REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED *(From - To)*
08 May 2017	Briefing Charts	05 April 2017 - 08 May 2017

4. TITLE AND SUBTITLE
Using Kokkos for Performant Cross-Platform Acceleration of Liquid Rocket Simulations

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Michael Carilli

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
Q1FZ

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory (AFMC)
AFRL/RQR
5 Pollux Drive
Edwards AFB, CA 93524-7048

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory (AFMC)
AFRL/RQR
5 Pollux Drive
Edwards AFB, CA 93524-7048

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-RQ-ED-VG-2017-066

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA Clearance Number: 17207 Clearance Date: 21 April 2017

13. SUPPLEMENTARY NOTES
For presentation at GPU Technology Conference 2017; San Jose, CA, USA; May 8, 2017
Prepared in collaboration with ERC, Inc. The U.S. Government is joint author of the work and has the right to use, modify, reproduce, release, perform, display, or disclose the work.

14. ABSTRACT
Viewgraph/Briefing Charts

15. SUBJECT TERMS
N/A

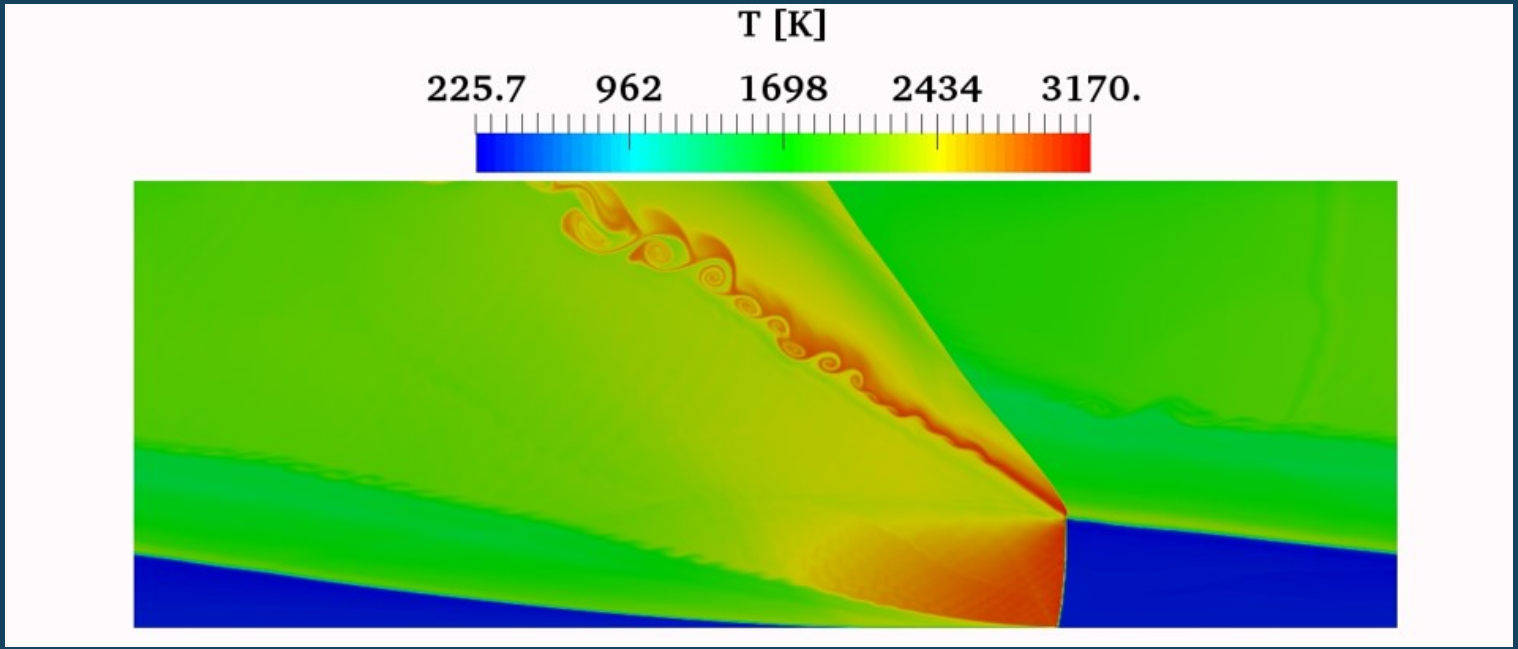
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Venke Sankaran
Unclassified	Unclassified	Unclassified	SAR	56	19b. TELEPHONE NUMBER *(Include area code)* N/A

Using Kokkos for Performant Cross-Platform Acceleration of Liquid Rocket Simulations

Dr. Michael Carilli
Contractor, ERC Incorporated
RQRC
AFRL-West

May 8, 2017

SPACE-LRC: Scalable Physics-based Advanced Computational Engineering Platform for Liquid Rocket Combustion Simulation

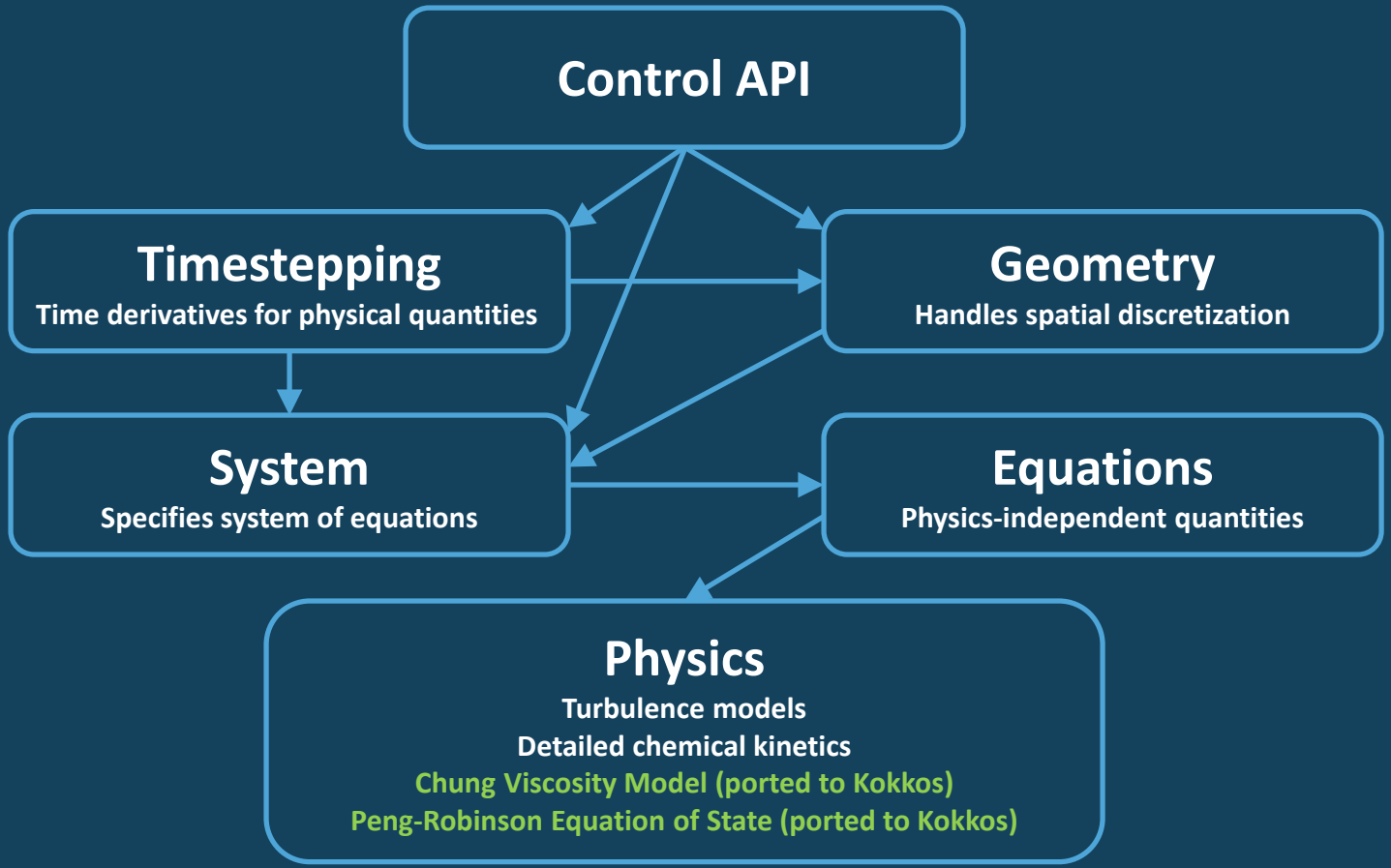


SPACE simulation of rotating detonation engine
(courtesy of Dr. Christopher Lietz)

Components of SPACE-LRC

- GEMS (Purdue University): Unstructured near-body solver
- CASTLES (AFRL-West): High-order Cartesian off-body solver
  - Kokkos is integrated into CASTLES
- PUNDIT (CREATE-AV): Mesh communication between GEMS and CASTLES
- SAMRAI (LLNL): Adaptive meshing for off-body

Structure of CASTLES



What is Kokkos?

“Performant cross platform parallelism”: write once, compile for anything.

Parallel patterns (for, reduce, scan) accept user-defined functors (like Thrust or Intel TBB)

Backends for Nvidia GPU, Intel Xeon, Xeon Phi, IBM Power8, others.

“View” data structure provides optimal layout:
cache-order access when compiled for CPU, coalesced access when compiled for GPU.

Thrust offers similar multi-platform backends – but less low level control and does not abstract data layout.

Programming Guide:
https://github.com/kokkos/kokkos/blob/master/doc/Kokkos_PG.pdf

At GTC 2017:
S7344 - Kokkos : The C++ Performance Portability Programming Model
S7253 - Kokkos Hierarchical Task-Data Parallelism for C++ HPC Applications

Enabling Kokkos in CASTLES

CASTLES is a Cartesian solver written in Fortran 90.

- Identify performance limiting subroutines
- Port Fortran subroutines to Kokkos C++
- Optimize ported routines
- Minimally invasive integration of Kokkos C++ with CASTLES (“code surgery”)

Identify critical subroutines – CPU profile

Quick and easy single-process profile with nvprof:

```
nvprof --cpu-profiling on
--cpu-profiling-mode top-down ./CASTLES.x
```

I like the top-down view…
easy to see global structure and call chains.

Can also do bottom up profile (default)

Identify critical subroutines – CPU profile

Quick and easy single-process profile with nvprof:

```
nvprof --cpu-profiling on
--cpu-profiling-mode top-down ./CASTLES.x
```

I like the top-down view…
easy to see global structure and call chains.

Can also do bottom up profile (default)

Looks like those “preos” and “chung” routines are burning a lot of CPU time

```
======== CPU profiling result (top down):
51.29% clone
| 51.29% start_thread
|   51.29% orte_progress_thread_engine
|     51.29% opal_libevent2021_event_base_loop
|       51.29% poll_dispatch
|         51.29% poll
48.54% MAIN__
| 48.45% interfacetime_mp_maintimeexplicit_
| | 48.45% interfacetime_mp_rhstimessp34_
| |   29.77% interfacegeom_mp_rhsgeomrescalc_
| |   | 15.46% interfacegeom_mp_rhsgeom3dresad1lr_
| |   | | 15.35% interfacesysexternal_mp_rhssysupdiss_
| |   | | | 15.35% interfacesysinternal_mp_rhssysscalarupdiss_
| |   | | |   9.85% eosmodule_mp_eoscalcrhoh0fromtp_
| |   | | |   | 9.64% eosmodule_mp_eosrhohfromtpprop_
| |   | | |   |   9.64% preosmodule_mp_preosrhohfromtpprop_
...
| |   | | |   5.18% eosmodule_mp_eosgammajacobianproperties_
| |   | | |     5.10% preosmodule_mp_preosgammajacobianproperties_
...
| |   | 13.90% interfacegeom_mp_rhsgeom3dviscres2_
| |   | | 13.84% interfacesysexternal_mp_rhssysviscflux_
| |   | |   13.32% preosmodule_mp_preosviscousfluxproperties_
| |   | |   | 7.85% chungtransmodule_mp_chungcalctransprop_
...
| |   | |   | 3.27% preosmodule_mp_preoscriticalstate_
...
| |   18.33% interfacegeom_mp_bcgeomrescalc_
| |   | 14.77% interfacegeom_mp_bcgeomsubin_
| |   | | 14.77% interfaceeqnfluids_mp_bcfluidseqnsubin_velocity_
| |   | |   14.77% preosmodule_mp_preoscalctfromhp_
...
| |   | 3.56% interfacesysexternal_mp_stepsys3dcalcqadd_
| |   |   3.53% eosmodule_mp_eosthermalproperties_
| |   |   | 3.50% preosmodule_mp_preosthermalproperties_
...
```

Peng-Robinson equation of state and Chung transport model

Peng-Robinson Equation of State:
Computes physical properties (density, enthalpy, etc.) for real gas mixtures at high pressure

Chung Transport Model:
Computes transport properties (viscosity, thermal conductivity, mass diffusivity) for real gas mixtures at high pressure

Many underlying subroutines shared between Chung and P-R.

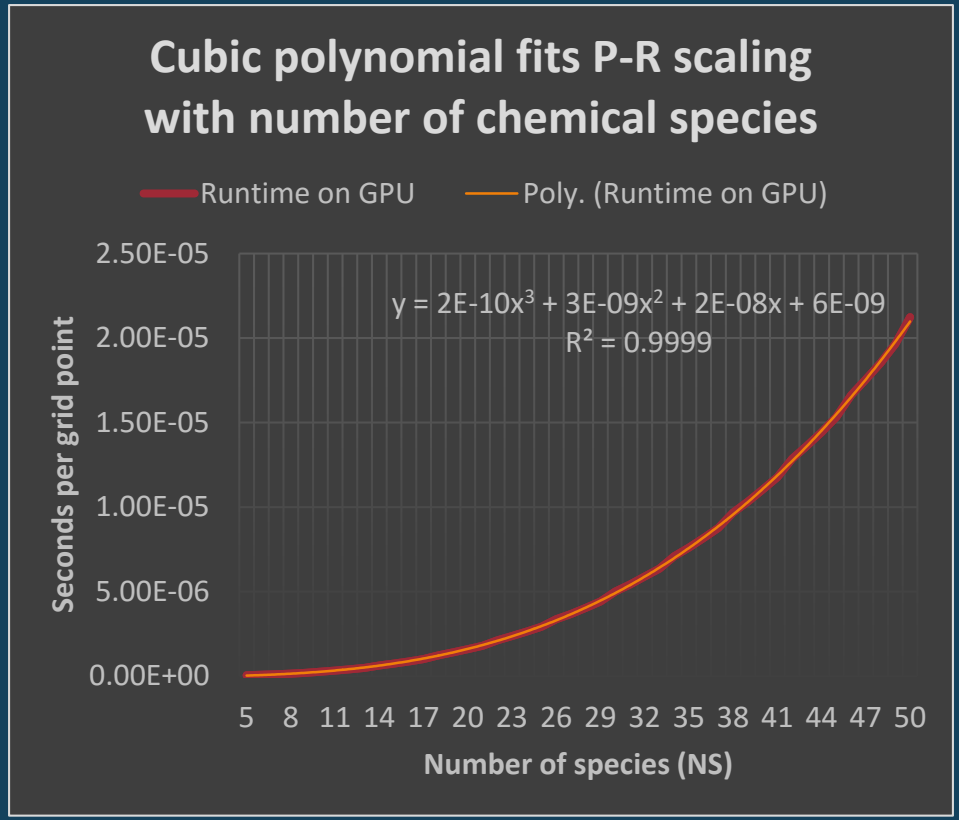
Properties are computed individually per cell
(or interpolated points at cell interfaces),
so **trivially parallel**

Relatively small data transfer, lengthy computation
=> perfect for GPU offload

Input/output data scales linearly with number of species (NS)

Subroutines contain single loops, double loops, triple loops over NS
=> runtime scales like $a*NS + b*NS^2 + c*NS^3$

Occupies significant majority of CASTLES runtime for ns >= 4ish



Architecture of my Kokkos framework

Designed for minimally-invasive operation alongside large Fortran code.

Frame

```
// Owns and allocates TVProperties object
TVProperties* tvproperties;
```

```
// Controls Kokkos initialization/finalization
void initialize(…);
void finalize(…);

TVProperties* gettvproperties();
```

Everything is controlled from Fortran through a single lightweight global Frame object.

Kernel launches and data comms are referred to TVProperties* owned by Frame.

Architecture of my Kokkos framework

Designed for minimally-invasive operation alongside large Fortran code.

Frame

```
// Owns and allocates TVProperties object
TVProperties* tvproperties;

// Controls Kokkos initialization/finalization
void initialize(…);
void finalize(…);

TVProperties* gettvproperties();
```

TVProperties

```
// Owns and allocates TVImpl object
TVImpl* impl;

// Public member functions to communicate data
// to/from Views in TVImpl
void populateInputStripe(…);
void populateOutputStripe(…);
void populateprEOSSharedData(…);
void populatechungSharedData(…);
…

// Public member functions to launch collections of
// kernels
void prEOSThermalProperties(…);
void prEOSViscousProperties(…);
void eosGammaJacobianProperties(…);
…
```

Everything is controlled from Fortran through a single lightweight global Frame object.

Kernel launches and data comms are referred to TVProperties* owned by Frame.

Architecture of my Kokkos framework

Designed for minimally-invasive operation alongside large Fortran code.

Frame

```
// Owns and allocates TVProperties object
TVProperties* tvproperties;

// Controls Kokkos initialization/finalization
void initialize(…);
void finalize(…);

TVProperties* gettvproperties();
```

TVProperties

```
// Owns and allocates TVImpl object
TVImpl* impl;

// Public member functions to communicate data
// to/from Views in TVImpl
void populateInputStripe(…);
void populateOutputStripe(…);
void populateprEOSSharedData(…);
void populatechungSharedData(…);
…

// Public member functions to launch collections of
// kernels
void prEOSThermalProperties(…);
void prEOSViscousProperties(…);
void eosGammaJacobianProperties(…);
…
```

TVImpl

```
// Contains members of TVProperties that don't need
// external visibility (pimpl idiom)
// Owns and allocates Kokkos Views
View1DType T;
View1DType P;
View1DType Yi;
…(several dozen of these)

// Owns std::unordered_maps to launch kernels
// and communicate data by name
unordered_map<string,View1DType>
      select1DViewByName;
unordered_map<string,View2DType>
      select2DViewByName;
// Owns Launcher for each kernel
// (lightweight wrapper with string identifier,
// inherits common timing routines from
// LauncherBase)
unordered_map<string,LauncherBase*> launchers;
```

```
void safeLaunch(…);
```

Everything is controlled from Fortran through a single lightweight global Frame object.

Kernel launches and data comms are referred to TVProperties* owned by Frame.

For modularity and consistency: one subroutine->one kernel

Fortran subroutine

```
pure subroutine prEOSCalcSoundSpeed(&
        rho, rhop, rhoT, hp, hT, c)
    use useGENKindDefs, only: dp
    implicit none
    real(dp), intent(in)  :: rho, rhop, rhoT, hp, hT
    real(dp), intent(out) :: c

    c = sqrt( rho*ht/&
        ( rho*rhop*ht + rhot*( 1.0_dp-rho*hp ) ) )

end subroutine prEOSCalcSoundSpeed
```

Operates on a single grid point at a time

Kokkos kernel launch

```
parallel_for( tvimpl->nActivePoints,
    KOKKOS_LAMBDA(const int& t)
    {
        c(t) = sqrt( rho(t)*hT(t)/
            ( rho(t)*rhoP(t)*hT(t)
            + rhoT(t)*( 1.0-rho(t)*hP(t) ) ) )
    } );
```

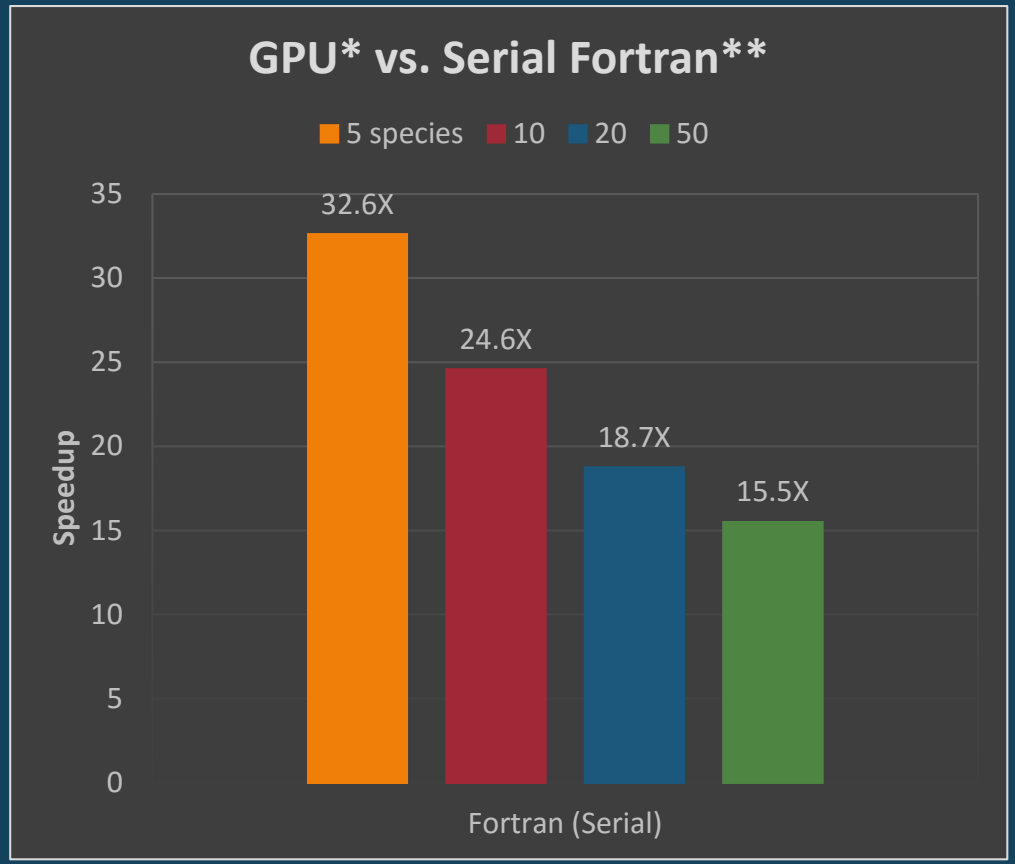
Operates on nActivePoints grid points in parallel

`c`, `rho`, `hT`, etc. are Kokkos Views,
captured by value from members of `TVImpl`
(View copy constructor is a lightweight shallow copy)

`t` is the parallel work index

There are roughly 50 of these that serve as building blocks.

GPU Speedups for Standalone Peng-Robinson



*Nvidia Kepler K40

**Intel Xeon E5-2620 v3 CPU

Good speedups overall.

GPU speedup is better for fewer species (NS)

- smaller per-thread data set => improved cache hit rates on GPU
- Smaller inner loops => vectorization less efficient on CPU

(a combination of GPU doing better and CPU doing a bit worse)

Integrating Kokkos with CASTLES: Interface Functions

C++ Interface functions (callable from Fortran) tell Frame object to initialize/finalize Kokkos, launch collections of kernels, or communicate data.

Decorated with 'extern "C"' to disallow name mangling, added trailing underscore_ expected by Fortran linker.

Interface function to initialize Kokkos and allocate internal storage

```
extern "C" void frame_initialize_( int device_id,
    int nGridPoints
    int ns
    int nq
    int iTurb )
{
    frame.initialize( device_id, // GPU device to select
        nGridPoints, // Chunk size for Kokkos launches
        ns,          // Num chemical species
        nq,          // Utility values
        iTurb );
}
```

Corresponding Fortran call

```
! Compute KokkosDeviceID as MPI rank%num devices
! Num devices is supplied by input file
call frame_initialize( %VAL( KokkosDeviceID,&
    %VAL( KokkosMaxBlock ),&
    %VAL( nspe ),&
    %VAL( nq ),&
    %VAL( iTurbType ) )
```

Interface function to launch collection of kernels for thermal and viscous properties

```
extern "C" void
frame_tvproperties_eosthermalandviscousproperties_(
    int nActivePoints )
{
    frame.gettvproperties()->eosThermalAndViscousProperties(
        nActivePoints );
}
```

Corresponding Fortran call

```
call
frame_tvproperties_eosthermalandviscousproperties&
    (%VAL(NumThisStripe))
```

Communicating Data

Data communication must translate between 4D Fortran pointers (x,y,z,dataindx) and Kokkos Views. For some computations, a halo of fringe points must be ignored.

C++ interface function

```
extern "C" void frame_castles_populateinputstripe_(
const char name[8],      // Name tag of destination View
double* data,            // Source pointer (from Fortran)
int nx, int ny, int nz,  // Dims of block (including fringes)
int SptX, int EptX,      // Fringe boundaries in x-direction
int SptY, int EptY,      // "                    y-direction
int SptZ, int EptZ,      // "                    z-direction
int SptData,             // Start of data region (slowest index)
int EptData,             // End of data region
int stripeStart,         // Start and end of selected x,y,z
int stripeEnd)           // stripe; used when looping over block
                         // in chunks (stripes) of fixed size
{
    frame.gettvproperties()->populateInputStripe(name,
        data, nx, ny, nz, SptX, EptX, SptY, EptY,
        SptZ, EptZ, SptData, EptData, stripeStart, stripeEnd);
}
```

Corresponding Fortran call

```
! Name tag of destination View
tag = "Q"//char(0)
call frame_castles_populateinputstripe(tag,&
    Q,&  ! 4D Fortran pointer, source of copy
    %VAL(NumX), %VAL(NumY), %VAL(NumZ),&
    %VAL(SptX), %VAL(EptX),&
    %VAL(SptY), %VAL(EptY),&
    %VAL(SptZ), %VAL(EptZ),&
    %VAL(SptData), %VAL(EptData),&
    %VAL(SptStripe), %VAL(EptStripe) )
```

Fortran <-> C++ communication works as follows:

1. C++ framework receives double* from Fortran
2. Iterates linearly through x,y,z values, copying data to Views and skipping fringe points.
3. In Views, x,y,z indices are flattened into a single parallel-work index, t.
4. After computation, reverse the process, copying data from Views back into double* storage with data layout expected by Fortran.

C++ framework must know xdim, ydim, zdim, and fringe boundaries to unpack and repack data. Annoying indexing math…

Data marshalling challenges

Challenge #1:

Kokkos launches need enough parallel work (enough grid points) to saturate GPU.

Solution:

Ensure availability of this process' entire block of data where Kokkos interface functions are called. Restructuring some Fortran calling functions was required, but minimal impact to code overall.

Challenge #2:

Block size handled by each process may change between timesteps, due to adaptive mesh refinement. Prefer not to reallocate Kokkos data structures, or worse, exhaust GPU memory.

Solution:

Launch Kokkos computations via a loop over this process' block in chunks of largeish but fixed size "KokkosMaxBlock."

KokkosMaxBlock is a tuning parameter in input file, large enough that one chunk's launch should saturate GPU when 10-20 processes are sharing the GPU via Nvidia Multi-Process Service.

KokkosMaxBlock = 8192 or 12288 usually gives good performance.

Cluster-level concerns: Multiple GPUs per node

Kokkos can handle multiple GPUs.

Standalone Kokkos application:

Pass
`--kokkos-ndevices=2`
on the command line and call
`Kokkos::initialize(int& argc, char* argv[])`
within code.

Kokkos will detect available GPUs and assign MPI ranks to GPUs round robin.

Minor Caveat:
If MPI process is bound to a specific set of cores, Kokkos does not try to select the optimally hardware co-located GPU
(this may have changed since last I checked).

My application (embedded deep within a big Fortran code):

Pass number of available GPU devices in input file.

Manually compute which device to use as (MPI rank%num devices).

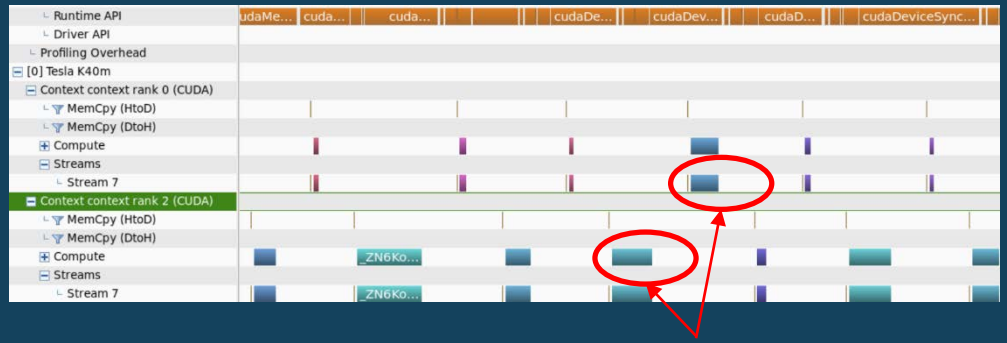
Tell this process' Kokkos kernels to use that device as follows:

```
Kokkos::InitArguments args;
args.device_id = device_id;
Kokkos::initialize( args );
```

Cluster-level concerns: Nvidia Multi-Process Service (MPS)

Without MPS:

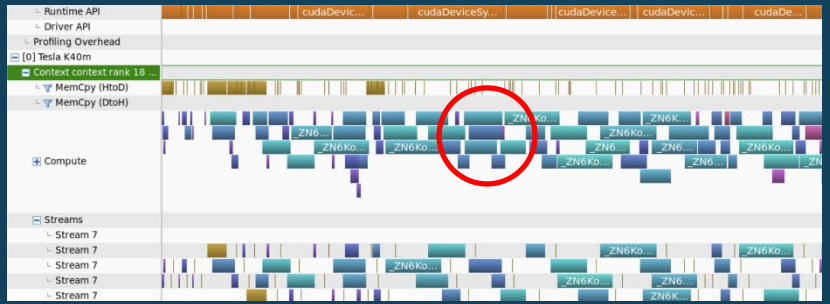
Each MPI process has its own CUDA context.
Multi-process profile shows one process at a time using a given GPU.



Kernels from different processes do not overlap

With MPS:

Multiple processes can share a given GPU simultaneously.

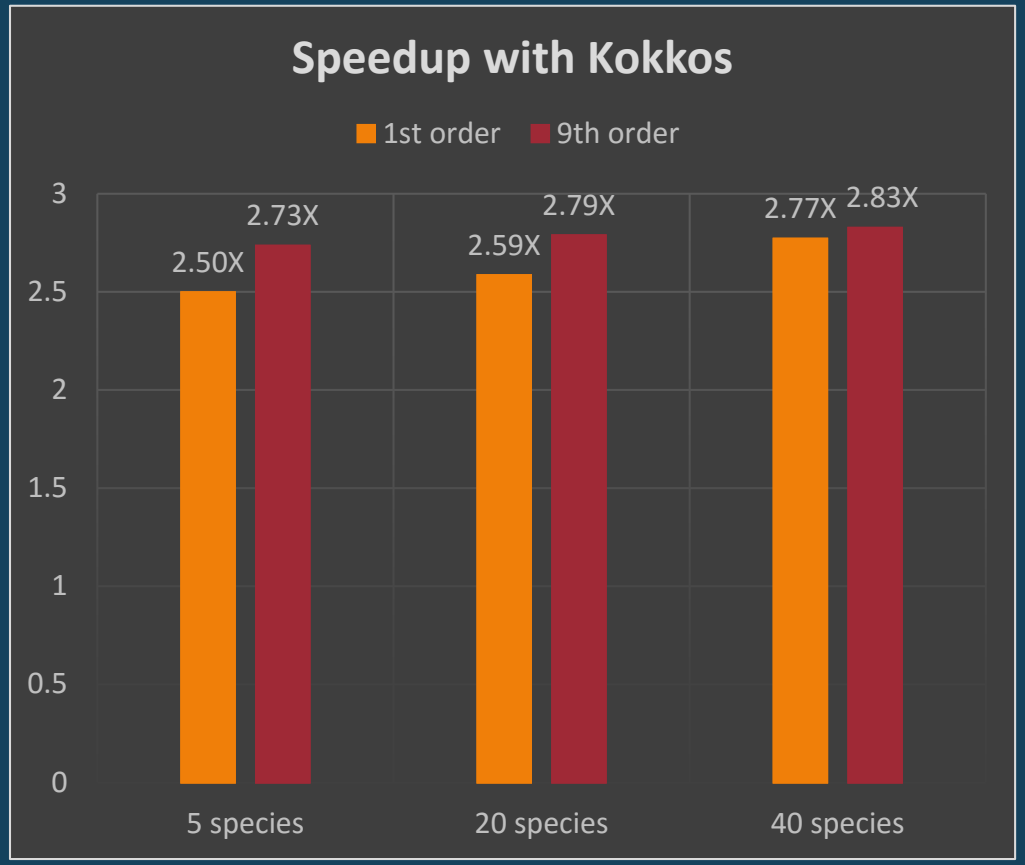


Kernels from different processes overlap
For small NS, turning on MPS makes overall application up to 3X faster

Better utilization and dramatic speedup for my application, and easy to use
(just run `nvidia-cuda-mps-control -d` on each compute node to start the daemons).

See http://on-demand.gputechconf.com/gtc/2016/presentation/s6142-jiri-kraus-multi-gpu-programming-mpi.pdf

GPU Speedup of Overall CASTLES+Kokkos



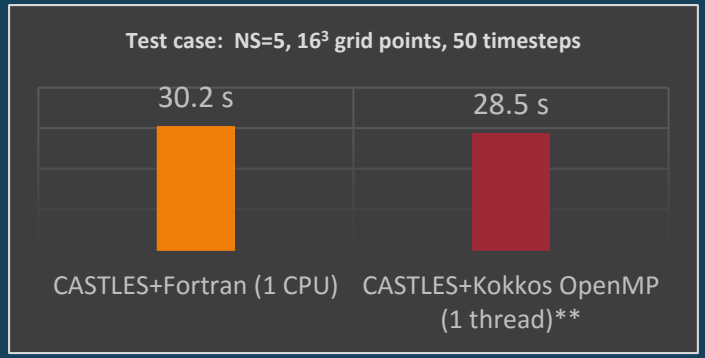
Production-style runs:
40 MPI ranks on 2 nodes.

- CASTLES Fortran uses 20 CPUs/node only.
- CASTLES+Kokkos uses 20 CPUs + 2 GPUs/node.
- Speedup computed as (CASTLES Fortran runtime)/(Castles+Kokkos runtime)

2.5-3.0X consistently observed across range of desirable problem parameters.

Kokkos on CPU matches Fortran on CPU

Can the Kokkos-enabled codebase compile for CPU as well as GPU, with good performance?



**KMP_AFFINITY=compact,1,granularity=core

Often, naively porting Fortran to C++ can result in a slowdown (e.g. compiler has a harder time optimizing/vectorizing loops). Need to use hardware-specific (Intel) compiler and manually tweak vector pragmas for some in-kernel loops, but in the end **Kokkos C++ is as fast as original Fortran.**

To compile for CPU, just change arguments to makefile.

nvcc ignores Intel pragmas. Kokkos-enabled source code is (almost entirely) same as used for GPU.

Only two kernels needed moderately divergent code for good performance on both CPU and GPU. Kokkos build system provides pragmas to select different code when compiling for different hardware:

```
KOKKOS_LAMBDA( const int& t )
{
    #ifdef KOKKOS_HAVE_CUDA
    …GPU-optimal code goes here…
    #else
    …CPU-optimal code goes here…
    #endif
}
```

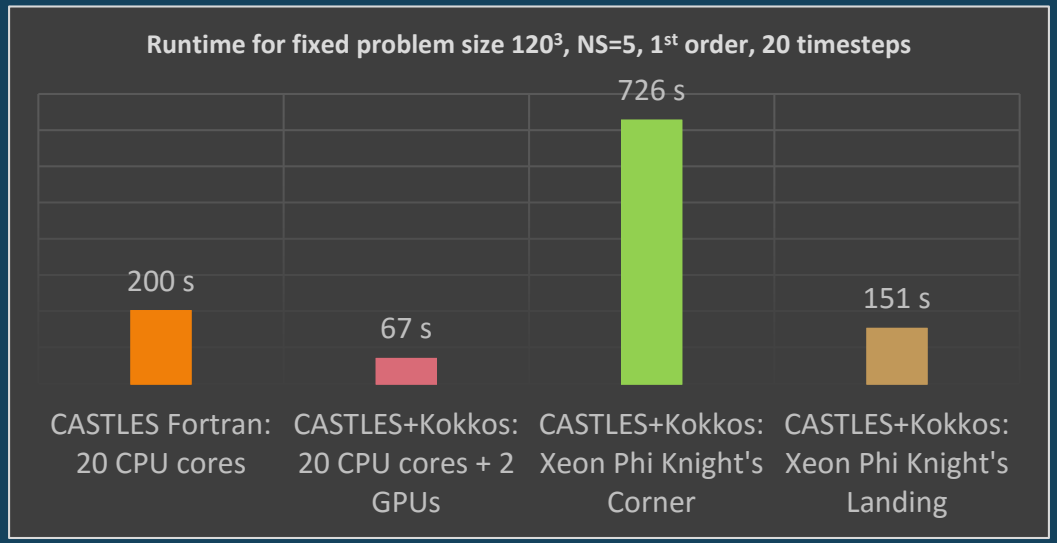
Kokkos promise of "performant cross-platform parallelism" more or less fulfilled.

Node level performance + comparison with Xeon Phi

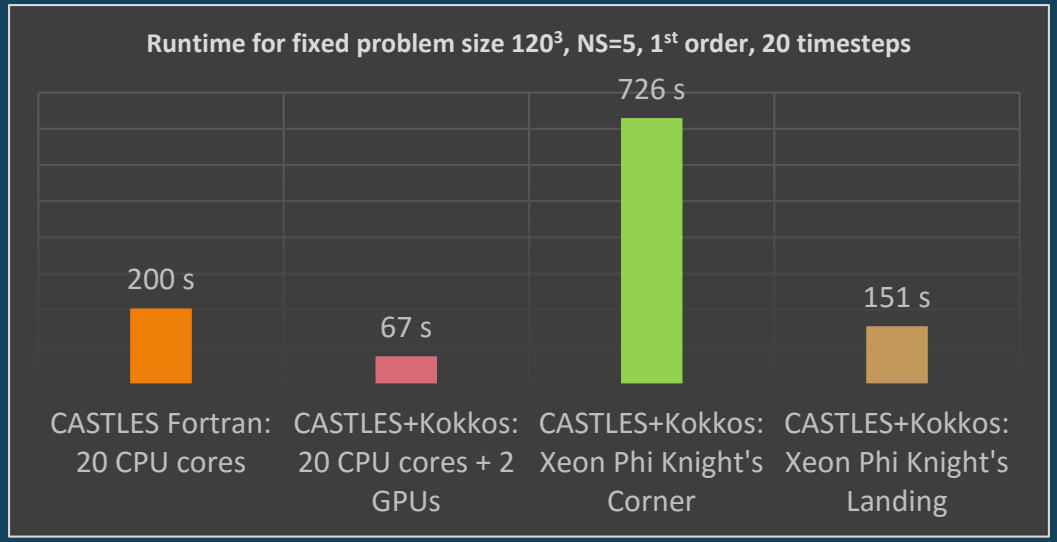
Kokkos runs on Xeon Phis in **native mode**:

- MPI+Kokkos processes see Phi cores as additional CPU cores.
- Kokkos computations are not offloaded GPU-style.
- Entire process runs on a set of Phi cores just like on a multicore CPU.

GPUs are offload coprocessors, so can't compare Phi vs. GPU apples-to-apples. But we can get an idea at node level.



System details



2x10 core Intel Xeon E5-2650 v3
Config file for Intel MPI:

```
-genv I_MPI_PIN_DOMAIN=auto:compact
-n 20 ./CASTLES.kokkos
```

Although cores are hyperthreaded (40 logical cores available),
adding more processes does not improve performance noticeably.

2x10 core Intel Xeon E5-2650 v3
+ 2 Kepler K40 GPUs.
Same MPI config as CASTLES Fortran.

One Knight's Corner 5110P
(60 cores, 240 logical processors).
Config file for Intel MPI:

```
-genv I_MPI_PIN_DOMAIN=4:compact –genv OMP_NUM_THREADS 4
-host mic0 –n 60 –env KMP_AFFINITY=compact,granularity=core ./CASTLES.knc
```

One Knight's Landing 7230 (64 cores, 256 logical processors), using SNC4 clustering
Config file for Intel MPI:

```
-genv I_MPI_PIN_DOMAIN=1:compact -genv OMP_NUM_THREADS 1
-n 256 –env KMP_AFFINITY=compact,granularity=core numactl –m 4,5,6,7 ./CASTLES.knl
```

Numactl –m 4,5,6,7 encourages first-touch allocation in onboard high-bandwidth memory.
I experimented with fewer MPI processes, bigger domains, and more OpenMP threads,
and found 256 procs with 1 thread/proc best.

Bandwidth Optimizations for Per-Grid-Point Inner Loops

P-R and Chung involve nested inner loops over chemical species NS (can be 50 or more).

Independent calculations for each grid point.

Toy example (shown as CPU-style serial loop over grid points):

```
// Loop over N grid points (trivially parallel)
for( int t = 0; t < N; t++ )
    // ax[], ay[], bx[], and by[]:
    // arrays of size NS*N that store per-grid-point input data.
    for( int y = 0; y < NS; y++ ) // NS ~ up to 50ish
        for( int x = 0; x < NS; x++ )
            output[N*(NS*y+x)+t] = ax[x*N+t]*ay[y*N+t] + bx[x*N+t]*by[y*N+t];
```

“Embarrassingly parallel,” and inner loops are simple…
but achieving high performance is an interesting problem!

Bandwidth Optimizations for Per-Grid-Point Inner Loops

P-R and Chung involve nested inner loops over chemical species NS (can be 50 or more).

Independent calculations for each grid point.

Toy example (shown as CPU-style serial loop over grid points):

```
// Loop over N grid points (trivially parallel)
for( int t = 0; t < N; t++ )
    // ax[], ay[], bx[], and by[]:
    // arrays of size NS*N that store per-grid-point input data.
    for( int y = 0; y < NS; y++ ) // NS ~ up to 50ish
        for( int x = 0; x < NS; x++ )
            output[N*(NS*y+x)+t] = ax[x*N+t]*ay[y*N+t] + bx[x*N+t]*by[y*N+t];
```

“Embarrassingly parallel,” and inner loops are simple…
but achieving high performance is an interesting problem!

Testing Parameters

Tesla K40 GPU

- 12 GB device memory
- 15 Kepler SMs

Kepler architecture:

- 192 single-precision cores and 64 double-precision cores per SM
- 100% occupancy = 2048 active threads per SM
- 65,536 registers available per SM
- 64KB L1 cache/shared memory per SM, configurable as either 48 KB L1 + 16 KB shared, 32 KB L1 + 32 KB shared, or 16 KB L1 + 32 KB shared
- 48 KB read-only cache (declare pointers with const __restrict__ to use this**)

Compiled with nvcc version 7.5, opt-in L1 caching, verbose to see register/local mem use, targeting compute capability 3.5

```
nvcc -Xptxas="-dlcm=ca" –Xptxas="-v" –arch=sm_35 kernels.cu
```

Runtime call to `cudaDeviceSetCacheConfig(cudaFuncCachePreferL1)` to set the 48 KB L1 + 16 KB shared option in case the compiler chooses to load via L1

For timing purposes, I use N=2048*120, NS=64, 960 blocks, 256 threads/block. On a K40 with 15 SMs, this is 8 full waves.

Kernel wall times averaged over 10 trials.

**In subsequent examples, I do not write "const." Although the Kepler Tuning Guide is pretty adamant that writing "const" is necessary to trigger loads via the 48 KB read-only cache, I found that for toy kernels presented here, the compiler uses read-only cache even if "const" is omitted.

Naïve Cuda Kernel – one thread per grid point

```
__global__ void naive( double* __restrict__ ax, double* __restrict__ bx,
                       double* __restrict__ ay, double* __restrict__ by, double* __restrict__ output )
{
    // Ordinarily we might wrap this in a grid stride loop…omitted to save space.
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    #pragma unroll 1 // Disallow compiler unrolling so we know what's happening.**
    for( int y = 0; y < NS; y++ )
        # pragma unroll 1
        for( int x = 0; x < NS; x++ )
            output[N*(NS*y+x)+t] = ax[x*N+t]*ay[y*N+t] + bx[x*N+t]*by[y*N+t];
}
```

**If we omit the "`#pragma unroll 1`"s and let the compiler unroll as it wishes, register use goes up (as expected), occupancy falls, and the "naïve" kernel's performance worsens. 100% occupancy is not always essential, but in this case, explicitly including the pragmas is better than relying on compiler heuristics.

Naïve Cuda Kernel – one thread per grid point

```
__global__ void naive( double* __restrict__ ax, double* __restrict__ bx,
                       double* __restrict__ ay, double* __restrict__ by, double* __restrict__ output )
{
    // Ordinarily we might wrap this in a grid stride loop…omitted to save space.
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    #pragma unroll 1 // Disallow compiler unrolling so we know what's happening.
    for( int y = 0; y < NS; y++ )
        # pragma unroll 1
        for( int x = 0; x < NS; x++ )
            output[N*(NS*y+x)+t] = ax[x*N+t]*ay[y*N+t] + bx[x*N+t]*by[y*N+t];
}
```

Grid point index "t" is the fast index for coalescing (corresponds to `Kokkos::LayoutLeft`)

Naïve Cuda Kernel – one thread per grid point

```
__global__ void naive( double* __restrict__ ax, double* __restrict__ bx,
                       double* __restrict__ ay, double* __restrict__ by, double* __restrict__ output )
{
    // Ordinarily we might wrap this in a grid stride loop…omitted to save space.
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    #pragma unroll 1 // Disallow compiler unrolling so we know what's happening.
    for( int y = 0; y < NS; y++ )
        # pragma unroll 1
        for( int x = 0; x < NS; x++ )
            output[N*(NS*y+x)+t] = ax[x*N+t]*ay[y*N+t] + bx[x*N+t]*by[y*N+t];
}
```

Grid point index "t" is the fast index for coalescing (corresponds to `Kokkos::LayoutLeft`)

y-dependent loads should hit in cache (or be promoted to registers) during loop over x. I find that manually hoisting y-loads to a register does not affect performance.

Naïve Cuda Kernel – one thread per grid point

```
__global__ void naive( double* __restrict__ ax, double* __restrict__ bx,
                       double* __restrict__ ay, double* __restrict__ by, double* __restrict__ output )
{
    // Ordinarily we might wrap this in a grid stride loop…omitted to save space.
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    #pragma unroll 1 // Disallow compiler unrolling so we know what's happening.
    for( int y = 0; y < NS; y++ )
        # pragma unroll 1
        for( int x = 0; x < NS; x++ )
            output[N*(NS*y+x)+t] = ax[x*N+t]*ay[y*N+t] + bx[x*N+t]*by[y*N+t];
}
```

Grid point index "t" is the fast index for coalescing (corresponds to `Kokkos::LayoutLeft`)

y-dependent loads should hit in cache (or be promoted to registers) during loop over x. I find that manually hoisting y-loads to a register does not affect performance.

Each x-load is used only once per outer y-loop iteration. Probably won't hit in cache on the next outer y-loop iteration.

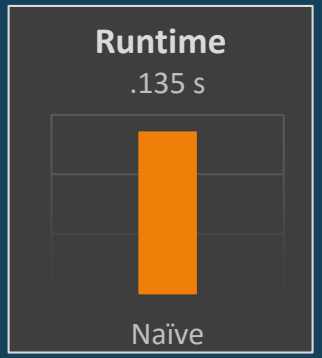
Naïve Cuda Kernel – one thread per grid point

```
__global__ void naive( double* __restrict__ ax, double* __restrict__ bx,
                       double* __restrict__ ay, double* __restrict__ by, double* __restrict__ output )
{
    // Ordinarily we might wrap this in a grid stride loop…omitted to save space.
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    #pragma unroll 1 // Disallow compiler unrolling so we know what's happening.**
    for( int y = 0; y < NS; y++ )
        # pragma unroll 1
        for( int x = 0; x < NS; x++ )
            output[N*(NS*y+x)+t] = ax[x*N+t]*ay[y*N+t] + bx[x*N+t]*by[y*N+t];
}
```

Grid point index "t" is the fast index for coalescing (corresponds to `Kokkos::LayoutLeft`)

y-dependent loads should hit in cache (or be promoted to registers) during loop over x. I find that manually hoisting y-loads to a register does not affect performance.

Each x-load is used only once per outer y-loop iteration. Probably won't hit in cache on the next outer y-loop iteration.



Kernel "naïve" is strongly bandwidth-bound, and accesses are already coalesced. What should we do?

Standard CPU-informed strategy: tile the loop?

Recall why loop tiling helps on CPU:

```
for( int yy = 0; yy < NS; yy += TILE_FACTOR )
    for( int x = 0; x < NS; x++ )
        for( int y = yy; y < yy + TILE_FACTOR; y++ )
            output[N*(NS*y+x)+t] = ax[x*N+t]*ay[y*N+t] + bx[x*N+t]*by[y*N+t];
```

Standard CPU-informed strategy: tile the loop?

Recall why loop tiling helps on CPU:

```
for( int yy = 0; yy < NS; yy += TILE_FACTOR )
    for( int x = 0; x < NS; x++ )
        for( int y = yy; y < yy + TILE_FACTOR; y++ )
            output[N*(NS*y+x)+t] = ax[x*N+t]*ay[y*N+t] + bx[x*N+t]*by[y*N+t];
```

X-dependent loads should hit in cache for the inner y-loop, and be reused TILE_FACTOR times

Standard CPU-informed strategy: tile the loop?

Recall why loop tiling helps on CPU:

```
for( int yy = 0; yy < NS; yy += TILE_FACTOR )
    for( int x = 0; x < NS; x++ )
        for( int y = yy; y < yy + TILE_FACTOR; y++ )
            output[N*(NS*y+x)+t] = ax[x*N+t]*ay[y*N+t] + bx[x*N+t]*by[y*N+t];
```

X-dependent loads should hit in cache for the inner y-loop, and be reused TILE_FACTOR times

Each x-iteration now treats TILE_FACTOR y-iterations instead of just one.

TILE_FACTOR y-dependent loads should hit in cache on each iteration of the x-loop

(in fact, for a typical CPU cache and modest values of NS like 64, the entire working set should easily fit in cache, and it's not necessary to tile the loop at all.)

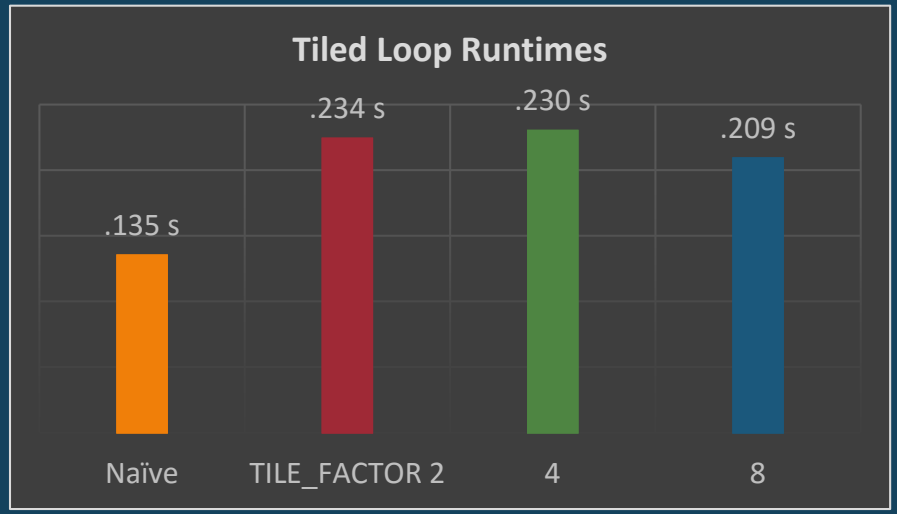
Pretty standard stuff…but do we expect this to work on a Kepler GPU?

Loop tiling on GPU

```
__global__ void tiled(…same args as naïve…)
{
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    for( int yy = 0; yy < NS; yy += TILE_FACTOR )
        for( int x = 0; x < NS; x++ )
            for( int y = yy; y < yy + TILE_FACTOR; y++ )
                output[N*(NS*y+x)+t] = ax[x*N+t]*ay[y*N+t]
                                     + bx[x*N+t]*by[y*N+t];
}
```

Loop tiling on GPU

```
__global__ void tiled(…same args as naïve…)
{
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    for( int yy = 0; yy < NS; yy += TILE_FACTOR )
        for( int x = 0; x < NS; x++ )
            for( int y = yy; y < yy + TILE_FACTOR; y++ )
                output[N*(NS*y+x)+t] = ax[x*N+t]*ay[y*N+t]
                                     + bx[x*N+t]*by[y*N+t];
}
```



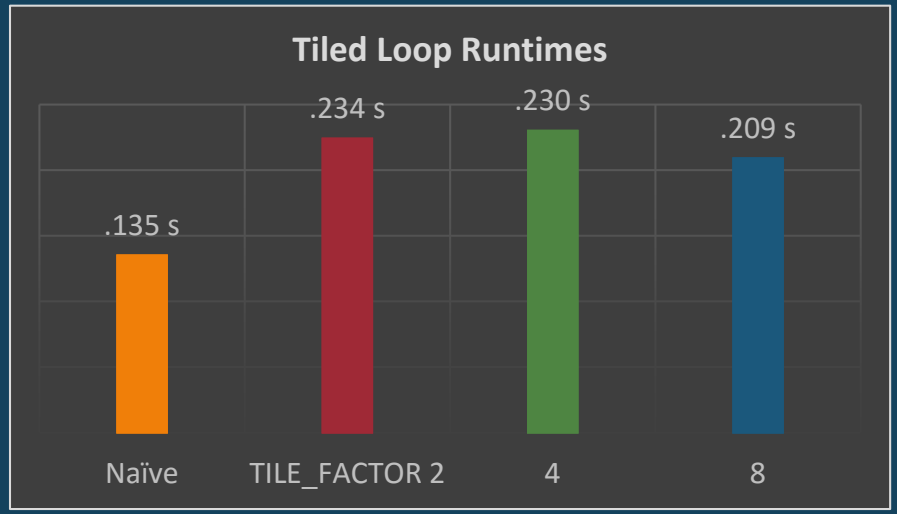
Tiling is worse than naïve. Cache per grid point (thread) is just too small.

Read-only cache and L1 cache are only 48 KB each.
Whichever compiler chooses to use:

100% occupancy = 2048 threads
48 KB/2048 threads = only 3 doubles' worth of cache per thread.

Loop tiling on GPU

```
__global__ void tiled(…same args as naïve…)
{
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    for( int yy = 0; yy < NS; yy += TILE_FACTOR )
        for( int x = 0; x < NS; x++ )
            for( int y = yy; y < yy + TILE_FACTOR; y++ )
                output[N*(NS*y+x)+t] = ax[x*N+t]*ay[y*N+t]
                                     + bx[x*N+t]*by[y*N+t];
}
```



Tiling is worse than naïve. Cache per grid point (thread) is just too small.

Read-only cache and L1 cache are only 48 KB each.
Whichever compiler chooses to use:

100% occupancy = 2048 threads
48 KB/2048 threads = only 3 doubles' worth of cache per thread.

nvprof confirms poor hit rates (results for TILE_FACTOR 2 shown):**

```
nvprof --kernels ::tiled:1 –metrics \
nc_cache_global_hit_rate,tex_cache_hit_rate ./a.out
. . .                                        Min        Max
. . . Non-Coherent Global Hit Rate       0.85%      0.85%
. . .       Texture Cache Hit Rate       0.65%      0.65%
```

**As mentioned previously, the compiler appears to use read-only/texture cache for loads.
I'm not sure why there are separate metrics to describe "read-only cache accesses" and "texture cache accesses" (it's the same hardware). Perhaps some Cuda ninja can explain?

Tile with reduced occupancy

100% occupancy is not a strict requirement for peak performance.
Lower occupancy = **more cache per grid point**.**

Manually suppress occupancy by giving each block “dummy” shared memory.

For example: 16 KB shared memory is available on each SM.
If we assign each block 4096 B smem, only 4 blocks can fit on each SM.
4*256 = 1024 threads. 1024/2048 = 50% occupancy.

```
__global__ void tiled_reduced_occupancy(…)
{
    extern __shared__ int smem[];
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    for( int yy = 0; yy < NS; yy += TILE_FACTOR )
        for( int x = 0; x < NS; x++ )
            for( int y = yy; y < yy + TILE_FACTOR; y++ )
                output[N*(NS*y+x)+t] = ax[x*N+t]*ay[y*N+t]
                                     + bx[x*N+t]*by[y*N+t];
}
```

**See “GPU Memory Bootcamp II: Beyond Best Practices” from GTC 2015 (http://on-demand.gputechconf.com/gtc/2015/presentation/S5376-Tony-Scudiero.pdf) for a more detailed discussion of occupancy vs. hit rate.

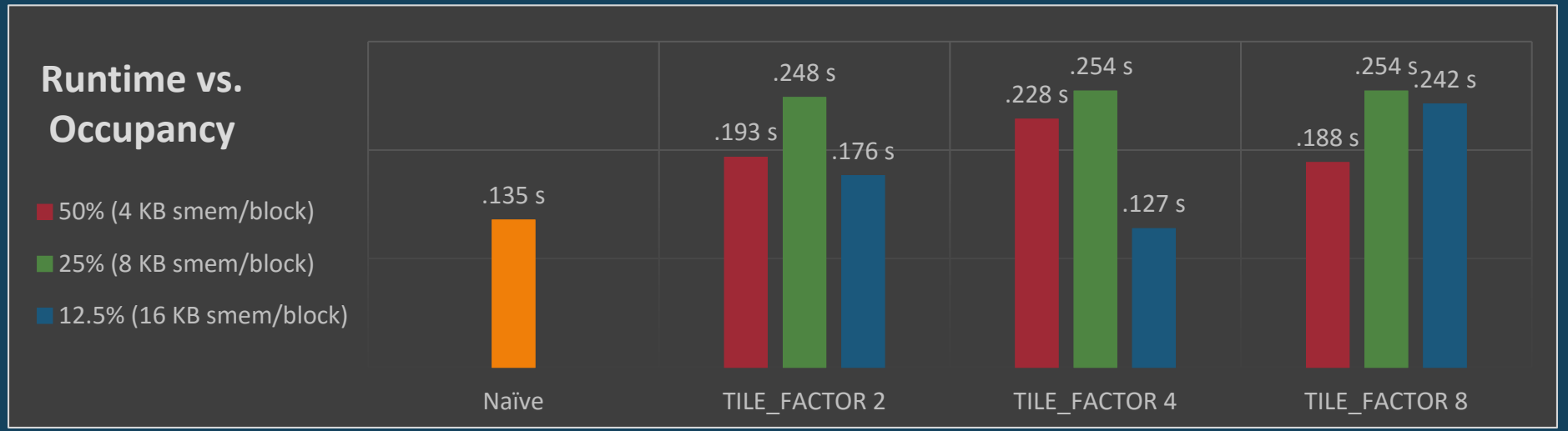
Tile with reduced occupancy

100% occupancy is not a strict requirement for peak performance.
Lower occupancy = **more cache per grid point**.

Manually suppress occupancy by giving each block “dummy” shared memory.

For example: 16 KB shared memory is available on each SM.
If we assign each block 4096 B smem, only 4 blocks can fit on each SM.
4*256 = 1024 threads. 1024/2048 = 50% occupancy.

```
__global__ void tiled_reduced_occupancy(…)
{
    extern __shared__ int smem[];
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    for( int yy = 0; yy < NS; yy += TILE_FACTOR )
        for( int x = 0; x < NS; x++ )
            for( int y = yy; y < yy + TILE_FACTOR; y++ )
                output[N*(NS*y+x)+t] = ax[x*N+t]*ay[y*N+t]
                                     + bx[x*N+t]*by[y*N+t];
}
```



Mostly worse than naïve.

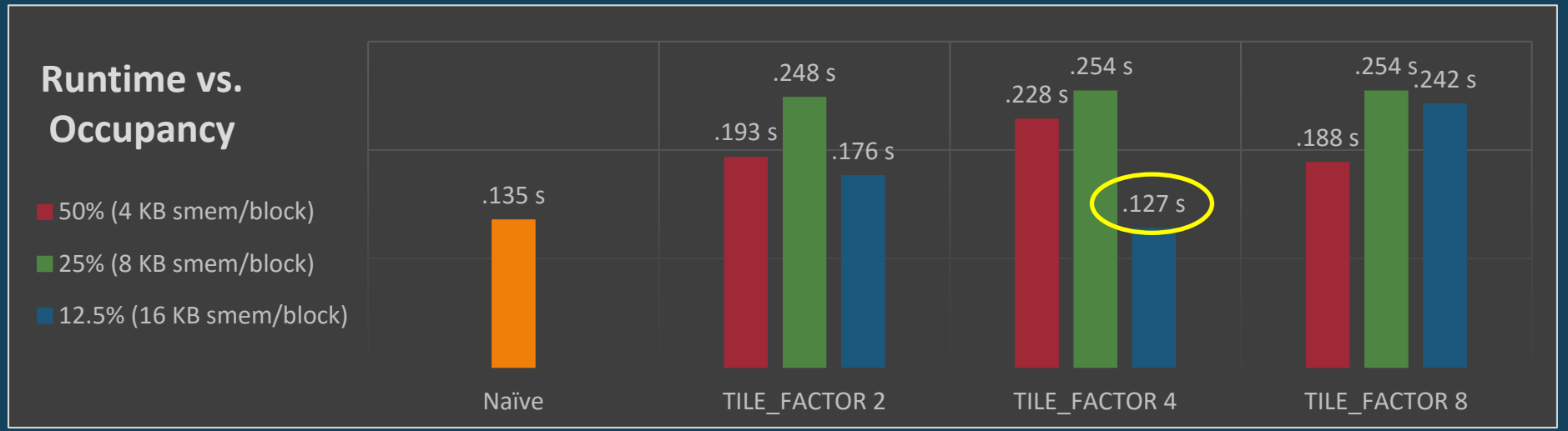
Tile with reduced occupancy

100% occupancy is not a strict requirement for peak performance.
Lower occupancy = **more cache per grid point**.

Manually suppress occupancy by giving each block "dummy" shared memory.

For example: 16 KB shared memory is available on each SM.
If we assign each block 4096 B smem, only 4 blocks can fit on each SM.
4*256 = 1024 threads. 1024/2048 = 50% occupancy.

```
__global__ void tiled_reduced_occupancy(…)
{
    extern __shared__ int smem[];
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    for( int yy = 0; yy < NS; yy += TILE_FACTOR )
        for( int x = 0; x < NS; x++ )
            for( int y = yy; y < yy + TILE_FACTOR; y++ )
                output[N*(NS*y+x)+t] = ax[x*N+t]*ay[y*N+t]
                                     + bx[x*N+t]*by[y*N+t];
}
```



Mostly worse than naïve. Sweet spot at TILE_FACTOR 4, 12.5% occupancy can be explained by cache hits:

```
nvprof --kernels ::tiled_reduced_occupancy:4 --metrics achieved_occupancy,nc_cache_global_hit_rate,tex_cache_hit_rate ./a.out
                          . . . Achieved Occupancy    0.124771    0.124771
                          . . . Non-Coherent Global Hit Rate      75.81%      75.81%
```

Tile using both L1 and read-only cache

On Kepler, 48 KB read-only cache and 64 KB L1+shared cache are independent. Use both!

Tile using thread-local arrays :
(placed in a local memory stack frame. Allocated in device global memory, but cached in L1)**

```
__global__ void tiled_local_arrays(…)
{
    double ay_local[TILE_FACTOR]; // Thread-local arrays
    double by_local[TILE_FACTOR]; // (placed in local memory)
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    for( int yy = 0; yy < NS; yy += TILE_FACTOR )
    {
        for( int y = yy; y < yy + TILE_FACTOR; y++ )
        {
            ay_local[y-yy] = ay[y*N+t];
            by_local[y-yy] = by[y*N+t];
        }
        for( int x = 0; x < NS; x++ )
            for( int y = yy; y < yy + TILE_FACTOR; y++ )
                output[N*(NS*y+x)+t] = ax[x*N+t]*ay_local[y-yy]
                                     + bx[x*N+t]*by_local[y-yy];
    }
}
```

**On Kepler, local loads are cached in L1. On Maxwell, L1/tex is a single unified cache, and local loads are cached in L2 only. Therefore, I expect tiling with local memory to be helpful on Kepler only. Maxwell has separate hardware for shared memory, so you could try using thread-local smem arrays instead.
See https://devblogs.nvidia.com/parallelforall/fast-dynamic-indexing-private-arrays-cuda/ for an in-depth discussion of where the compiler places thread-local arrays.
See http://docs.nvidia.com/cuda/kepler-tuning-guide/#l1-cache and http://docs.nvidia.com/cuda/maxwell-tuning-guide/#l1-cache for microarchitecture details.

Tile using both L1 and read-only cache

On Kepler, 48 KB read-only cache and 64 KB L1+shared cache are independent. Use both!

Tile using thread-local arrays :
(placed in a local memory stack frame. Allocated in device global memory, but cached in L1)

```
__global__ void tiled_local_arrays(…)
{
    double ay_local[TILE_FACTOR]; // Thread-local arrays
    double by_local[TILE_FACTOR]; // (placed in local memory)
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    for( int yy = 0; yy < NS; yy += TILE_FACTOR )
    {
        for( int y = yy; y < yy + TILE_FACTOR; y++ )
        {
            ay_local[y-yy] = ay[y*N+t];
            by_local[y-yy] = by[y*N+t];
        }
        for( int x = 0; x < NS; x++ )
            for( int y = yy; y < yy + TILE_FACTOR; y++ )
                output[N*(NS*y+x)+t] = ax[x*N+t]*ay_local[y-yy]
                                     + bx[x*N+t]*by_local[y-yy];
    }
}
```

Thread-local arrays for Y-dependent loads (cached in L1)

Tile using both L1 and read-only cache

On Kepler, 48 KB read-only cache and 64 KB L1+shared cache are independent. Use both!

Tile using thread-local arrays :
(placed in a local memory stack frame. Allocated in device global memory, but cached in L1)

```
__global__ void tiled_local_arrays(…)
{
    double ay_local[TILE_FACTOR]; // Thread-local arrays
    double by_local[TILE_FACTOR]; // (placed in local memory)
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    for( int yy = 0; yy < NS; yy += TILE_FACTOR )
    {
        for( int y = yy; y < yy + TILE_FACTOR; y++ )
        {
            ay_local[y-yy] = ay[y*N+t];
            by_local[y-yy] = by[y*N+t];
        }
        for( int x = 0; x < NS; x++ )
            for( int y = yy; y < yy + TILE_FACTOR; y++ )
                output[N*(NS*y+x)+t] = ax[x*N+t]*ay_local[y-yy]
                                     + bx[x*N+t]*by_local[y-yy];
    }
}
```

Thread-local arrays for Y-dependent loads (cached in L1)

X-dependent loads cached in read-only

Tile using both L1 and read-only cache

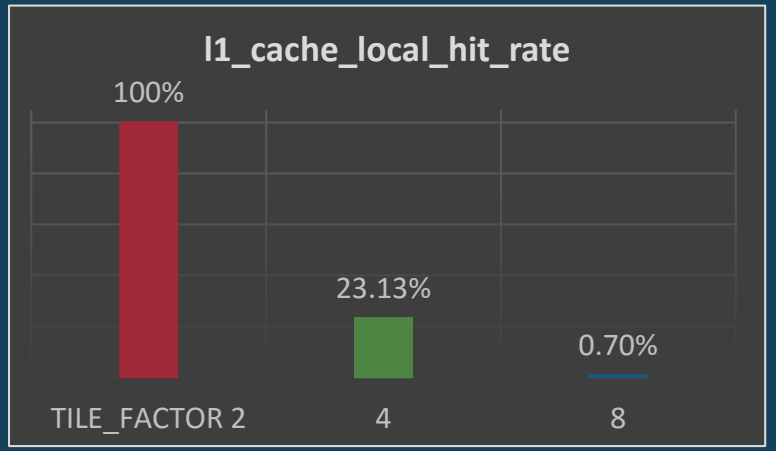
On Kepler, 48 KB read-only cache and 64 KB L1+shared cache are independent. Use both!

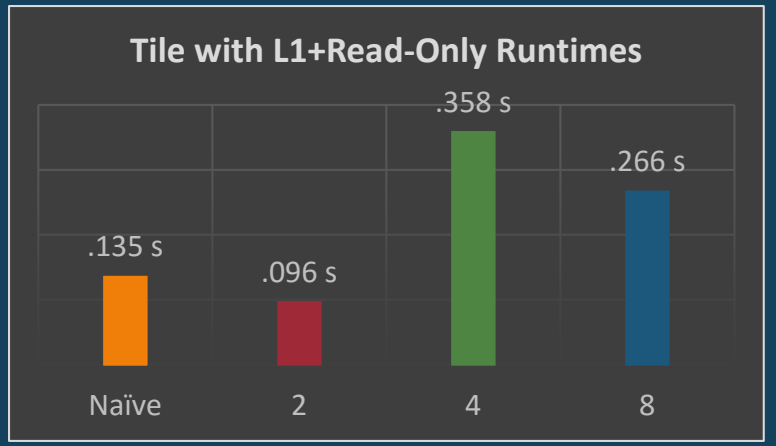
Tile using thread-local arrays :
(placed in a local memory stack frame. Allocated in device global memory, but cached in L1)

```
__global__ void tiled_local_arrays(…)
{
    double ay_local[TILE_FACTOR]; // Thread-local arrays
    double by_local[TILE_FACTOR]; // (placed in local memory)
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    for( int yy = 0; yy < NS; yy += TILE_FACTOR )
    {
        for( int y = yy; y < yy + TILE_FACTOR; y++ )
        {
            ay_local[y-yy] = ay[y*N+t];
            by_local[y-yy] = by[y*N+t];
        }
        for( int x = 0; x < NS; x++ )
            for( int y = yy; y < yy + TILE_FACTOR; y++ )
                output[N*(NS*y+x)+t] = ax[x*N+t]*ay_local[y-yy]
                                     + bx[x*N+t]*by_local[y-yy];
    }
}
```

Thread-local arrays for Y-dependent loads (cached in L1)

X-dependent loads cached in read-only





Fast for TILE_FACTOR = 2! L1 cache fields all y-dependent loads (100% hit rate)

Slower for TILE_FACTOR = 4 and 8. Hit rate decreases.

Tile with explicit register use (“unroll-and-jam”)

Kepler SM has 65,536 4B registers = 262 KB of near-core memory available as registers.

>2.5X more than read-only and L1 caches combined.

```
__global__ void unroll_and_jam_by2_registers(…)
{
    // Encourage these to be placed in registers
    double ay_local0, by_local0, ay_local1, by_local1;
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    #pragma unroll 1
    for( int yy = 0; yy < NS; yy += 2 )
    {   ay_local0 = ay[(yy+0)*N+t];
        by_local0 = by[(yy+0)*N+t];
        ay_local1 = ay[(yy+1)*N+t];
        by_local1 = by[(yy+1)*N+t];
        # pragma unroll 1
        for( int x = 0; x < NS; x++ )
        {
            output[N*(NS*(yy+0)+x)+t] = ax[x*N+t]*ay_local0
                                      + bx[x*N+t]*by_local0;
            output[N*(NS*(yy+1)+x)+t] = ax[x*N+t]*ay_local1
                                      + bx[x*N+t]*by_local1;
        }
    }
}
```

Tile with explicit register use (“unroll-and-jam”)

Kepler SM has 65,536 4B registers = 262 KB of near-core memory available as registers.

>2.5X more than read-only and L1 caches combined.

```
__global__ void unroll_and_jam_by2_registers(…)
{
    // Encourage these to be placed in registers
    double ay_local0, by_local0, ay_local1, by_local1;
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    #pragma unroll 1
    for( int yy = 0; yy < NS; yy += 2 )
    {   ay_local0 = ay[(yy+0)*N+t];
        by_local0 = by[(yy+0)*N+t];
        ay_local1 = ay[(yy+1)*N+t];
        by_local1 = by[(yy+1)*N+t];
        # pragma unroll 1
        for( int x = 0; x < NS; x++ )
        {
            output[N*(NS*(yy+0)+x)+t] = ax[x*N+t]*ay_local0
                                      + bx[x*N+t]*by_local0;
            output[N*(NS*(yy+1)+x)+t] = ax[x*N+t]*ay_local1
                                      + bx[x*N+t]*by_local1;
        }
    }
}
```

Y-dependent loads reused x times in x-loop

Tile with explicit register use (“unroll-and-jam”)

Kepler SM has 65,536 4B registers = 262 KB of near-core memory available as registers.

>2.5X more than read-only and L1 caches combined.

```
__global__ void unroll_and_jam_by2_registers(…)
{
    // Encourage these to be placed in registers
    double ay_local0, by_local0, ay_local1, by_local1;
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    #pragma unroll 1
    for( int yy = 0; yy < NS; yy += 2 )
    {   ay_local0 = ay[(yy+0)*N+t];
        by_local0 = by[(yy+0)*N+t];
        ay_local1 = ay[(yy+1)*N+t];
        by_local1 = by[(yy+1)*N+t];
        # pragma unroll 1
        for( int x = 0; x < NS; x++ )
        {
            output[N*(NS*(yy+0)+x)+t] = ax[x*N+t]*ay_local0
                                      + bx[x*N+t]*by_local0;
            output[N*(NS*(yy+1)+x)+t] = ax[x*N+t]*ay_local1
                                      + bx[x*N+t]*by_local1;
        }
    }
}
```

Y-dependent loads reused x times in x-loop

X-dependent loads used twice

Tile with explicit register use (“unroll-and-jam”)

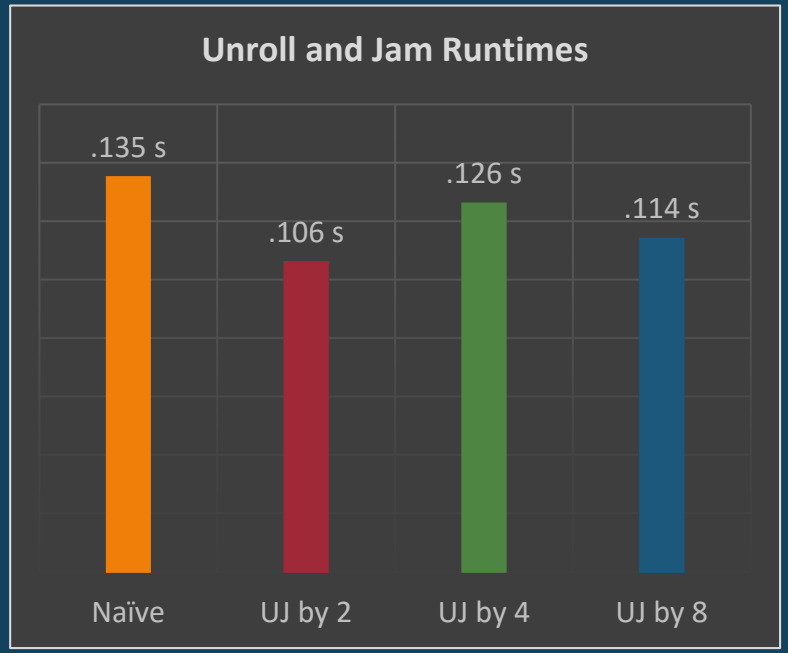
Kepler SM has 65,536 4B registers = 262 KB of near-core memory available as registers.

>2.5X more than read-only and L1 caches combined.

```
__global__ void unroll_and_jam_by2_registers(…)
{
    // Encourage these to be placed in registers
    double ay_local0, by_local0, ay_local1, by_local1;
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    #pragma unroll 1
    for( int yy = 0; yy < NS; yy += 2 )
    {   ay_local0 = ay[(yy+0)*N+t];
        by_local0 = by[(yy+0)*N+t];
        ay_local1 = ay[(yy+1)*N+t];
        by_local1 = by[(yy+1)*N+t];
        # pragma unroll 1
        for( int x = 0; x < NS; x++ )
        {
            output[N*(NS*(yy+0)+x)+t] = ax[x*N+t]*ay_local0
                                      + bx[x*N+t]*by_local0;
            output[N*(NS*(yy+1)+x)+t] = ax[x*N+t]*ay_local1
                                      + bx[x*N+t]*by_local1;
        }
    }
}
```

Y-dependent loads reused x times in x-loop

X-dependent loads used twice



In practice I like this approach.

At 50% occupancy you can use up to 64 registers (32 DP values) for tiling. Unrolling by 2 or 4 is not too annoying for a few performance-limiting kernels.

…but don’t do it for all your kernels.
“Premature optimization is the root of all evil”

Cooperative pattern

Each grid point handled by ~~a single thread~~ a warp.

```
__global__ void warp_team(…)
{
    int warpid = ( threadIdx.x + blockIdx.x*blockDim.x )/32;
    int laneid = threadIdx.x%32;
    int t = warpid;
    #pragma unroll 1
    for( int y = laneid; y < NS; y += 32 )
    {
        double ayy = ay[NS*t+y];
        double byy = by[NS*t+y];
        # pragma unroll 1
        for( int x = 0; x < NS; x++ )
            output[NS*NS*t+NS*x+y] = ax[NS*t+x]*ayy + bx[NS*t+x]*byy;
    }
}
```

Cooperative pattern

Each grid point handled by ~~a single thread~~ a warp.

```
__global__ void warp_team(…)
{
    int warpid = ( threadIdx.x + blockIdx.x*blockDim.x )/32;
    int laneid = threadIdx.x%32;
    int t = warpid;
    #pragma unroll 1
    for( int y = laneid; y < NS; y += 32 )
    {
        double ayy = ay[NS*t+y];
        double byy = by[NS*t+y];
        # pragma unroll 1
        for( int x = 0; x < NS; x++ )
            output[NS*NS*t+NS*x+y] = ax[NS*t+x]*ayy + bx[NS*t+x]*byy;
    }
}
```

Y-dependent loads are coalesced.

Cooperative pattern

Each grid point handled by ~~a single thread~~ a warp.

```
__global__ void warp_team(…)
{
    int warpid = ( threadIdx.x + blockIdx.x*blockDim.x )/32;
    int laneid = threadIdx.x%32;
    int t = warpid;
    #pragma unroll 1
    for( int y = laneid; y < NS; y += 32 )
    {
        double ayy = ay[NS*t+y];
        double byy = by[NS*t+y];
        # pragma unroll 1
        for( int x = 0; x < NS; x++ )
            output[NS*NS*t+NS*x+y] = ax[NS*t+x]*ayy + bx[NS*t+x]*byy;
    }
}
```

Y-dependent loads are coalesced.

X-dependent loads broadcast a value across the warp. Uncoalesced.

X-loads are uncoalesced…BUT next x-iteration accesses next contiguous location in memory…
AND **effective cache per grid point is now 32X higher**…perhaps the next x-load will hit?

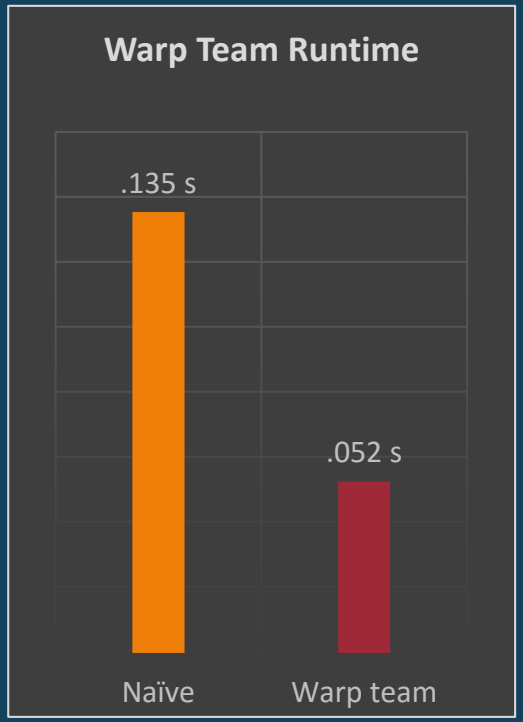
Cooperative pattern is fastest!

Each grid point handled by ~~a single thread~~ a warp.

```
__global__ void warp_team(…)
{
    int warpid = ( threadIdx.x + blockIdx.x*blockDim.x )/32;
    int laneid = threadIdx.x%32;
    int t = warpid;
    #pragma unroll 1
    for( int y = laneid; y < NS; y += 32 )
    {
        double ayy = ay[NS*t+y];
        double byy = by[NS*t+y];
        # pragma unroll 1
        for( int x = 0; x < NS; x++ )
            output[NS*NS*t+NS*x+y] = ax[NS*t+x]*ayy + bx[NS*t+x]*byy;
    }
}
```

Y-dependent loads are coalesced.

X-dependent loads broadcast a value across the warp. Uncoalesced.



X-loads are uncoalesced…BUT next x-iteration accesses next contiguous location in memory…
AND **effective cache per grid point is now 32X higher**…perhaps the next x-load will hit?

Nvprof confirms: high hit rates => fast kernel!!

```
nc_cache_global_hit_rate = 95.39%, tex_cache_hit_rate = 95.39%
```

Downside to cooperative: need different memory layout.

Kernel with each thread handling a grid point

```
__global__ void naive(…)
{
    int t = threadIdx.x + blockIdx.x*blockDim.x;
    #pragma unroll 1
    for( int y = 0; y < NS; y++ )
        # pragma unroll 1
        for( int x = 0; x < NS; x++ )
            output[N*(NS*y+x)+t] =
                ax[x*N+t]*ay[y*N+t] +
                bx[x*N+t]*by[y*N+t];
}
```

Grid point index `t` is fast index for `output, ax, ay, bx, by`.

Consecutive threads handle consecutive grid points => coalesced access.

Corresponds to `Kokkos::LayoutLeft`

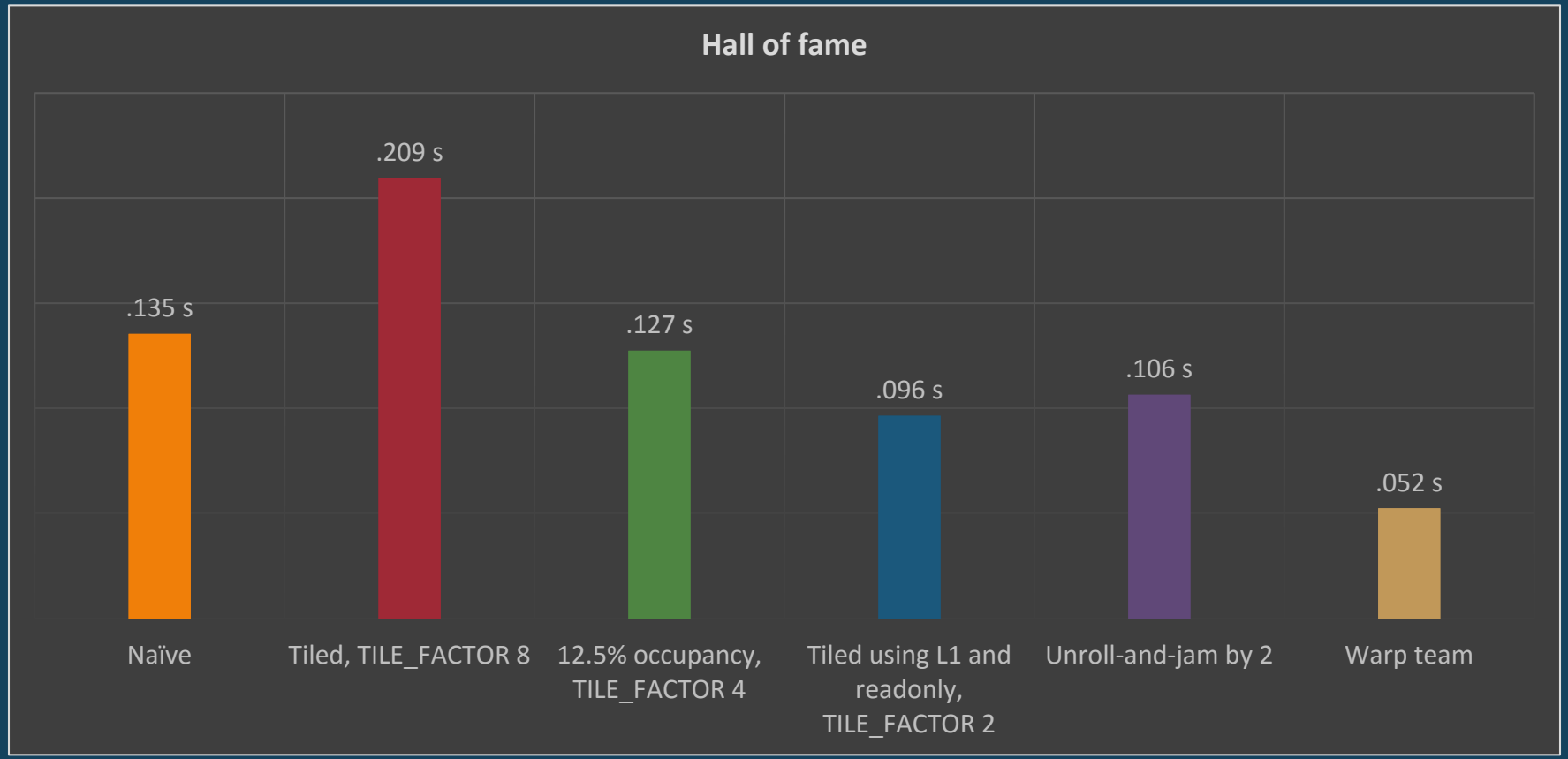
Cooperative kernel

```
__global__ void warp_team(…)
{
    int warpid = (threadIdx.x + blockIdx.x*blockDim.x)/32;
    int laneid = threadIdx.x%32;
    int t = warpid;
    #pragma unroll 1
    for( int y = laneid; y < Ns; y += 32 )
    {
        double ayy = ay[NS*t+y];
        double byy = by[NS*t+y];
        # pragma unroll 1
        for( int x = 0; x < NS; x++ )
            output[NS*NS*t+NS*x+y] =
                ax[NS*t+x]*ayy +
                bx[NS*t+x]*byy;
    }
}
```

Each warp handles one grid point.

Fast index must be species index `x` or `y` (they are symmetric) for spatially local accesses by warps.

Corresponds to `Kokkos::LayoutRight`



Questions?