Developing NEPTUNE for U.S. Naval Weather Prediction

John Michalakes
University Corporation for Atmospheric Research/CPAESS
Boulder, Colorado USA
michalak@ucar.edu

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

18th ECMWF Workshop on HPC in Meteorology, 25 Sept. 2018
Acknowledgements

- NEPTUNE developers at NRL/MRY and NPS
  James Doyle, Kevin Viner, Sasa Gabersek, Matus Martini, Dave Ryglicki,
  Dave Flagg, Sarah King, Sergey Frolov, and Francis Giraldo
- Martin Berzins, Mike Kirby, 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

1NEPTUNE: Navy Environmental Prediction sysTem Utilizing the NUMA2corE
2NUMA: 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
  - Excessive data movement lowers C.I.
  - Low locality increases mem. latency

- **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

NOAA Benchmarks, April 2015
Edison Cores Required for Operational Speed Threshold

NEPTUNE (blue) 6.6x slower than FV3 in NOAA benchmarks from 2015‡
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

https://github.com/michalakes/visckernel
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

![Graph showing performance comparison between different configurations](image-url)
What can we control? Data layout and loops

**Element Loop Nesting**

<table>
<thead>
<tr>
<th>Memory Layout</th>
<th>Element Loop Nesting</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inner/Outer</strong></td>
<td>(null)</td>
</tr>
</tbody>
</table>
| **Inner/Inner**        | + 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 |
| **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 \((np,nv,ne)\)

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

**Element-outer loops**
- do \(v \leftarrow 1, nv\)
- do \(p \leftarrow 1, np\)
- do \(e \leftarrow 1, ne\)

**Element-inner loops**
- do \(e \leftarrow 1, ne\)
- do \(v \leftarrow 1, nv\)
- do \(p \leftarrow 1, np\)
### PX Optimization (element-outer)

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

#### Memory Layout

- **Inner/Outer**
  - 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 Loop Nesting

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

- **Original version of hot-spot diffusion kernel on Skylake**
- **“PX” optimized version of hot-spot diffusion kernel on Skylake**

<table>
<thead>
<tr>
<th>Configuration</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></td>
</tr>
<tr>
<td>Kokkos Gold 6140 2.3 GHz 36c/72t</td>
<td></td>
</tr>
<tr>
<td>EPX OpenACC V100 (atomic)</td>
<td></td>
</tr>
<tr>
<td>EPX OpenMP Gold 6140 2.3 GHz 36c/72t</td>
<td></td>
</tr>
<tr>
<td>EPX OpenMP THX2</td>
<td></td>
</tr>
</tbody>
</table>
Overall impact of first optimization pass

Whole code optimization

- **Vectorization of hot-spots**
- OpenMP SMPD threading
- **New Grid-to-Memory Layouts for Memory Locality**
- Asynchronous MPI communication
- Bit-for-bit reproducibility (for debugging)
- Application of TAU from ParaTools under PETTT

---

### Speedup from Optimization

<table>
<thead>
<tr>
<th>System</th>
<th>Original</th>
<th>Optimized 1</th>
<th>Optimized 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haswell (E5-2640 v3, 2x8 cores, &quot;Sandy&quot;)</td>
<td>1.47</td>
<td>1.23</td>
<td>1.30</td>
</tr>
<tr>
<td>Broadwell (E5-2697-v4, 2x18 cores, &quot;Cheyenne&quot;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Skylake (Gold 6148, 2x20 cores, &quot;Endeavor&quot;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Knights Landing (7250, 68 cores, &quot;Mia&quot;)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Higher is better
EPX (element-inner) Optimization

Element Loop Nesting

**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

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

- **Xeon, ARM & NEC VE**
  - OpenMP with vectorization

- **Nvidia: OpenACC**
  - (Thanks Dave Norton, PGI)

---

**Subroutine** `create_laplacian_EPX`:

```fortran
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

---

**Doib = 1, neblk**

```fortran
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)
```

---

**Extent of array in innermost element dimension** (16 on CPU)

**Extent of vectorized-loops (array syntax) in routine**

(same as LEBLK, outer loop executes only once)
EPX (element-inner) Optimization – GPU

**Element Loop Nesting**

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

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

```plaintext
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
```

**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
EPX (element-inner) Optimization

- Xeon, ARM & NEC VE OpenMP
- Nvidia: OpenACC

(Thanks Dave Norton, PGI)

Original version

“EPX” optimized on V100, Skylake and THX2

Time in Seconds (lower is better)

0.364
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00

Orig. Gold 6140 2.3 GHz 36c/72t
PX Gold 6140 2.3 GHz 36c/72t
Kokkos V100 (atomic)
Kokkos Gold 6140 2.3 GHz 36c/72t
EPX OpenACC V100 (atomic)
EPX OpenMP Gold 6140 2.3 GHz 36c/72t
EPX OpenMP THX2

0.060
0.074
0.106
Kokkos Implementation

- **Kokkos::LayoutRight**
  - Element-outer arrays
    - Dimension $(np, nv, ne)$
  - Element-inner arrays
    - Dimension $(ne, np, nv)$

**Memory Layout**

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

- **Inner/Inner**
  - 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

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

**Loop Nesting**

- **Element-outer loops**
  - $do \ e \leftarrow 1, ne$
  - $do \ v \leftarrow 1, nv$
  - $do \ p \leftarrow 1, np$

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

// 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);
}

(Thanks: C. Trott, Sandia NL)
Performance results

- Orig. Gold 6140 2.3 GHz 36c/72t: 0.364
- PX Gold 6140 2.3 GHz 36c/72t: 0.079
- Kokkos V100 (atomic): 0.066
- Kokkos Gold 6140 2.3 GHz 36c/72t: 0.066
- EPX OpenACC V100 (atomic): 0.060
- EPX OpenMP Gold 6140 2.3 GHz 36c/72t: 0.074
- EPX OpenMP THX2: 0.106

Time in Seconds (lower is better)
Performance results

Single source portable across GPU and CPU
### Performance results

#### Table: Performance metrics

<table>
<thead>
<tr>
<th>Metric</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>

#### Diagram: Time in Seconds (lower is better)

- **SP over DP (Full Node):**
  - Orig: 1.11
  - PX: 1.10
  - EPX: 2.09
  - Kokkos: 1.12

---

**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
    • Kokkos fails to exploit vectorization on CPU (8%) because of strictly element-outer loops that only benefit OpenMP threading.
    • Kokkos has mechanism for vectorizing explicitly (not arch. agnostic but fix coming)
  - 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
**Next steps**

- Additional kernels covering NEPTUNE dynamics
- Effects of varying workloads, numerical order
- Evaluate other DSL approaches: GridTools, PSyKAI
- Whole code prototypes and testing
- Recommendation on refactoring with costs, benefits and timelines for different options

<table>
<thead>
<tr>
<th></th>
<th>Perf./Port.</th>
<th>Effort</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>PX</td>
<td>Fortran, CPU only, Limited vectorization</td>
<td>Low: Already have this. Good modularity &amp; object-orientedness</td>
<td>Good performance on CPU architectures</td>
</tr>
<tr>
<td>EPX</td>
<td>Fortran, Fine-grain, Single-source</td>
<td>Moderate: Refactor data structures, loops; Preserve modularity, object-orientedness</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>
</tr>
<tr>
<td>Kokkos</td>
<td>C++, Fine-grain, Single-source, GPU: native CUDA, CPU: maybe vector?</td>
<td>Extreme: C++ rewrite!</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

**SP over DP (Full Node)**

<table>
<thead>
<tr>
<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>
</tr>
</tbody>
</table>

Effect of Floating Point Precision