ib = 1, neblk
  nr = LEBLK
  !$acc loop
  do ie = 1, nr, EVEC
    call create_laplacian_ep3(ib, ie, min(ie+EVEC-1,nr))
  end do
end do
```

```
subroutine create_laplacian_EPX(ib, es, ee, nvar &
  ...)

  ! KSI, ETA, ZETA Derivatives
  qq_e(es:ee,m) = qq_e(es:ee,m) + q_visc(es:ee,ipe,)
  qq_n(es:ee,m) = qq_n(es:ee,m) + q_visc(es:ee,ipn,)
  qq_c(es:ee,m) = qq_c(es:ee,m) + q_visc(es:ee,ipc,)

  ee = es = +1 previous thread’s index, GPU memory accesses coalesced
```

```
mem. pressure)
```
EPX (element-inner) Optimization – CPU

**Element Loop Nesting**

- Xeon, ARM & NEC VE
  - OpenMP with vectorization
- Nvidia: OpenACC
  - (Thanks Dave Norton, PGI)

**do ib = 1, neblk**

nrun = LEBLK

*(adjust nrun for partial blocks here)*

!$acc loop

do ie = 1, nrun, EVEC

call create_laplacian_ep3( ib, ie, min(ie+EVEC-1,nrun))

**Dimension**

`dimension(ne,np,nv)`

**subroutine create_laplacian_EPX( ib, es, ee, nvar & ...**

! KSI, ETA, ZETA Derivatives

qq_e(es:ee,m) = qq_e(es:ee,m) + q_visc(es:ee,ipe,r
qq_n(es:ee,m) = qq_n(es:ee,m) + q_visc(es:ee,ipn,r
qq_c(es:ee,m) = qq_c(es:ee,m) + q_visc(es:ee,ipc,r

es = 1, ee = LEBLK – each statement vectorizes

**Element Loop Nesting**

- **element-inner loops**
  - do v ← 1,nv
  - do p ← 1,np
  - do e ← 1,ne

- **element-outer loops**
  - do e ← 1,ne
  - do v ← 1,nv
  - do p ← 1,np
EPX (element-inner) Optimization

- Xeon, ARM, & NEC VE
  - OpenMP with vectorization
- Nvidia: OpenACC
  (Thanks Dave Norton, PGI)

“EPX” optimized on V100, Skylake and THX2

<table>
<thead>
<tr>
<th></th>
<th>dss</th>
<th>create_laplacian</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>0.364</td>
<td></td>
</tr>
</tbody>
</table>

Time in Seconds (lower is better)

- Orig. Gold 6140 2.3 GHz 36c/72t: 0.060
- PX Gold 6140 2.3 GHz 36c/72t: 0.074
- Kokkos V100 (atomic): 0.106
- EPX OpenACC V100 (atomic): 0.060
- EPX OpenMP Gold 6140 2.3 GHz 36c/72t: 0.074
- EPX OpenMP THX2: 0.106
Kokkos Implementation

Kokkos::LayoutRight

Element-outer arrays
dimension(np,nv,ne)

Element-inner arrays
dimension(ne,np,nv)

Memory Layout

Loop Nesting

Inner/Outer
- Fine-grain dimension is unit-stride, dependency-free, and arbitrary length
- Having fine-grain innermost requires array temporaries (cache, mem. pressure)

Outer/Outer
- Cache-local elements, good locality
- Vector dimension is limited to short, often non unit-stride accesses

Inner/Inner
- Fine-grain dimension is unit-stride, dependency-free, and arbitrary length

Outer/Inner
- Fine-grain dimension is unit-stride, dependency-free, and arbitrary length
- Coalesced accesses to memory by successive GPU threads

- GPU parallelizes outer loop over Gangs then SIMT threads
- CPU parallelizes outer loop over OpenMP threads option to vectorized inner loop using hierarchical parallelism

- do e ← 1,ne
  do v ← 1,nv
  do p ← 1,np

- Template-meta programming lib.
- Single source (C++)
  - Xeon, ARM & NEC VE
  - Nvidia V100
- https://github.com/kokkos
(Thanks: C. Trott, Sandia NL)
## Kokkos Implementation

### Kokkos::LayoutRight

- **unit stride**
- **unit stride**
- **unit stride**
- **unit stride**

**Outer/Outer**

- **Cache-local elements**, good locality
- **Vector dimension is limited to short, often non unit-stride accesses**

**Inner/Inner**

- **Fine-grain dimension is unit-stride, dependency-free, and arbitrary length**
- **Having fine-grain innermost requires array temporaries (cache, mem. pressure)**

**Outer/Inner**

- **Fine-grain dimension is unit-stride, dependency-free, and arbitrary length**
- **Coalesced accesses to memory by successive GPU threads**

### element-outer arrays

- **dimension(np,nv,ne)**
- **unit stride**
- **unit stride**
- **unit stride**

**Kokkos::LayoutRight**

- **GPU parallelizes outer loop over Gangs then SIMT threads**
- **CPU parallelizes outer loop over OpenMP threads option to vectorized inner loop using hierarchical parallelism**

**• Template-meta programming lib.**
**• Single source (C++)**
**• Xeon, ARM & NEC VE**
**• Nvidia V100**
**• https://github.com/kokkos**

(Thanks: C. Trott, Sandia NL)

---

```cpp
// Define Functor Class and Operators
typedef Kokkos::View<double [nelem][nvar][npts]> ViewNvarType;
class CreateLaplacianFunctor {
  ViewNvarType _q, _rhs;
  KOKKOS_INLINE_FUNCTION
  CreateLaplacianFunctor(
    const ViewNvarType q, const ViewNvarType rhs
  ) : _q(q), _rhs(rhs) {};

  KOKKOS_INLINE_FUNCTION
  void operator()(CreateLaplacianTag, const size_t ie) const{ // compute laplacian
    ...
  }

  KOKKOS_INLINE_FUNCTION
  void operator()(CreateGlobalTag, const size_t ie) const{ // DSS
    ...
  }
};

int main ( int argc, char *argv[] ){
  ViewNvarType rhs("rhs"), q("q"); // construct views
  // Executable
  Kokkos::initialize( argc, argv );
  Kokkos::parallel_for(Kokkos::RangePolicy<CreateLaplacianTag>(0,nelem),CreateLaplacian);
  Kokkos::parallel_for(Kokkos::RangePolicy<CreateGlobalTag>(0,nelem),CreateLaplacian);
```

---

//  Define Functor Class and Operators

typedef Kokkos::View<double [nelem][nvar][npts]> ViewNvarType;
class CreateLaplacianFunctor {
  ViewNvarType _q, _rhs;
  KOKKOS_INLINE_FUNCTION
  CreateLaplacianFunctor(
    const ViewNvarType q, const ViewNvarType rhs
  ) : _q(q), _rhs(rhs) {};

  KOKKOS_INLINE_FUNCTION
  void operator()(CreateLaplacianTag, const size_t ie) const{ // compute laplacian
    ...
  }

  KOKKOS_INLINE_FUNCTION
  void operator()(CreateGlobalTag, const size_t ie) const{ // DSS
    ...
  }
};

int main ( int argc, char *argv[] ){
  ViewNvarType rhs("rhs"), q("q"); // construct views
  // Executable
  Kokkos::initialize( argc, argv );
  Kokkos::parallel_for(Kokkos::RangePolicy<CreateLaplacianTag>(0,nelem),CreateLaplacian);
  Kokkos::parallel_for(Kokkos::RangePolicy<CreateGlobalTag>(0,nelem),CreateLaplacian);
```
Performance results

single source portable across GPU and CPU
Performance results

Times for Diffusion Kernel  
(Double Precision, Full Node)

<table>
<thead>
<tr>
<th></th>
<th>Orig</th>
<th>PX</th>
<th>EPX</th>
<th>KOKKOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar (M)</td>
<td>153</td>
<td>383</td>
<td>6</td>
<td>480</td>
</tr>
<tr>
<td>128 (2 word) vector (M)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>256 (4 word) vector (M)</td>
<td>15</td>
<td>39</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>512 (8 word) vector (M)</td>
<td>112</td>
<td>0</td>
<td>70</td>
<td>0</td>
</tr>
<tr>
<td>Total instructions (M)</td>
<td>279</td>
<td>422</td>
<td>76</td>
<td>495</td>
</tr>
<tr>
<td>DP ops (M)</td>
<td>1,104</td>
<td>539</td>
<td>564</td>
<td>539</td>
</tr>
<tr>
<td>% vec</td>
<td>0.75</td>
<td>0.22</td>
<td>0.87</td>
<td>0.08</td>
</tr>
<tr>
<td>L1 misses (M)</td>
<td>9.5</td>
<td>na</td>
<td>19.3</td>
<td>10.1</td>
</tr>
<tr>
<td>L2 misses (M)</td>
<td>7.3</td>
<td>na</td>
<td>3.9</td>
<td>6.8</td>
</tr>
<tr>
<td>L3 misses (M)</td>
<td>5.8</td>
<td>na</td>
<td>3.7</td>
<td>5.1</td>
</tr>
<tr>
<td>r/w MB</td>
<td>414</td>
<td>na</td>
<td>294</td>
<td>399</td>
</tr>
<tr>
<td>Comp. Intens.</td>
<td>2.66</td>
<td>na</td>
<td>1.91</td>
<td>1.35</td>
</tr>
<tr>
<td>20 steps sec 1 thread</td>
<td>3.876</td>
<td>2.378</td>
<td>2.252</td>
<td>1.923</td>
</tr>
<tr>
<td>GFLOPs</td>
<td>14.2</td>
<td>11.3</td>
<td>12.5</td>
<td>14.0</td>
</tr>
<tr>
<td>% peak</td>
<td>0.19</td>
<td>0.15</td>
<td>0.17</td>
<td>0.19</td>
</tr>
<tr>
<td>speedup rel orig</td>
<td>1.0</td>
<td>1.6</td>
<td>1.7</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Effect of Floating Point Precision

Skylake TAU/PAPI Performance Metrics  
(Double Prec., Single Core)
Diffusion Kernel Performance Summary

Competitive Performance over Programming Models and Devices

✅ Kokkos
- GPU: Excellent fine-grained utilization on GPU
  - Good occupancy; 25% to 100%; moderate register pressure
- CPU: Nearly identical performance to GPU
  - Failed to exploit vectorization on CPU (8%) because of strictly element-outer loops that only benefit OpenMP threading (hierarchical parallelism may improve)
  - The C++ compiler makes a difference: use icpc, not g++
- Good environment, user support: https://github.com/kokkos/kokkos/issues

✅ Element-inner (EPX) Fortran
- GPU: Best V100 performance with OpenACC
  - Lower occupancy: 18.8% to 31% occupancy; high register pressure
- CPU: Skylake 20 percent slower than GPU
  - Excellent 85% vector utilization on CPU (both AVX512 and ARM)
  - Large working set and L1 pressure and AVX-512 clock penalty
  - Dramatic 2x benefit from single-precision

- Element-outer (PX, the current whole-code optimized prototype)
  - CPU-only, close to Kokkos CPU performance if vectorization disabled to avoid compiler-generated scatter gathers around non unit-stride loops
Should we refactor NEPTUNE and how?

- Need more information:
  - Additional kernels covering more of NEPTUNE dynamics
  - Whole code prototypes and testing
- Recommendation with costs, benefits and timelines for different options
- Decision by project management and sponsor, then implementation and testing

### Perf./Port.

<table>
<thead>
<tr>
<th>PX</th>
<th>EPX</th>
<th>Kokkos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>Fortran</td>
<td>C++</td>
</tr>
<tr>
<td>CPU only</td>
<td>Fine-grain</td>
<td>Fine-grain</td>
</tr>
<tr>
<td>Limited</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vectorization</td>
<td>CPU: OpenMP/Vector</td>
<td>GPU: native CUDA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU: maybe vector?</td>
</tr>
</tbody>
</table>

### Effort

<table>
<thead>
<tr>
<th>PX</th>
<th>EPX</th>
<th>Kokkos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low: Already have this. Good modularity &amp; object-oriendedness</td>
<td>Moderate: Refactor data structures, loops; Preserve modularity, object-oriendedness</td>
<td>Extreme: C++ rewrite!</td>
</tr>
</tbody>
</table>

### Benefits

<table>
<thead>
<tr>
<th>PX</th>
<th>EPX</th>
<th>Kokkos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good performance on CPU architectures</td>
<td>Good performance on CPU and GPU; Application-level control over performance-critical factors; Positioned for next-gen architectures, but refactoring may be needed in future</td>
<td>Good performance on CPU and GPU; Piggyback DOE’s Kokkos for future architectures; C++ resources and marketplace</td>
</tr>
</tbody>
</table>
Performance results

Times for Diffusion Kernel
(Double Precision, Full Node)

<table>
<thead>
<tr>
<th>SP over DP (Full Node)</th>
<th>Orig</th>
<th>PX</th>
<th>EPX</th>
<th>Kokkos</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.11</td>
<td>1.10</td>
<td>2.09</td>
<td>1.12</td>
<td></td>
</tr>
</tbody>
</table>

Effect of Floating Point Precision

Skylake TAU/PAPI Performance Metrics
(Double Prec., Single Core)

<table>
<thead>
<tr>
<th></th>
<th>Orig</th>
<th>PX</th>
<th>EPX</th>
<th>KOKKOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar (M)</td>
<td>153</td>
<td>383</td>
<td>6</td>
<td>480</td>
</tr>
<tr>
<td>128 (2 word) vector (M)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>256 (4 word) vector (M)</td>
<td>15</td>
<td>39</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>512 (8 word) vector (M)</td>
<td>112</td>
<td>0</td>
<td>70</td>
<td>0</td>
</tr>
<tr>
<td>Total instructions (M)</td>
<td>279</td>
<td>422</td>
<td>76</td>
<td>495</td>
</tr>
<tr>
<td>DP ops (M)</td>
<td>1,104</td>
<td>539</td>
<td>564</td>
<td>539</td>
</tr>
<tr>
<td>% vec</td>
<td>0.75</td>
<td>0.22</td>
<td>0.87</td>
<td>0.08</td>
</tr>
<tr>
<td>L1 misses (M)</td>
<td>9.5</td>
<td>na</td>
<td>19.3</td>
<td>10.1</td>
</tr>
<tr>
<td>L2 misses (M)</td>
<td>7.3</td>
<td>na</td>
<td>3.9</td>
<td>6.8</td>
</tr>
<tr>
<td>L3 misses (M)</td>
<td>5.8</td>
<td>na</td>
<td>3.7</td>
<td>5.1</td>
</tr>
<tr>
<td>r/w MB</td>
<td>414</td>
<td>na</td>
<td>294</td>
<td>399</td>
</tr>
<tr>
<td>Comp. Intens.</td>
<td>2.66</td>
<td>na</td>
<td>1.91</td>
<td>1.35</td>
</tr>
<tr>
<td>20 steps sec 1 thread</td>
<td>3.876</td>
<td>2.378</td>
<td>2.252</td>
<td>1.923</td>
</tr>
<tr>
<td>GFLOPs</td>
<td>14.2</td>
<td>11.3</td>
<td>12.5</td>
<td>14.0</td>
</tr>
<tr>
<td>% peak</td>
<td>0.19</td>
<td>0.15</td>
<td>0.17</td>
<td>0.19</td>
</tr>
<tr>
<td>speedup rel orig</td>
<td>1.0</td>
<td>1.6</td>
<td>1.7</td>
<td>2.0</td>
</tr>
</tbody>
</table>
Kokkos Implementation (Kernels Only)

Kokkos::LayoutRight

// Define Functor Class and Operators
typedef Kokkos::View<double [nelem][nvar][npts]> ViewNvarType;

class CreateLaplacianFunctor {
    ViewNvarType _q, _rhs;

    KOKKOS_INLINE_FUNCTION
    CreateLaplacianFunctor(const ViewNvarType q, const ViewNvarType rhs) :
        _q(q), _rhs(rhs) {} 

    KOKKOS_INLINE_FUNCTION
    void operator()(CreateLaplacianTag, const size_t ie) const {
        // compute laplacian
    }

    KOKKOS_INLINE_FUNCTION
    void operator()(CreateGlobalTag, const size_t ie) const {
        // DSS
    }
}

Kokkos Implementation on V100 and Skylake

(Thanks: C. Trott, Sandia NL)
EPX (element-inner) Optimization

do ib = 1, neblk
    nrun = LEBLK
    (adjust nrun for partial blocks here)
enddo

!$acc loop
    do ie = 1, nrun, EVEC
        call create_laplacian_ep3( ib, ie, min(ie+EVEC-1,nrun)
enddo

subroutine create_laplacian_EPX( ib, es, ee, nvar &
    
    !KSI, ETA, ZETA Derivatives
    qq_e(es:ee,m) = qq_e(es:ee,m) + q_visc(es:ee,ipe,m)*dpsi(l,i)
    qq_n(es:ee,m) = qq_n(es:ee,m) + q_visc(es:ee,ipn,m)*dpsi(l,j)
    qq_c(es:ee,m) = qq_c(es:ee,m) + q_visc(es:ee,ipc,m)*dpsi(l,k)
end subroutine

- Xeon, ARM & NEC VE
  OpenMP with vectorization
- Nvidia: OpenACC
  (Thanks Dave Norton, PGI)