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

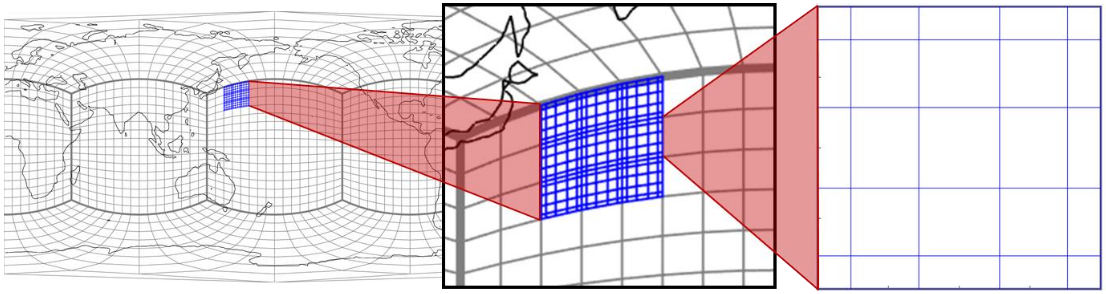
– Christian Trott: Sandia NL (Kokkos)

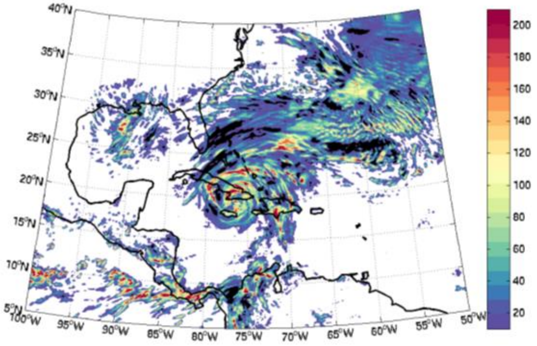
– Doug Doerfler: Lawrence Berkeley NL (Roofline)

– Vendors: Cavium, Intel, NEC, NVIDIA, Portland Group

NEPTUNE/NUMA

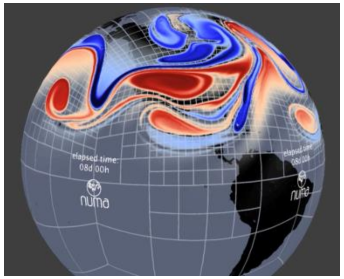
- Navy's Next Generation Prediction System
- Spectral element dynamics on a cubed sphere
  - Based on NUMA (Frank Giraldo, NPS)
  - Higher-order continuous Galerkin
  - Cubed sphere grid





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

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

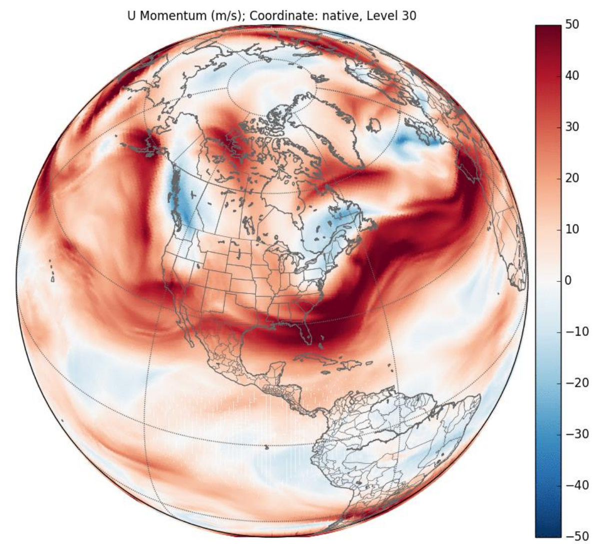


Example of Adaptive Grid tracking a severe event courtesy: Frank Giraldo, NPS

[1]***NEPTUNE:*** ***N****avy* ***E****nvironmental* ***P****rediction sys****T****em* ***U****tilizing the* ***N****UMA*[2] *cor****E***
[2]***NUMA****:* ***N****onhydrostatic* ***U****nified* ***M****odel of the* ***A****tmosphere (Giraldo et. al. 2013)*

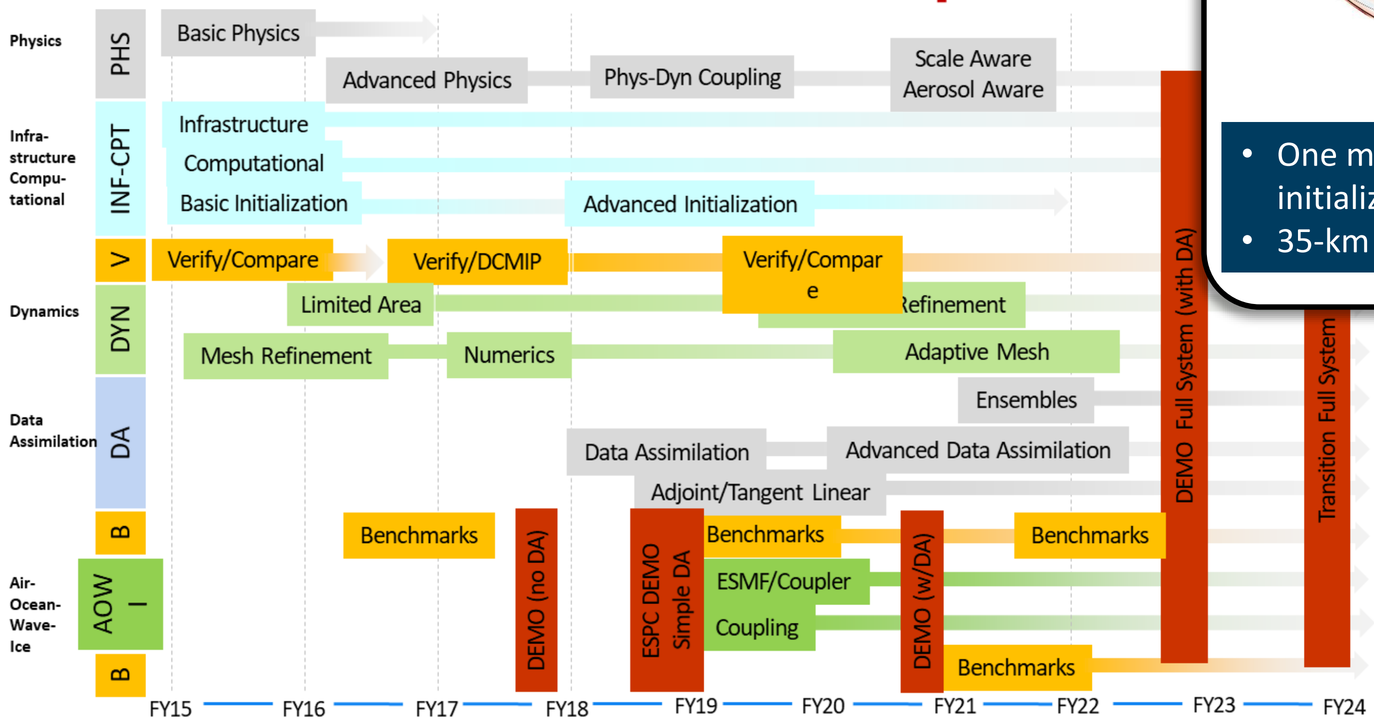
NEPTUNE/NUMA

- 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



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

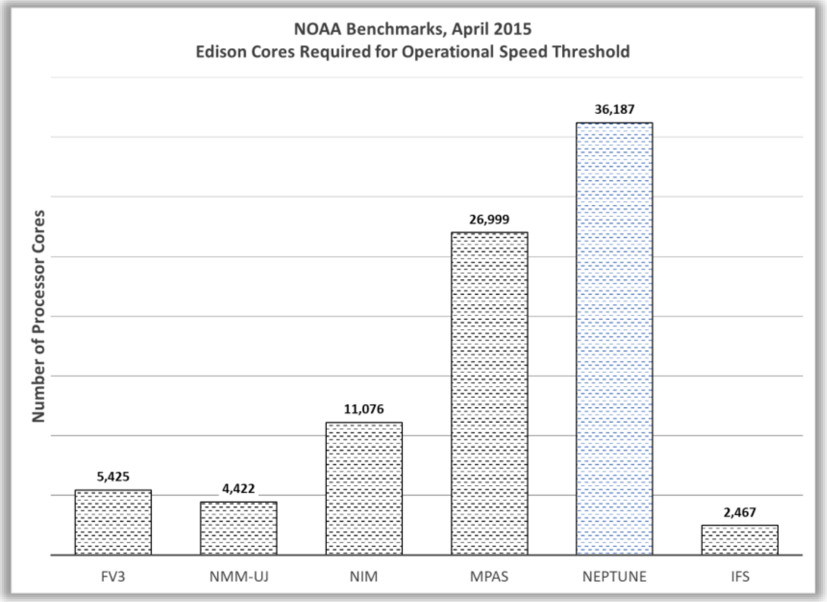
NEPTUNE Roadmap



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



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

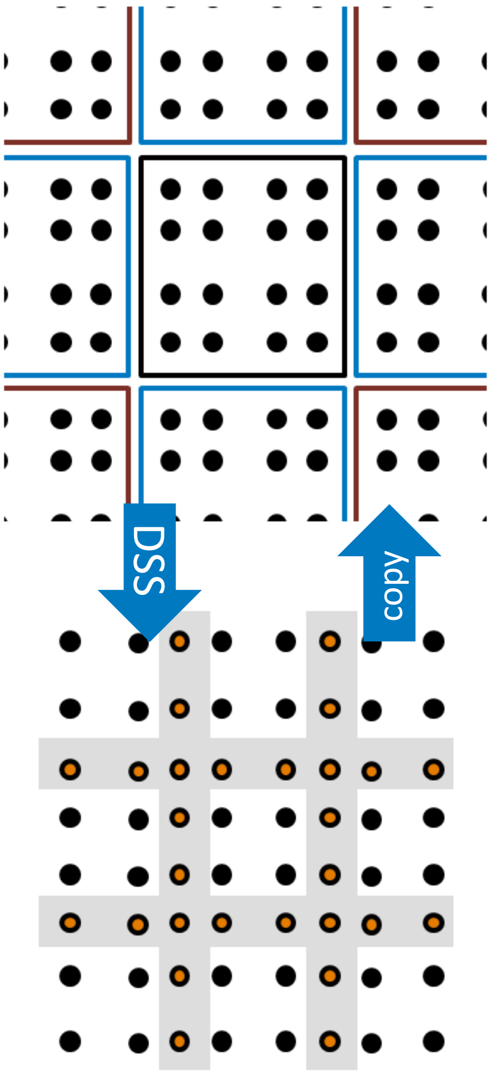
†Abdi, D. S., Wilcox, L. C., Warburton, T. C., & Giraldo, F. X. (2017). A GPU-accelerated continuous and discontinuous Galerkin non-hydrostatic atmospheric model. *The International Journal of High Performance Computing Applications*, 1094342017694427.

‡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**

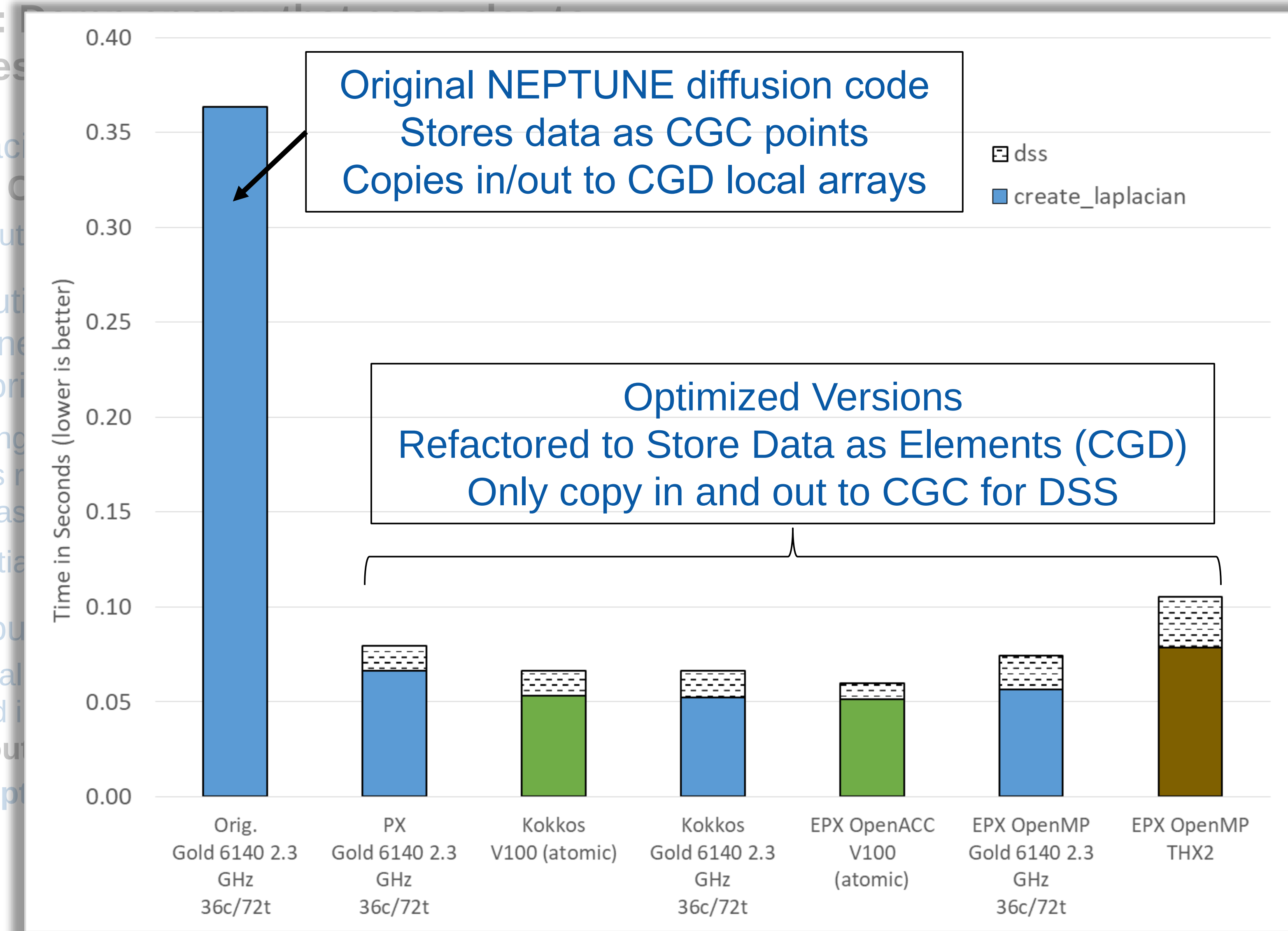


CGD layout: Continuous Galerkin with **Replicated** Points

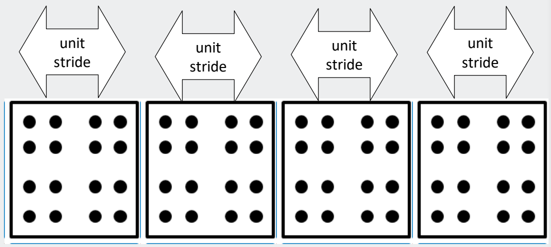
CGC layout: Continuous Galerkin with **Shared** Points

https://github.com/michalakes/visckernel

Diffusion kernel: create_laplacian

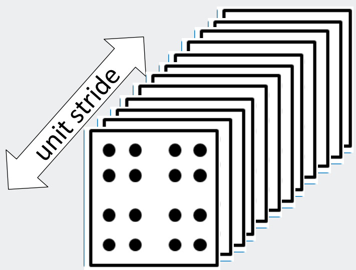


What can we control? Data layout and loops



element-**outer** arrays

dimension(np,nv,ne)



element-**inner** arrays

dimension(ne,np,nv)

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

Element Loop Nesting

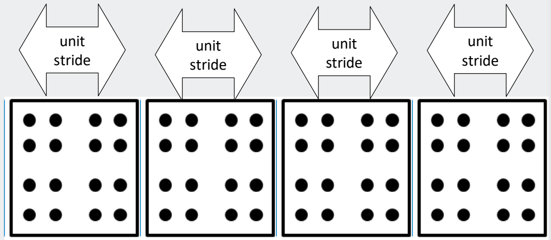
element-**inner** loops

```
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
```

PX Optimization (element-outer)



element-**outer** arrays

dimension(np,nv,ne)

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

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)

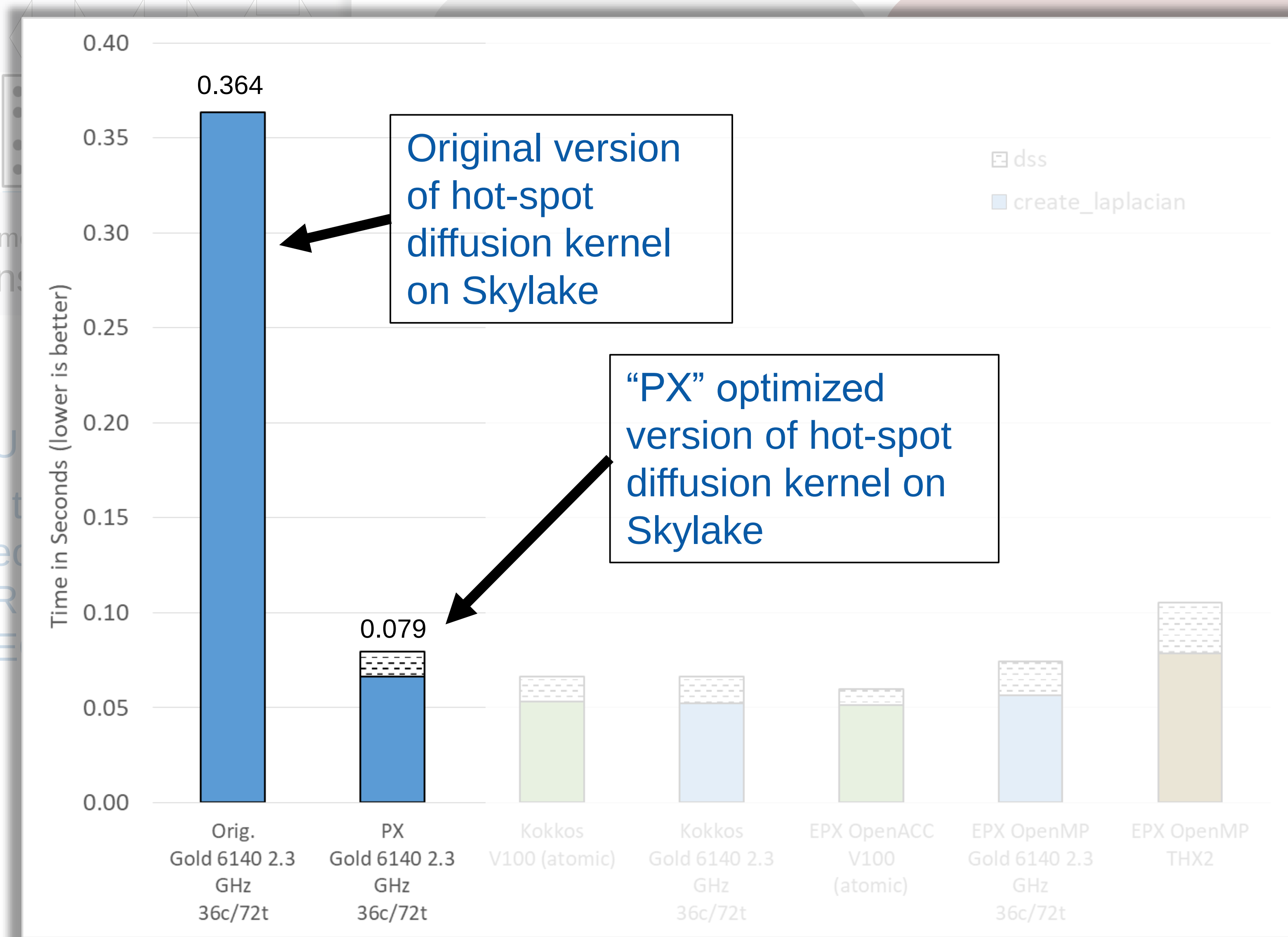
Outer/Inner
+ Fine-grain dimension is unit-stride, dependency-free, and arbitrary length
+ Coalesced accesses to memory by successive GPU threads

Element Loop Nesting

element-**outer** loops

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

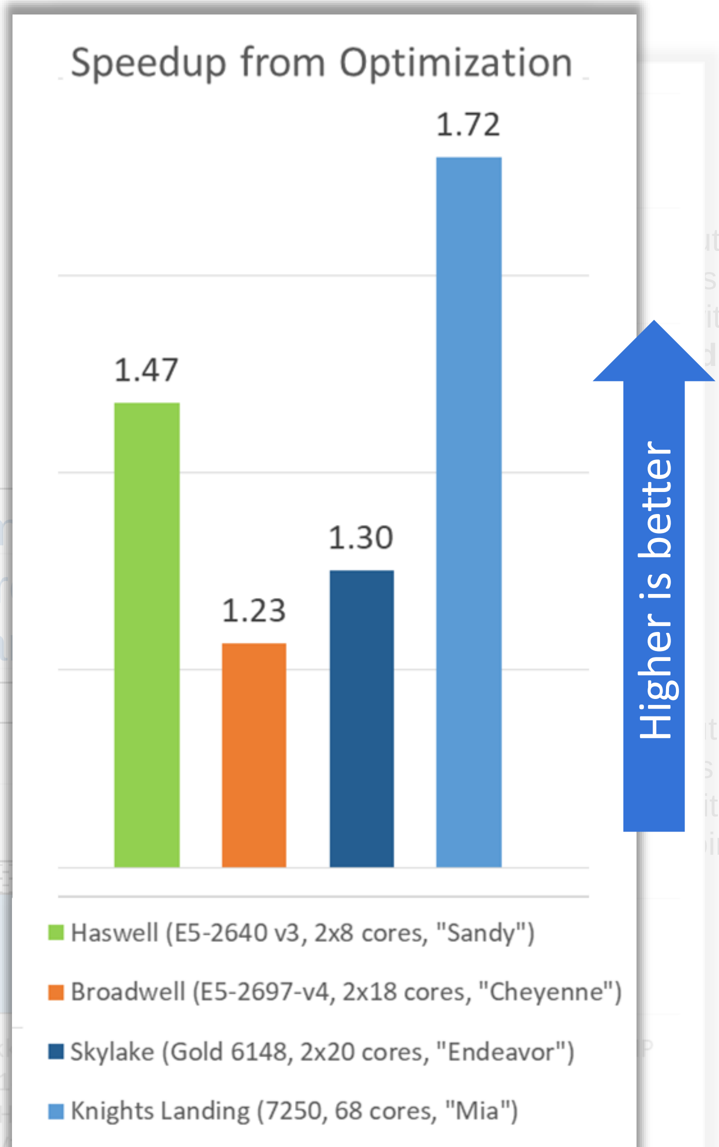
PX Optimization (element-outer)



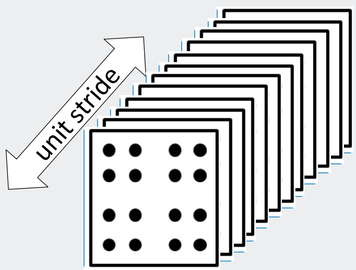
Overall impact of first optimization pass

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



EPX (element-inner) Optimization



element-**inner** arrays

dimension(ne,np,nv)

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

Element Loop Nesting

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

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 – CPU

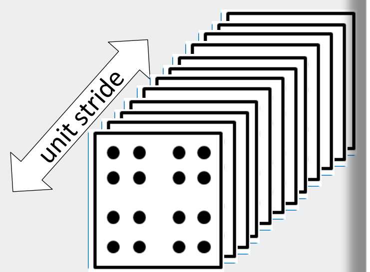
```
do ib = 1,neblk
  nrun = LEBLK
   (adjust nrun for partial blocks here)
!$acc loop
  do ie = 1, nrun, EVEC
    call create_laplacian_ep3( ib, ie, min(ie+EVEC-1,nrun)
```

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)

```
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
   qq_n(es:ee,m) = qq_n(es:ee,m) + q_visc(es:ee,ipn,m
   qq_c(es:ee,m) = qq_c(es:ee,m) + q_visc(es:ee,ipc,m
```

es = 1, ee = LEBLK – each statement vectorizes



element-**inner** arrays

dimension(ne,np,nv)

mem. pressure)

Element Loop Nesting

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

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 – GPU

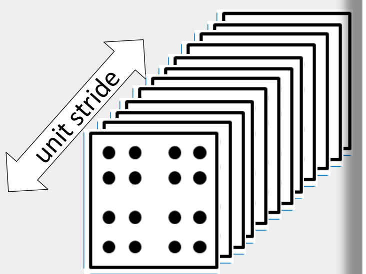
```
do ib = 1,neblk
  nrun = LEBLK
   (adjust nrun for partial blocks here)
!$acc loop
  do ie = 1, nrun, EVEC
    call create_laplacian_ep3( ib, ie, min(ie+EVEC-1,nrun)
```

extent of array in innermost element dimension (32 on GPU)

(Set to 1, so outer loop partitioned over 32 threads)

```
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
```

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



element-**inner** arrays

dimension(ne,np,nv)

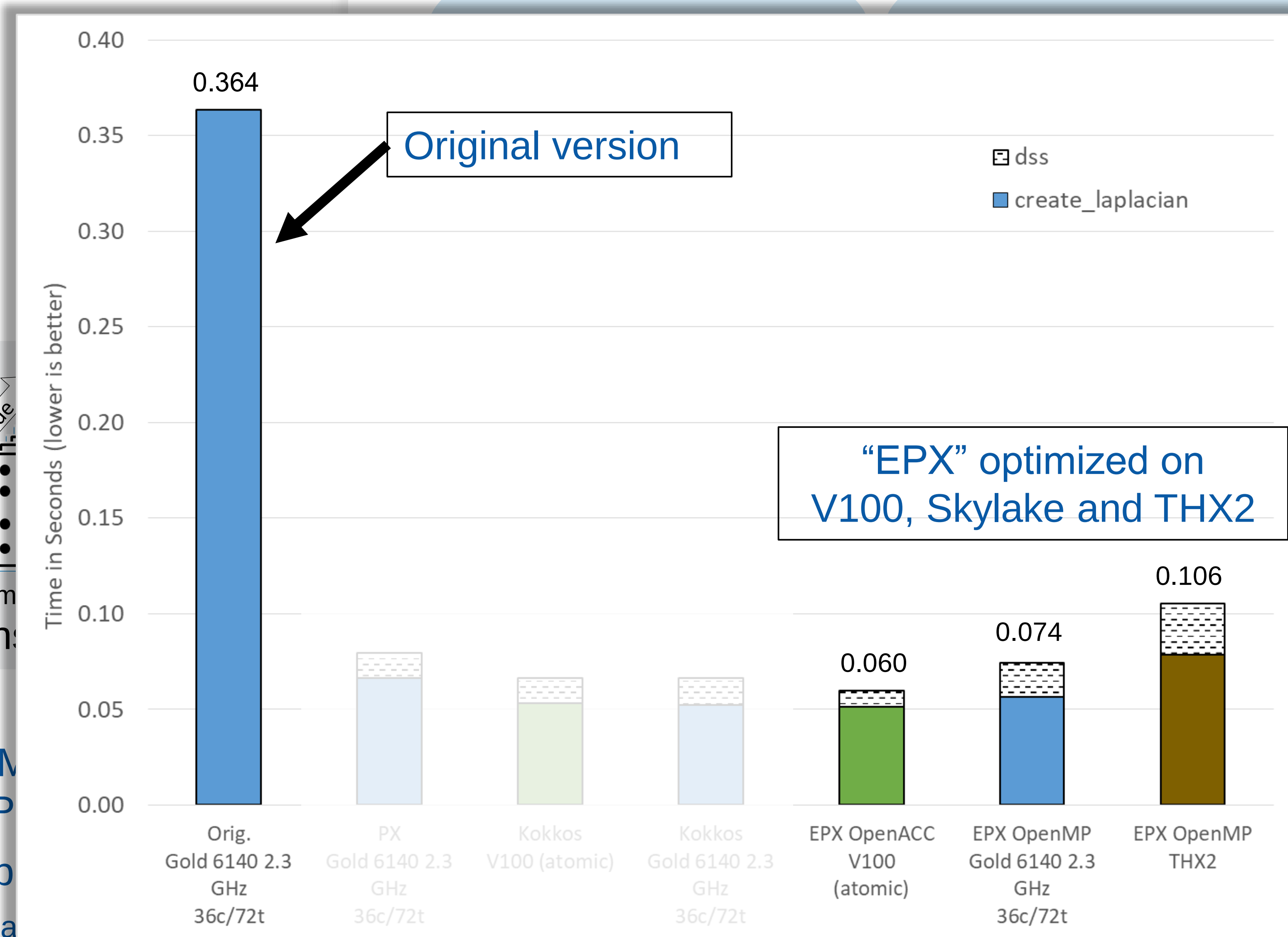
mem. pressure)

Element Loop Nesting

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

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

EPX (element-inner) Optimization



unit stride

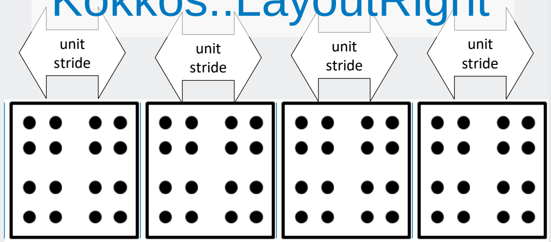
elem

dimens

- Xeon, ARM
  OpenMP
- Nvidia: Op
  (Thanks Da

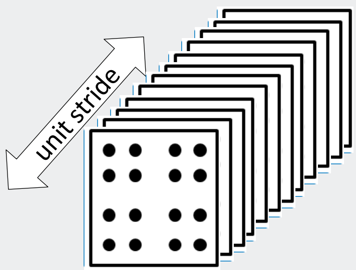
Kokkos Implementation

Kokkos::LayoutRight



element-**outer** arrays

dimension(np,nv,ne)



element-**inner** arrays

dimension(ne,np,nv)

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

(Thanks: C. Trott, Sandia NL)

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

Loop Nesting

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

element-**outer** loops

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

Kokkos Implementation

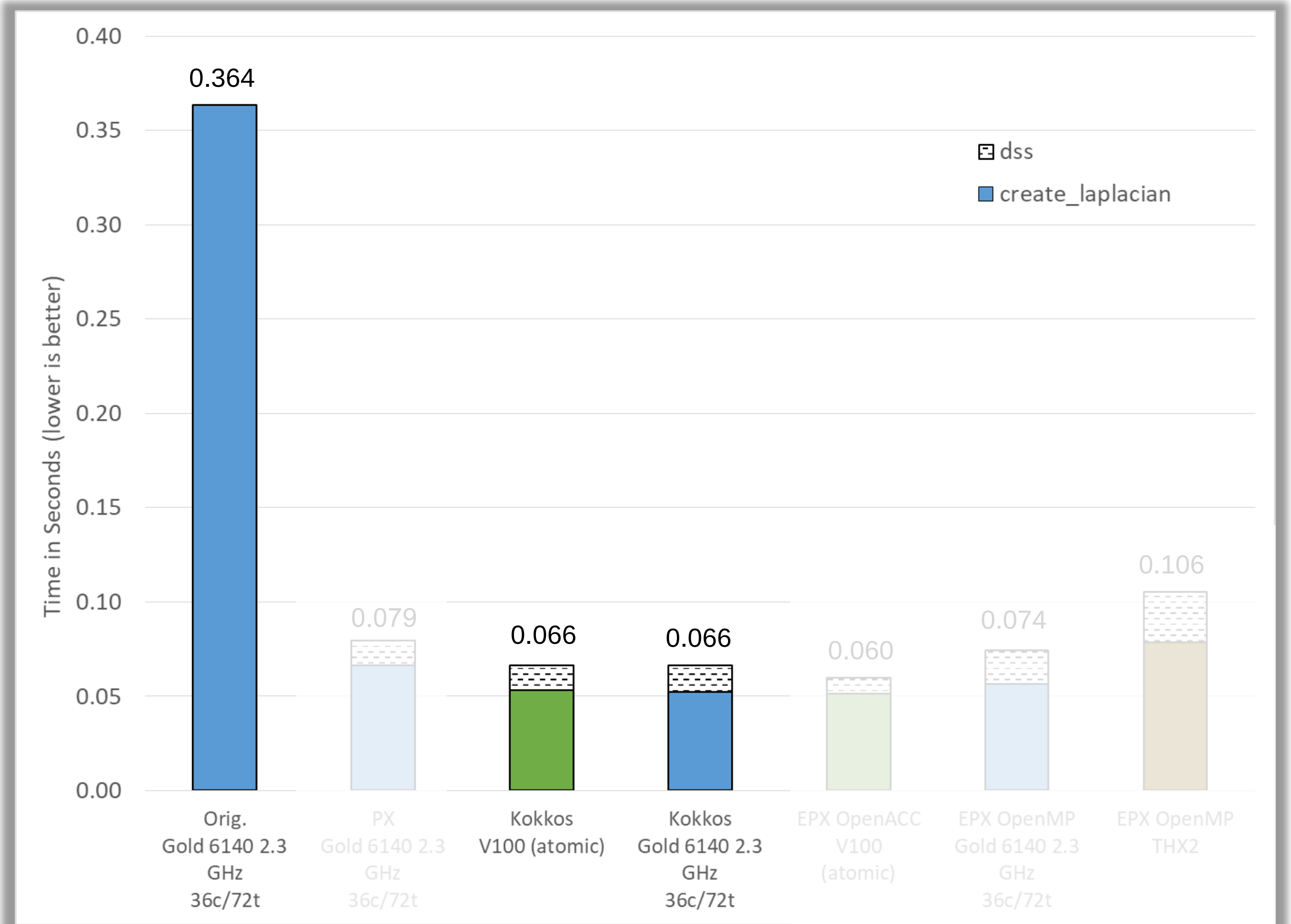
```
//  Define Functor Class and Operators
typedef Kokkos::View<double [nelem][nvar][npts]>    ViewNvarType ;
class CreateLaplacianFunctor {
  ViewNvarType  _q, _rhs ;
  KOKKOS_INLINE_FUNCTION
  CreateLaplacianFunctor(
                        const ViewNvarType q , const ViewNvarType rhs
           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

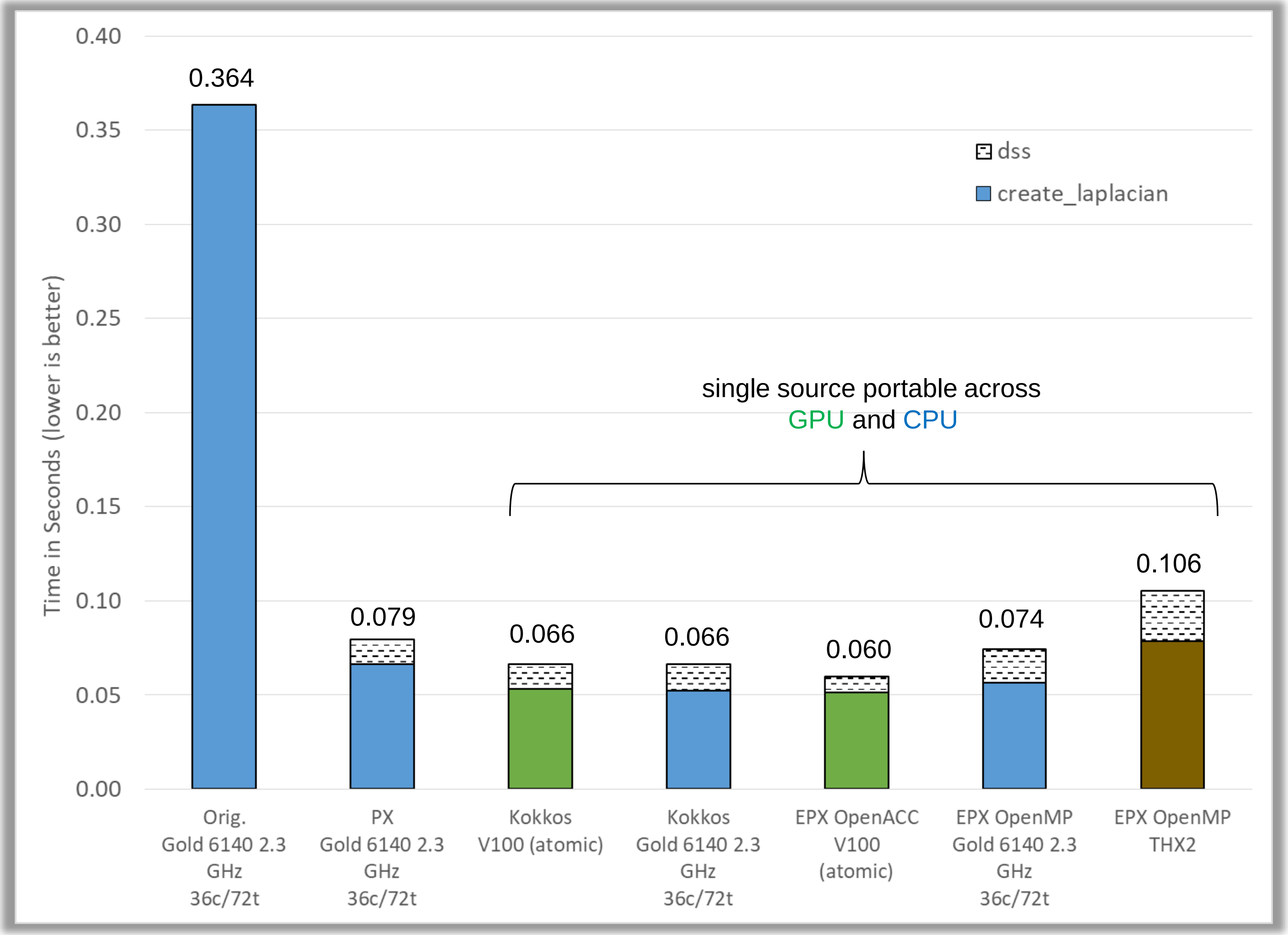
- Te
- Si
- https://github.com/kokkos

(Thanks: C. Trott, Sandia NL)

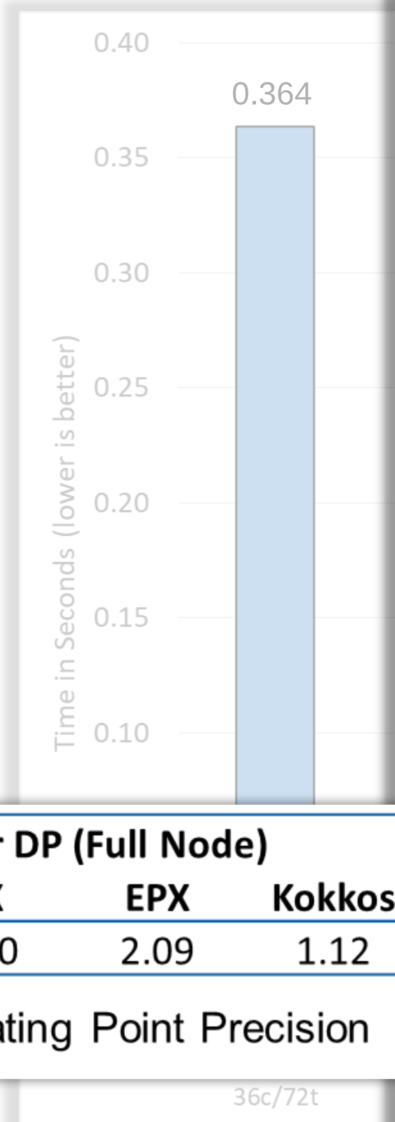
Performance results



Performance results



Performance results



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

Effect of Floating Point Precision

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

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

Diffusion Kernel Performance Summary

Competitive Performance over Programming Models and Devices

performance portable

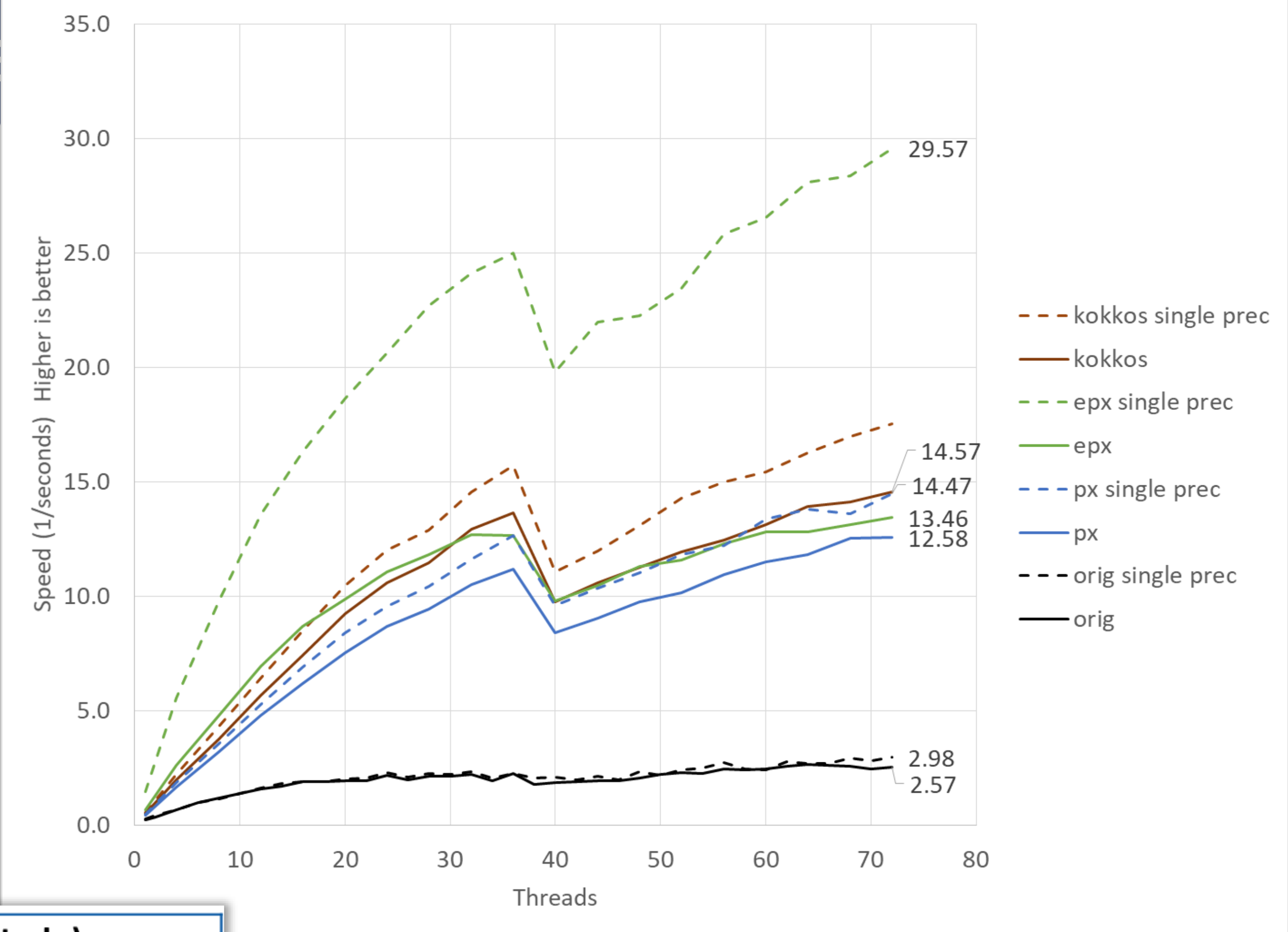
- ✓ Kokkos
  - – GPU: Excellent fine-grained utilization on GPU
    - Good occupancy; 25% to 100%; moderate register pressure
  - – CPU: Nearly identical performance to GPU
    - Kokkos fails to exploit vectorization on CPU (8%) because of strictly element-outer loops that only benefit OpenMP threading.
    - 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, PSyKAl
- Whole code prototypes and testing
- Recommendation on refactoring with costs, benefits and timelines for different options

	Perf./Port.	Effort	Benefits
PX	Fortran CPU only Limited vectorization	Low: Already have this. Good modularity & object-orientedness	Good performance on CPU architectures
EPX	Fortran Fine-grain || Single-source CPU: OpenMP/Vector GPU: OpenACC	Moderate: Refactor data structures, loops; Preserve modularity, object-orientedness	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
Kokkos	C++ Fine-grain || Single-source GPU: native CUDA CPU: maybe vector?	Extreme: C++ rewrite!	Good performance on CPU and GPU; Piggyback DOE's Kokkos for future architectures; C++ resources and marketplace

Performa



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

Effect of Floating Point Precision