Kokkos: Performance, Portability and Productivity

H. Carter Edwards, Christian R Trott, Daniel Sunderland

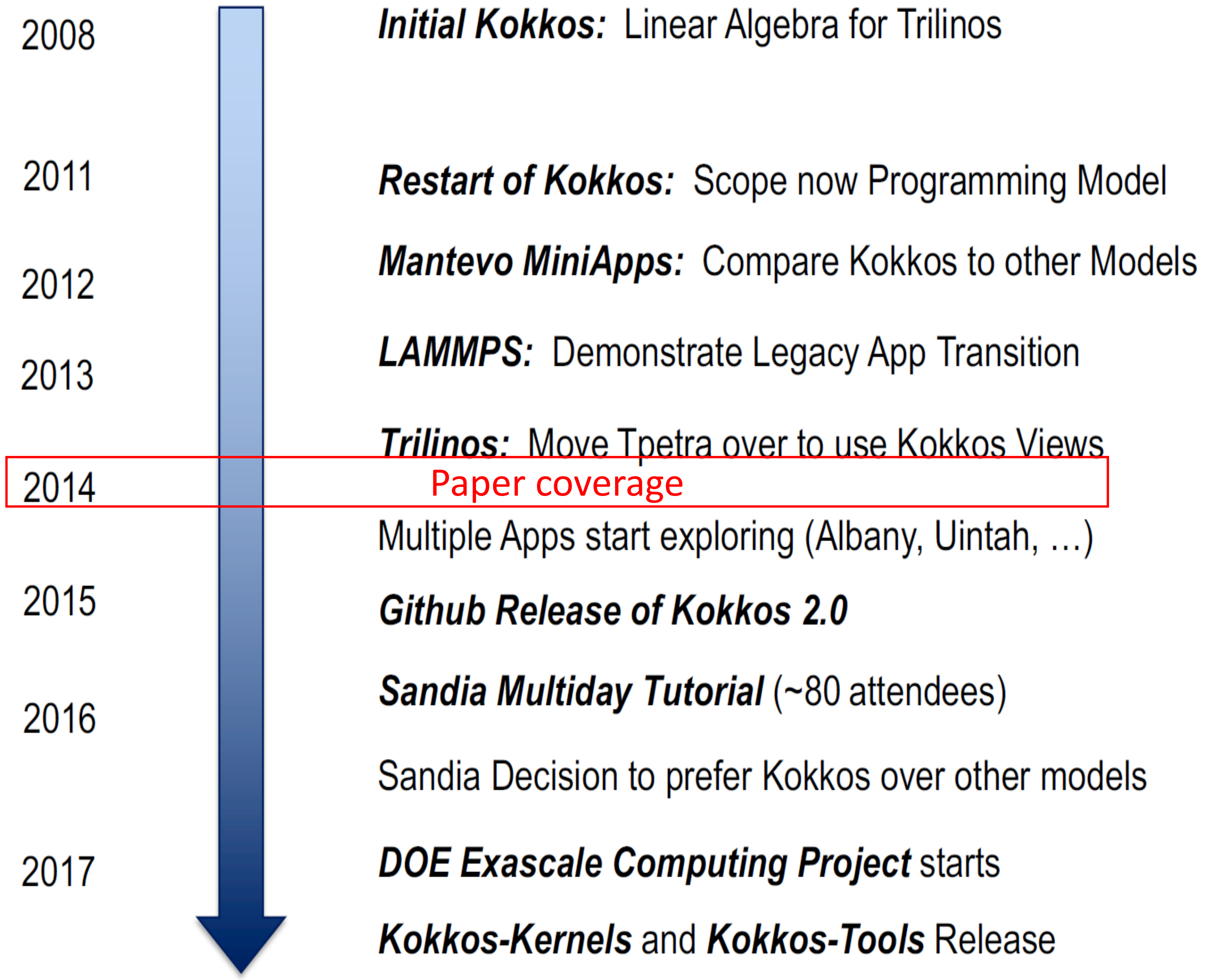
Sandia National Laboratories

Slides prepared by Vikram Sharma Mailthody for CS598APK course at UIUC.

The paper goes over abstractions, APIs and some unit test results on different applications

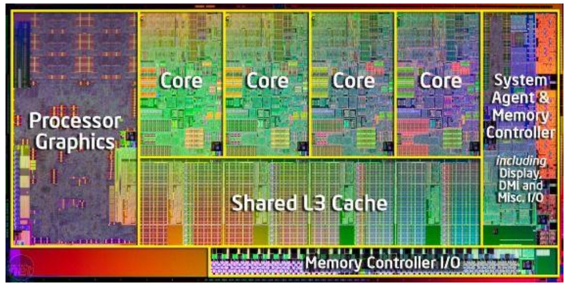
Timeline

Stolen from Kokkos website

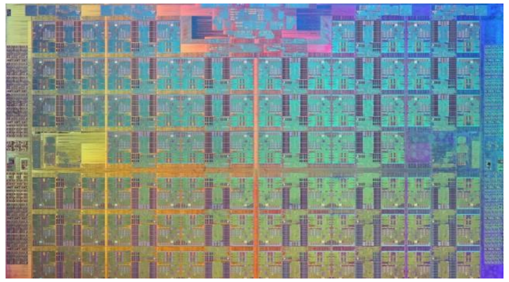


2008	***Initial Kokkos:*** Linear Algebra for Trilinos
2011	***Restart of Kokkos:*** Scope now Programming Model
2012	***Mantevo MiniApps:*** Compare Kokkos to other Models
2013	***LAMMPS:*** Demonstrate Legacy App Transition
	Trilinos: Move Tpetra over to use Kokkos Views
2014	Paper coverage
	Multiple Apps start exploring (Albany, Uintah, …)
2015	***Github Release of Kokkos 2.0***
2016	***Sandia Multiday Tutorial*** (~80 attendees)
	Sandia Decision to prefer Kokkos over other models
2017	***DOE Exascale Computing Project*** starts
	Kokkos-Kernels and ***Kokkos-Tools*** Release

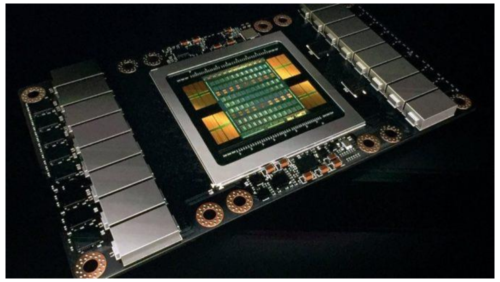
Motivation



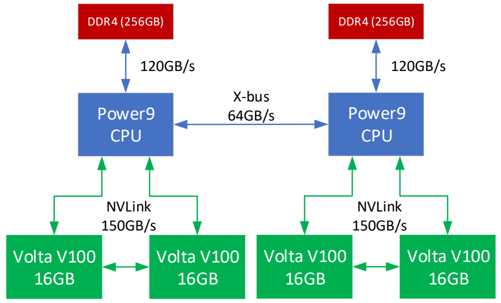
Multicore



Manycore



GPU

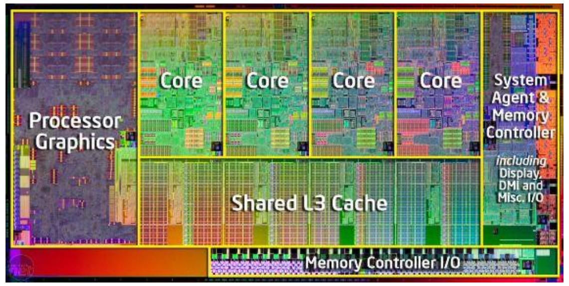


CPU + GPU

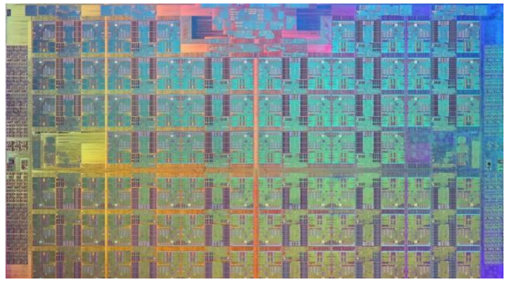
Motivation

- Many core –threads ↑ Memory per thread ↓ and Heterogeneity ↑
- To meet HPC requirement and maintain scalability (load balance, resilience)
  - Need to exploit fine-grain parallelism offered by diverse architecture
  - Shift from MPI-Only model

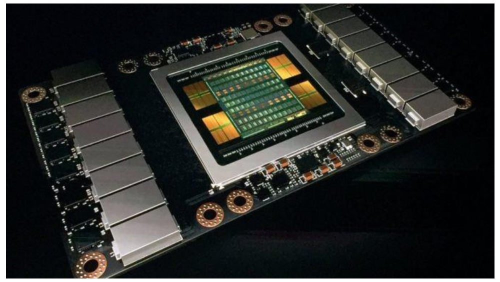
- Major obstacle to performance portability is the diverse and conflicting set of constraints on memory access patterns across devices.



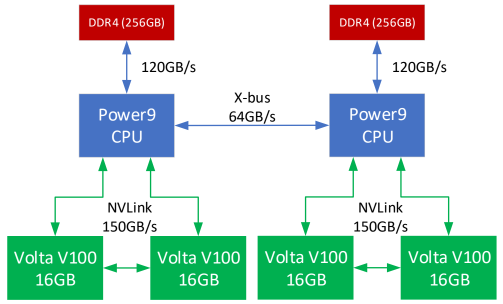
Multicore



Manycore



GPU



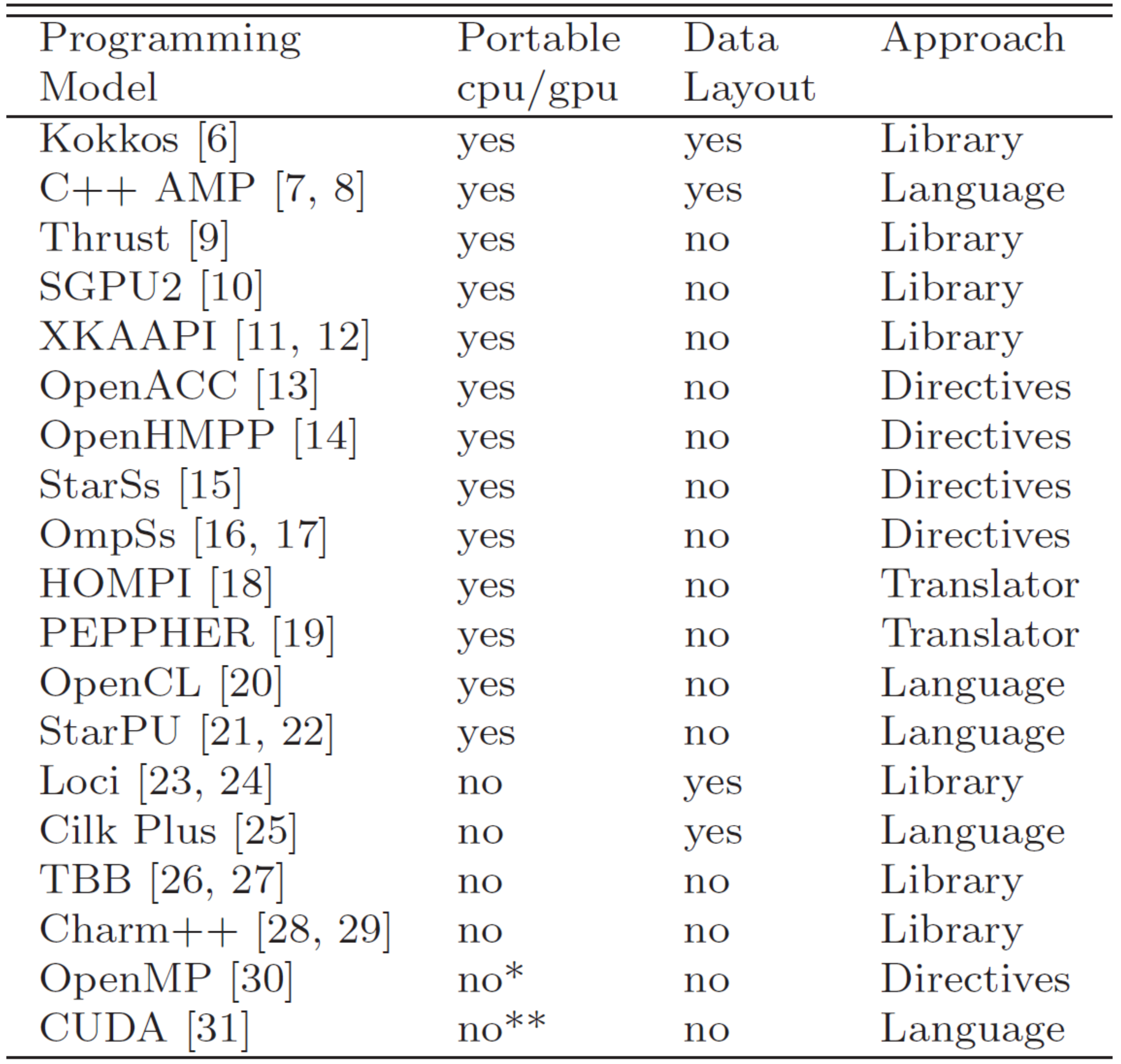
CPU + GPU

Motivation

- Contemporary programming models – Good for manycore parallelism

- But fail to address memory access patterns

Table 1: Programming models for manycore parallelism.



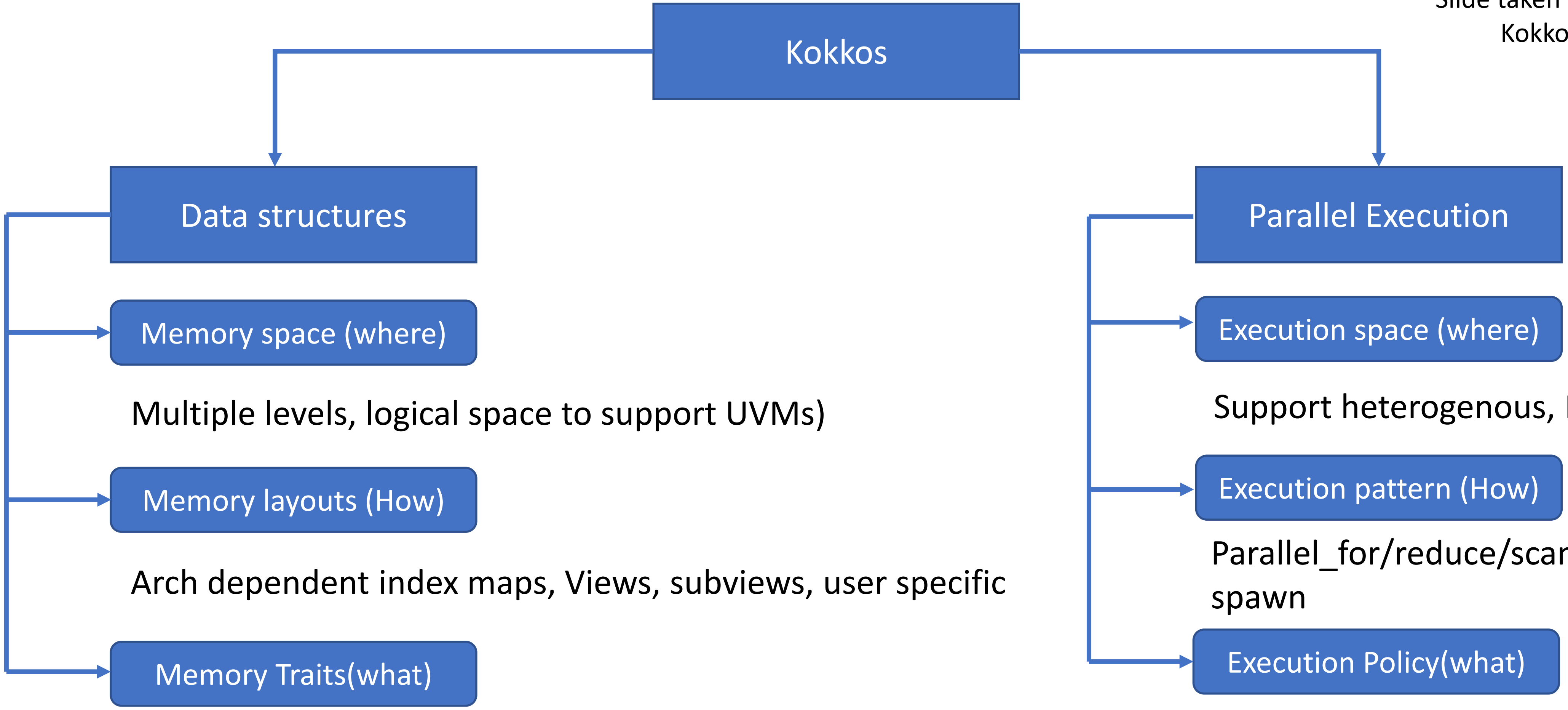
Programming Model	Portable cpu/gpu	Data Layout	Approach
Kokkos [6]	yes	yes	Library
C++ AMP [7, 8]	yes	yes	Language
Thrust [9]	yes	no	Library
SGPU2 [10]	yes	no	Library
XKAAPI [11, 12]	yes	no	Library
OpenACC [13]	yes	no	Directives
OpenHMPP [14]	yes	no	Directives
StarSs [15]	yes	no	Directives
OmpSs [16, 17]	yes	no	Directives
HOMPI [18]	yes	no	Translator
PEPPHER [19]	yes	no	Translator
OpenCL [20]	yes	no	Language
StarPU [21, 22]	yes	no	Language
Loci [23, 24]	no	yes	Library
Cilk Plus [25]	no	yes	Language
TBB [26, 27]	no	no	Library
Charm++ [28, 29]	no	no	Library
OpenMP [30]	no*	no	Directives
CUDA [31]	no**	no	Language

Goals

- To build a C++ library that is efficient
  - Template based library
- Obtain same performance as a variant of code that is written to the specific device
  - Support best data layout strategies specific to architecture
- That provides a programming model with performance portability across diverse devices
  - By unifying abstractions for fine-grain parallelism and memory access patterns
- Targeted to scientific and engineering codes.
  - Supports from sparse linear algebra to broadly usable libraries (mini – apps specific)

Kokkos: Programming model abstractions

Slide taken (and modified) from Kokkos Tutorial deck 2018



Multiple levels, logical space to support UVMs)

Arch dependent index maps, Views, subviews, user specific

Access intent (stream, random)
Access behavior (atomic)
Specialized load primitives - textures

Support heterogenous, N -levels

Parallel_for/reduce/scan, nesting, task spawn

Range, team, etc
Dynamic/static scheduling
Non-persistent scratch-pads, etc

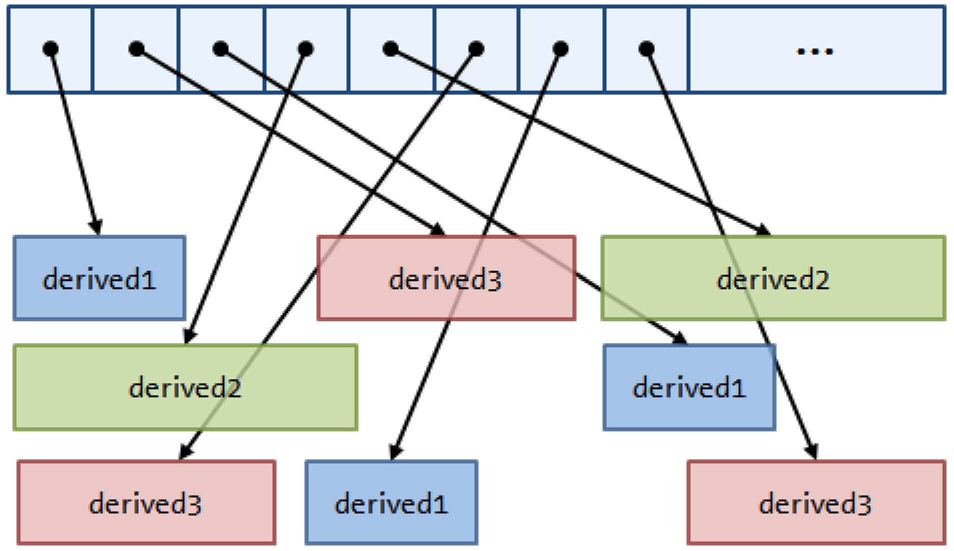
Kokkos: PM abstractions - Fundamentals

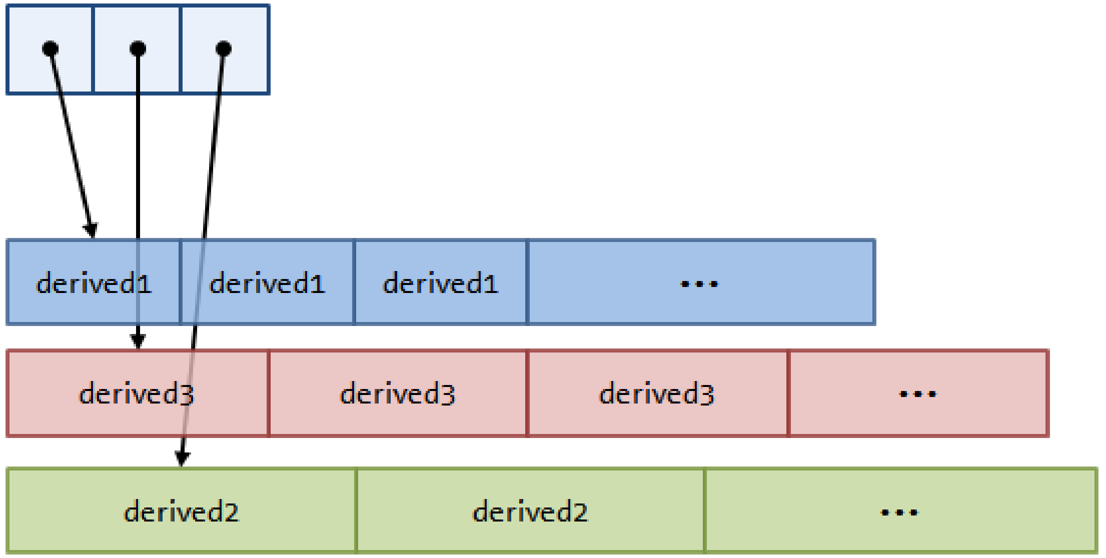
1. Execute computation kernels are in fine-grain data parallel within an **execution spaces.**
2. Computational kernels operate on multidimensional arrays resides in **memory spaces.**
3. Multidimensional arrays supports **polymorphic data layout.**

In other words:

- Execution space is just where the program executes. (Threads in execute space, data in memory space.)
- Execution space has accessibility and performance relationship with memory space. (CPU-GPU – UVM)

Polymorphic data layout





SOA

AOS

Lets say we have a base abstract class with implementation derived1, derived2, and derived3 Then, memory layout of std::vector<base*> shown above.

Adjacent elements are not allocated continuously.
Issues with dynamic polymorphism.

Even though the derived types are unknown, they typically become available during compile time.

Build a container with internal data structure pointing to as many vectors or segments as there are derived class.

Kokkos: Multidimensional arrays

1. Consists of
   1. a set of datums $\{x_i\}$ of the same value type
   2. an index X_s defined by the Cartesian product of integer ranges
   3. A layout X_L - a bijective map between the previous two.

2. A function contains
   1. A sequence of nested loops over dimensions of an array X_s
   2. Access array datums via the layout X_L

To change the memory access pattern – one either needs to change the memory layout or the dimension ordering of X_s

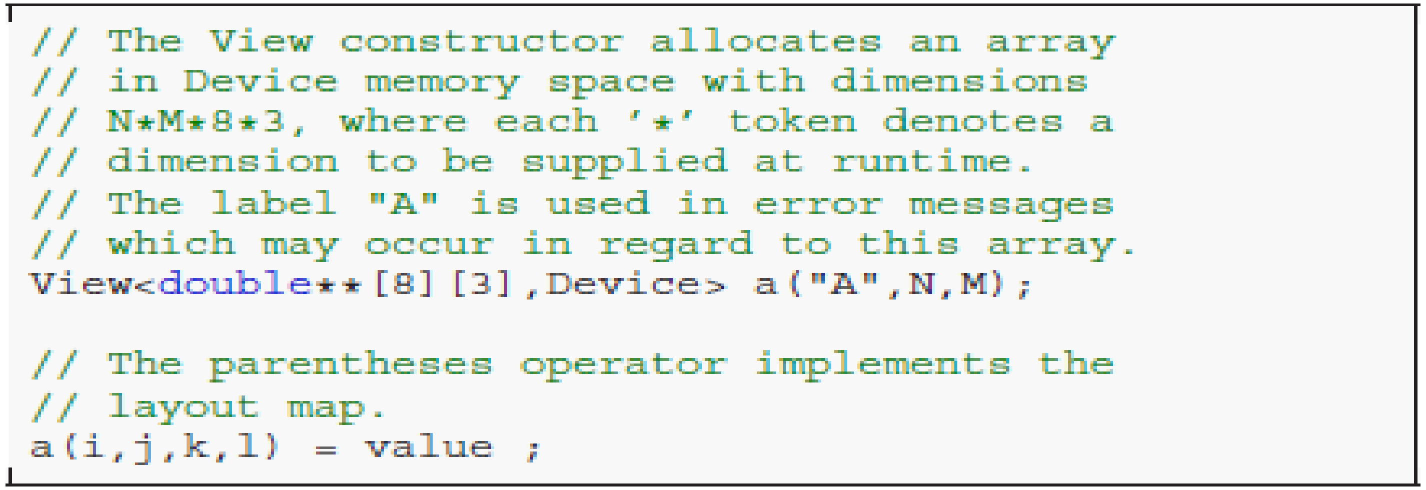
Kokkos array layouts are chosen at compile time!

Using “**Views**”

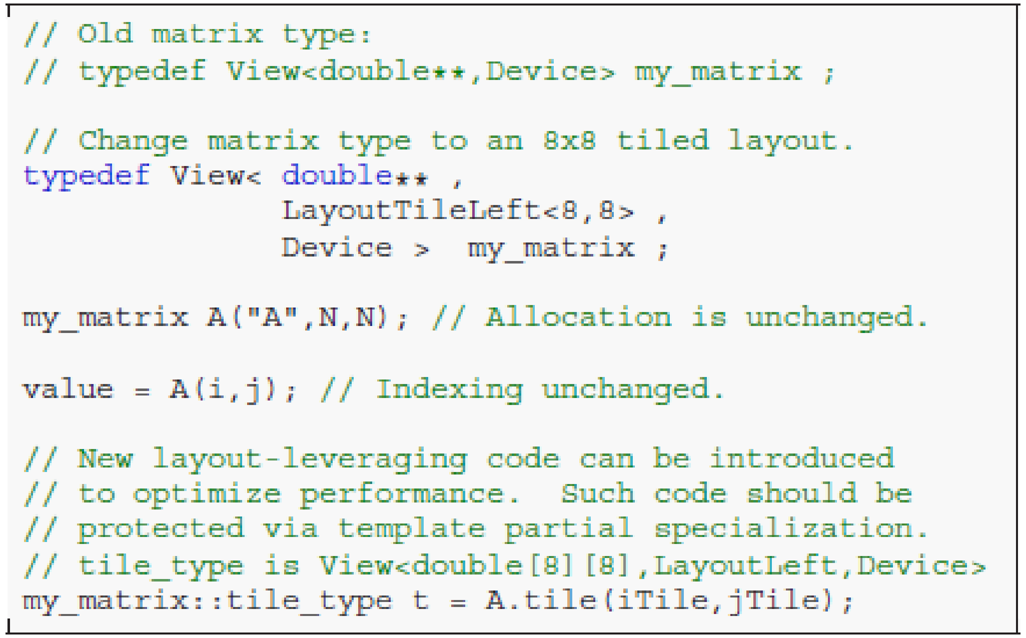
Kokkos: Arrays – Declaration, allocation and access

Views

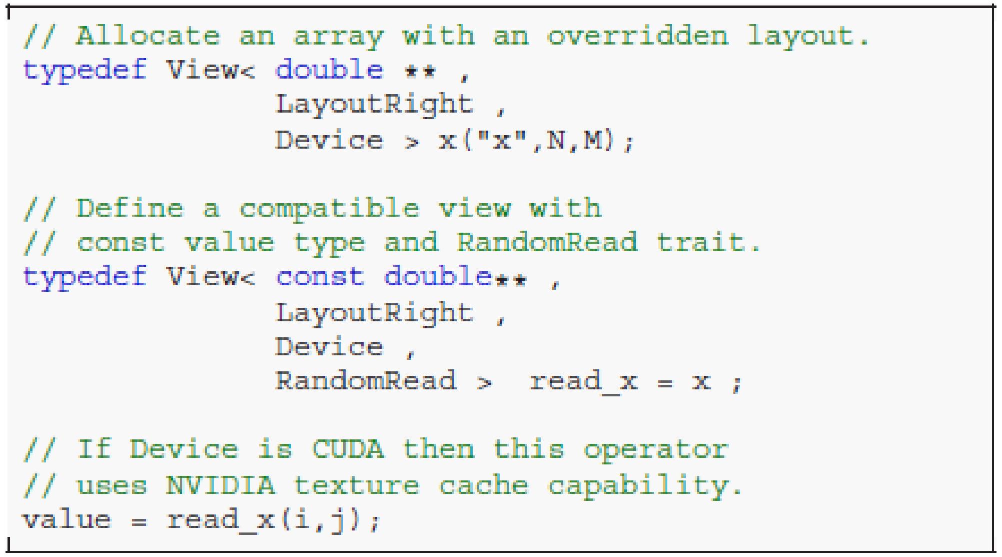
- C++ class with a pointer to array data and metadata
- Semantics similar to std::shared_ptr
- Supports multi-dimensions - scalar, vector, matrix ...
- Size and #dimension is fixed at the compile time
- Copy constructs and assignments are ***shallow***
- Automatic deallocation when last view of the memory is destroyed or reassigned. (ref counting)
- Supports deep_copy. Layout, memory traits and memory space.



```
// The View constructor allocates an array
// in Device memory space with dimensions
// N*M*8*3, where each '*' token denotes a
// dimension to be supplied at runtime.
// The label "A" is used in error messages
// which may occur in regard to this array.
View<double**[8][3],Device> a("A",N,M);

// The parentheses operator implements the
// layout map.
a(i,j,k,l) = value ;
```



```
// Old matrix type:
// typedef View<double**,Device> my_matrix ;

// Change matrix type to an 8x8 tiled layout.
typedef View< double** ,
              LayoutTileLeft<8,8> ,
              Device >  my_matrix ;

my_matrix A("A",N,N); // Allocation is unchanged.

value = A(i,j); // Indexing unchanged.

// New layout-leveraging code can be introduced
// to optimize performance.  Such code should be
// protected via template partial specialization.
// tile_type is View<double[8][8],LayoutLeft,Device>
my_matrix::tile_type t = A.tile(iTile,jTile);
```



```
// Allocate an array with an overridden layout.
typedef View< double ** ,
              LayoutRight ,
              Device > x("x",N,M);

// Define a compatible view with
// const value type and RandomRead trait.
typedef View< const double** ,
              LayoutRight ,
              Device ,
              RandomRead >  read_x = x ;

// If Device is CUDA then this operator
// uses NVIDIA texture cache capability.
value = read_x(i,j);
```

Kokkos: Arrays – Declaration, allocation and access

Other Available memory spaces: HostSpace, CudaSpace, CudaUVMSpace, HostMirror, etc.

Other Available Behavioral traits: StreamingLoad, StreamingStore, RandomRead, RandomWrite

Memory Traits: Atomics, ReadOnce, ReadWrite, RandomAccess

Kokkos: Memory Access – quick look

- Every View has a Layout set at compile-time through a template parameter.
- LayoutRight (row major) and LayoutLeft (column major) are common.
- Views in HostSpace default to LayoutRight and Views in CudaSpace default to LayoutLeft. (caching vs coalesced)
- Layouts are extensible and flexible.
- Kokkos maps parallel work indices and multidimensional array layout for performance portable memory access patterns.
- There is nothing in OpenMP, OpenACC, or OpenCL to manage layouts.
  You'll need multiple versions of code or pay the performance penalty

Kokkos: Parallel Execution

```
for( idx = 0; idx < Max; ++idx){
    sum[idx] = calSum(..data..);
}
```

Kokkos maps work to the cores:

- Each iteration of computation body is a unit of work
- An iteration index identifies particular unit of work
- Iteration range identifies the amount of work

Kokkos maps each iteration indices to cores and then run them in parallel.

Kokkos: Parallel Execution

Computation bodies or kernels are given as functors or function objects.

```
ParallFunctor functor;
Kokkos::parallel_for(numIteration, functor);
```

Work items are assigned to functors one-by-one

```
struct Functor{
    void operator()(const size_t idx) const {…}
}
```

Kokkos: Parallel Execution

Computation bodies or kernels are given as functors or function objects.

```
ParallFunctor functor;
Kokkos::parallel_for(numIteration, functor);
```

Work items are assigned to functors one-by-one

```
struct Functor{
    void operator()(const size_t idx) const {…}
}
```

Kokkos: Parallel Execution

Computation bodies or kernels are given as functors or function objects.

```
Pa
Ko
```

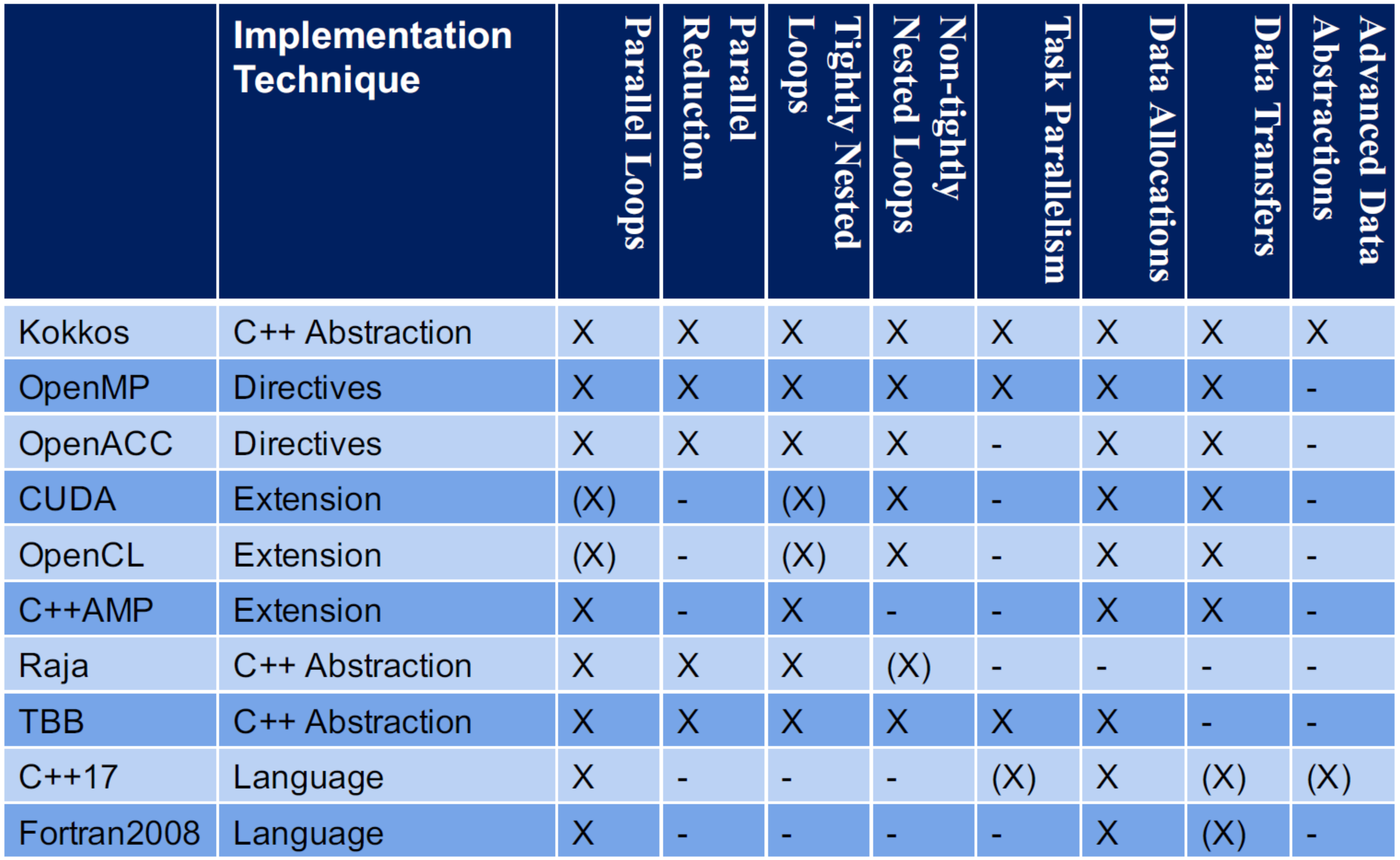
W

Kokkos runtime does not guarantee concurrency and ordering

Parallel functor body must have access to all the data it needs through functor’s data members

```
struct Functor{
    void operator()(const size_t idx) const {…}
}
```

Kokkos: What more?



	Implementation Technique	Parallel Loops	Parallel Reduction	Tightly Nested Loops	Non-tightly Nested Loops	Task Parallelism	Data Allocations	Data Transfers	Advanced Data Abstractions
Kokkos	C++ Abstraction	X	X	X	X	X	X	X	X
OpenMP	Directives	X	X	X	X	X	X	X	-
OpenACC	Directives	X	X	X	X	-	X	X	-
CUDA	Extension	(X)	-	(X)	X	-	X	X	-
OpenCL	Extension	(X)	-	(X)	X	-	X	X	-
C++AMP	Extension	X	-	X	-	-	X	X	-
Raja	C++ Abstraction	X	X	X	(X)	-	-	-	-
TBB	C++ Abstraction	X	X	X	X	X	X	-	-
C++17	Language	X	-	-	-	(X)	X	(X)	(X)
Fortran2008	Language	X	-	-	-	-	X	(X)	-

Taken from the reports.

Kokkos: Parallel Execution - AXPBY

```
void axpby(int n, double* z, double alpha, const double* x,
double beta, const double* y){
       for(int i=0; i<n; i++)
           z[i] = alpha*x[i] + beta*y[i];
}
```

```
void axpby(int n, View<double*> z, double alpha, View<const
double*> x, double beta, View<const double*y){
     parallel_for("AXpBY", n, KOKKOS_LAMDA (const int& i){
                      z[i] = alpha*x[i] + beta*y[i];
     });
}
```

Parallel pattern: for loop

Kokkos: Parallel Execution - AXPBY

```
void axpby(int n, double* z, double alpha, const double* x,
double beta, const double* y){
       for(int i=0; i<n; i++)
           z[i] = alpha*x[i] + beta*y[i];
}
```

```
void axpby(int n, View<double*> z, double alpha, View<const
double*> x, double beta, View<const double*y){
     parallel_for("AXpBY", n, KOKKOS_LAMDA (const int& i){
                z[i] = alpha*x[i] + beta*y[i];
     });
}
```

String Label for debug

Kokkos: Parallel Execution - AXPBY

```
void axpby(int n, double* z, double alpha, const double* x,
double beta, const double* y){
       for(int i=0; i<n; i++)
           z[i] = alpha*x[i] + beta*y[i];
}
```

```
void axpby(int n, View<double*> z, double alpha, View<const
double*> x, double beta, View<const double*y){
     parallel_for("AXpBY", n, KOKKOS_LAMDA (const int& i){
                    z[i] = alpha*x[i] + beta*y[i];
     });
}
```

Exec Policy: do n itrs

Kokkos: Parallel Execution - AXPBY

```
void axpby(int n, double* z, double alpha, const double* x,
double beta, const double* y){
       for(int i=0; i<n; i++)
           z[i] = alpha*x[i] + beta*y[i];
}
```

```
void axpby(int n, View<double*> z, double alpha, View<const
double*> x, double beta, View<const double*y){
     parallel_for("AXpBY", n, KOKKOS_LAMDA (const int& i){
                      z[i] = alpha*x[i] + beta*y[i];
     });
}
```

Its handle: integer index

Kokkos: Parallel Execution - AXPBY

```
void axpby(int n, double* z, double alpha, const double* x,
double beta, const double* y){
      for(int i=0; i<n; i++)
          z[i] = alpha*x[i] + beta*y[i];
}
```

```
void axpby(int n, View<double*> z, double alpha, View<const
double*> x, double beta, View<const double*y){
     parallel_for("AXpBY", n, KOKKOS_LAMDA (const int& i){
                    z[i] = alpha*x[i] + beta*y[i];
     });
}
```

Loop body

Kokkos: Parallel Execution – Dot product

```
double dot(int n, const double* n, const double* y){
       double sum = 0.0;
       for(int i=0;i<n;i++)
              sum += x[i] * y[i]
       return sum;
}


double dot(int n, View<const double*> n, View<const double*> y){
       double x_dot_y= 0.0;
       parallel_reduce("Dot", n, KOKKOS_LAMDA( const int&i, double& sum){
              sum += x[i] * y[i];
       }, x_dot_y);
       return x_dot_y;
}
```

Kokkos: Parallel Execution – Dot product

```
double dot(int n, const double* n, const double* y){
      double sum = 0.0;
      for(int i=0;i<n;i++)
            sum += x[i] * y[i]
      return sum;
}


double dot(int n, View<const double*> n, View<const double*> y){
      double x_dot_y= 0.0;
      parallel_reduce("Dot", n, KOKKOS_LAMDA( const int&i, double& sum){
            sum += x[i] * y[i];
      }, x_dot_y);
      return x_dot_y;
}
```

Parallel pattern: loop with reduction

Kokkos: Parallel Execution – Dot product

```
double dot(int n, const double* n, const double* y){
      double sum = 0.0;
      for(int i=0;i<n;i++)
            sum += x[i] * y[i]
      return sum;
}


double dot(int n, View<const double*> n, View<const double*> y){
      double x_dot_y= 0.0;
      parallel_reduce("Dot", n, KOKKOS_LAMDA( const int&i, double& sum){
            sum += x[i] * y[i];
      }, x_dot_y);
      return x_dot_y;
}
```

Itr index + thread-local reduction Variable.

Kokkos: Parallel Execution – Inner product

```
Kokkos :: parallel_reduce (N,
     KOKKOS_LAMBDA ( const int row , double & valueToUpdate ) {
     double thisRowsSum = 0;
     for (int col = 0; col < M; ++ col ) {
          thisRowsSum += A(row ,col) * x( col );
     }
     valueToUpdate += y( row ) * thisRowsSum ;
}, result );
```

What if we don't have enough rows to saturate the GPU?

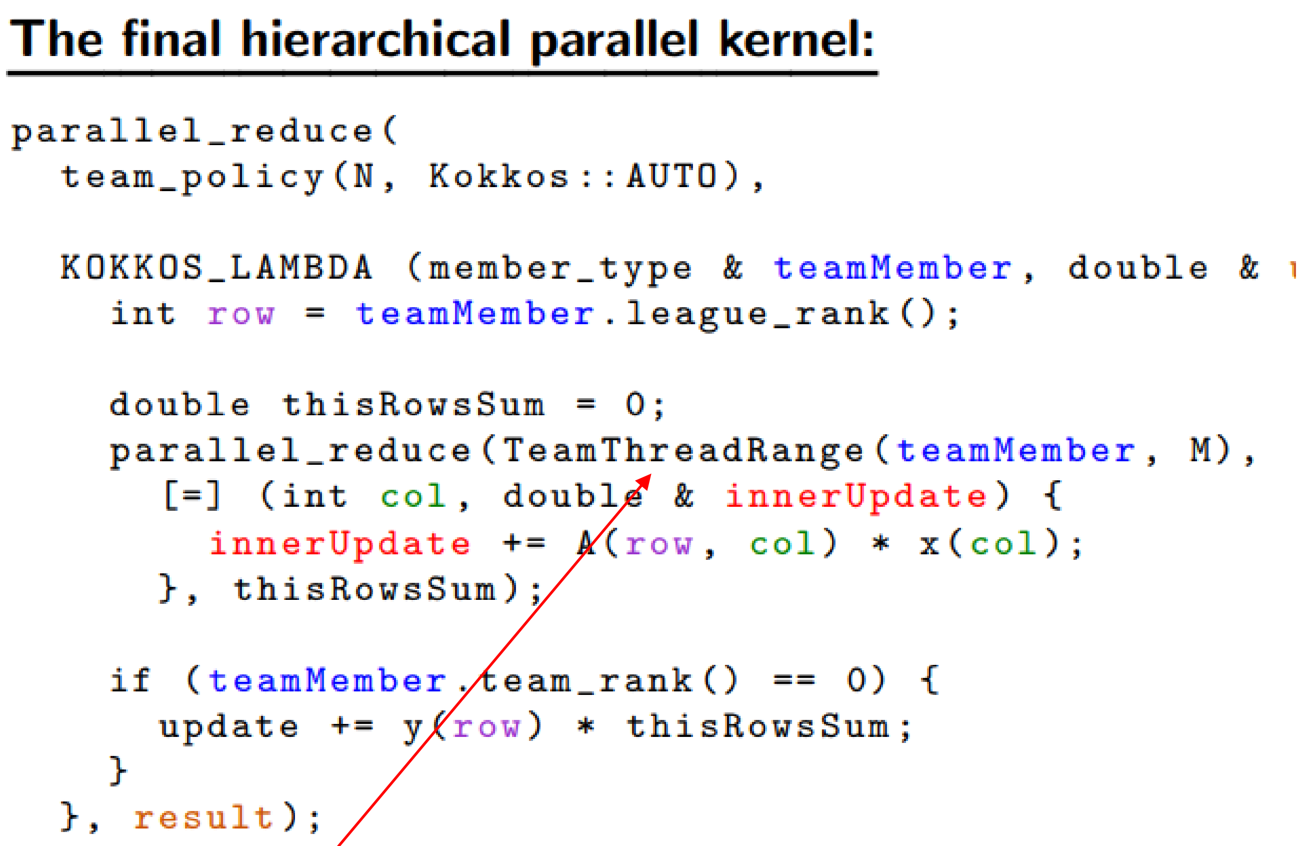
```
1. Atomics
2. Thread Teams
```

Poor performance
Doing each individual row with atomics is like doing scalar integration with atomics.

Kokkos: Parallel Execution – Inner product

High-level strategy:

1. Do one parallel launch of N teams of M threads.
2. Each thread performs one entry in the row.
3. The threads within teams perform a reduction.
4. The thread teams perform a reduction

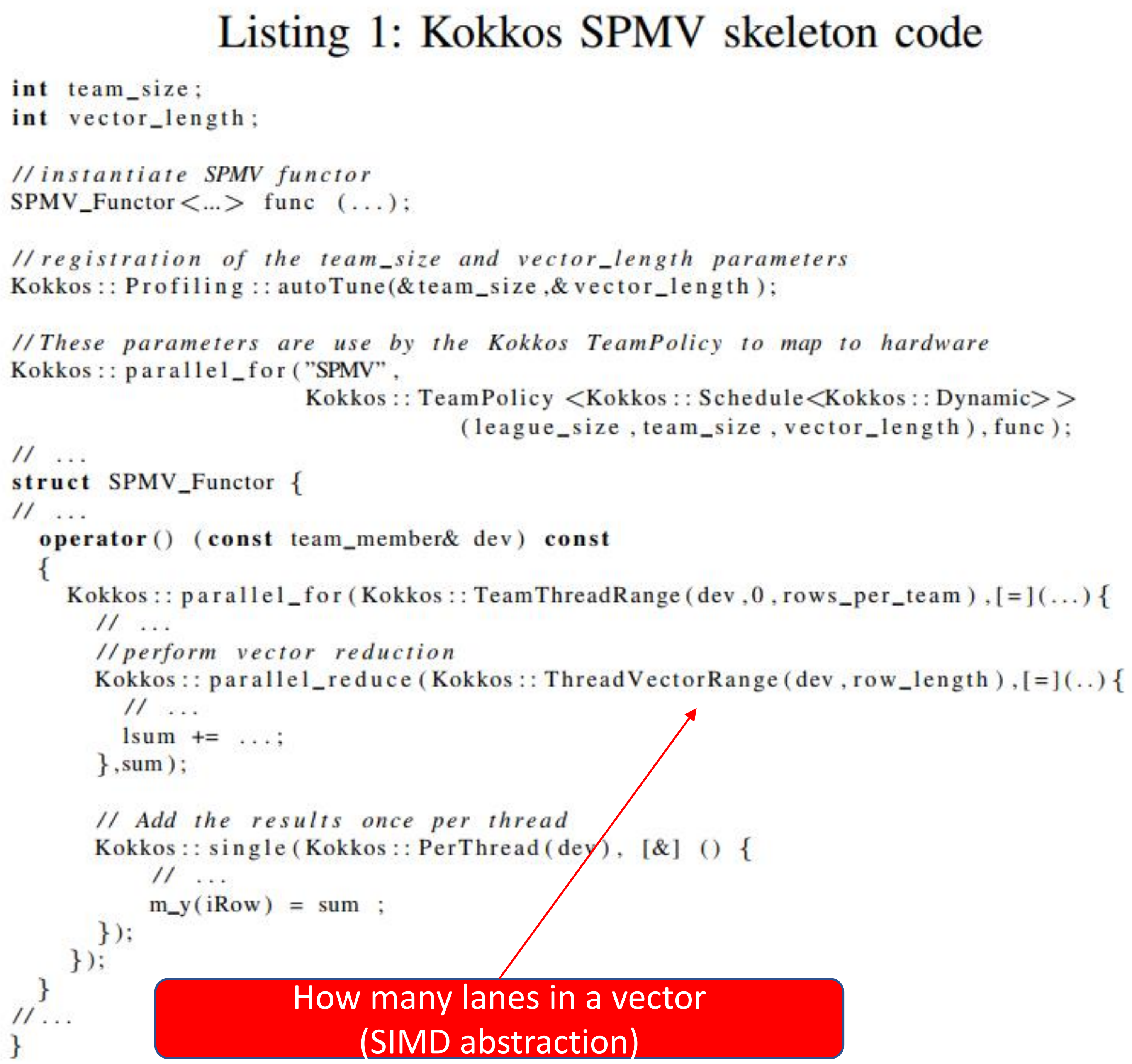


The final hierarchical parallel kernel:

```
parallel_reduce(
  team_policy(N, Kokkos::AUTO),

  KOKKOS_LAMBDA (member_type & teamMember, double &
    int row = teamMember.league_rank();

    double thisRowsSum = 0;
    parallel_reduce(TeamThreadRange(teamMember, M),
      [=] (int col, double & innerUpdate) {
        innerUpdate += A(row, col) * x(col);
      }, thisRowsSum);

    if (teamMember.team_rank() == 0) {
      update += y(row) * thisRowsSum;
    }
  }, result);
```

How many threads in a team

Kokkos: Another example



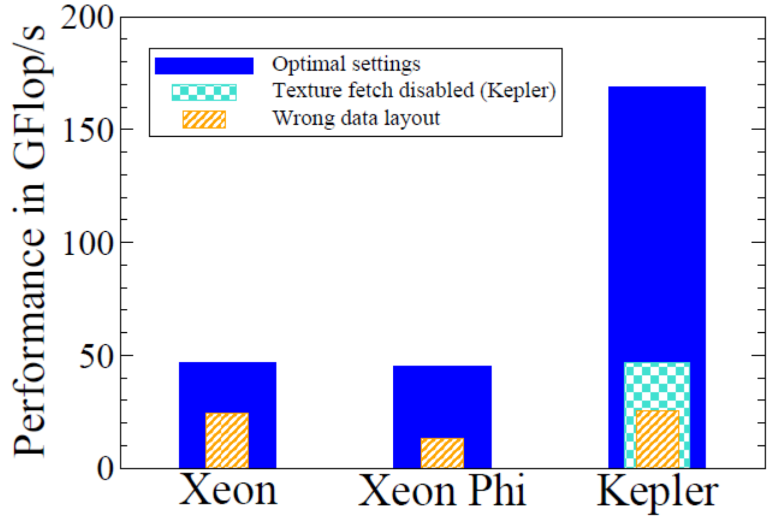
Listing 1: Kokkos SPMV skeleton code

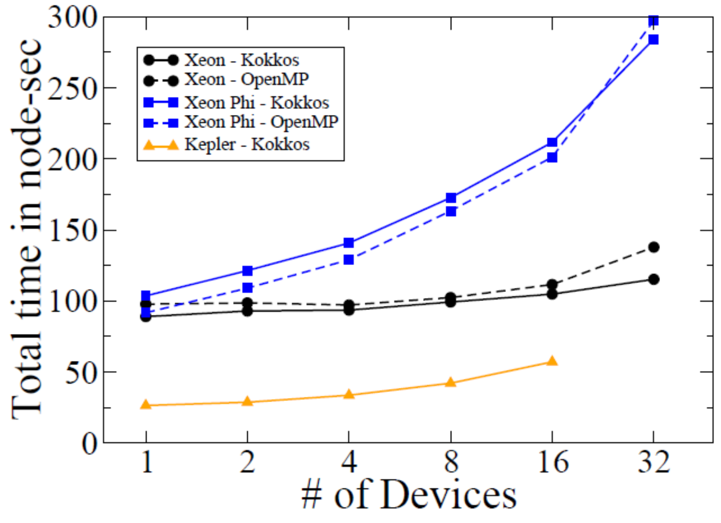
```
int team_size;
int vector_length;

//instantiate SPMV functor
SPMV_Functor<...> func (...);

//registration of the team_size and vector_length parameters
Kokkos::Profiling::autoTune(&team_size,&vector_length);

//These parameters are use by the Kokkos TeamPolicy to map to hardware
Kokkos::parallel_for("SPMV",
                     Kokkos::TeamPolicy <Kokkos::Schedule<Kokkos::Dynamic>>
                              (league_size, team_size, vector_length), func);
// ...
struct SPMV_Functor {
// ...
  operator() (const team_member& dev) const
  {
    Kokkos::parallel_for(Kokkos::TeamThreadRange(dev,0,rows_per_team),[=](...){
      // ...
      //perform vector reduction
      Kokkos::parallel_reduce(Kokkos::ThreadVectorRange(dev,row_length),[=](..){
        // ...
        lsum += ...;
      },sum);

      // Add the results once per thread
      Kokkos::single(Kokkos::PerThread(dev), [&] () {
          // ...
          m_y(iRow) = sum ;
      });
    });
  }
//...
}
```

How many lanes in a vector
(SIMD abstraction)

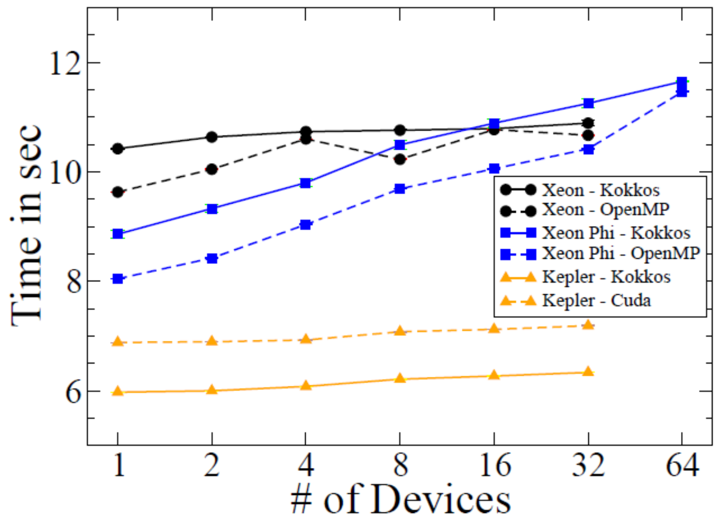
Kokkos: Evaluation

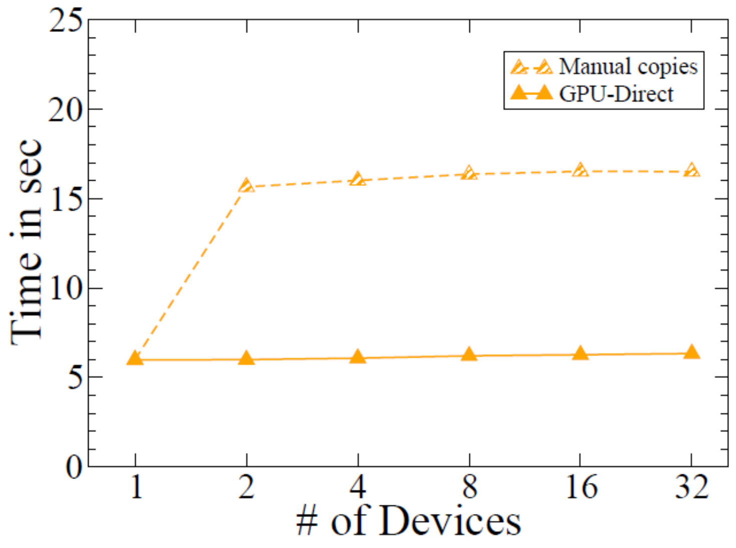
miniMD



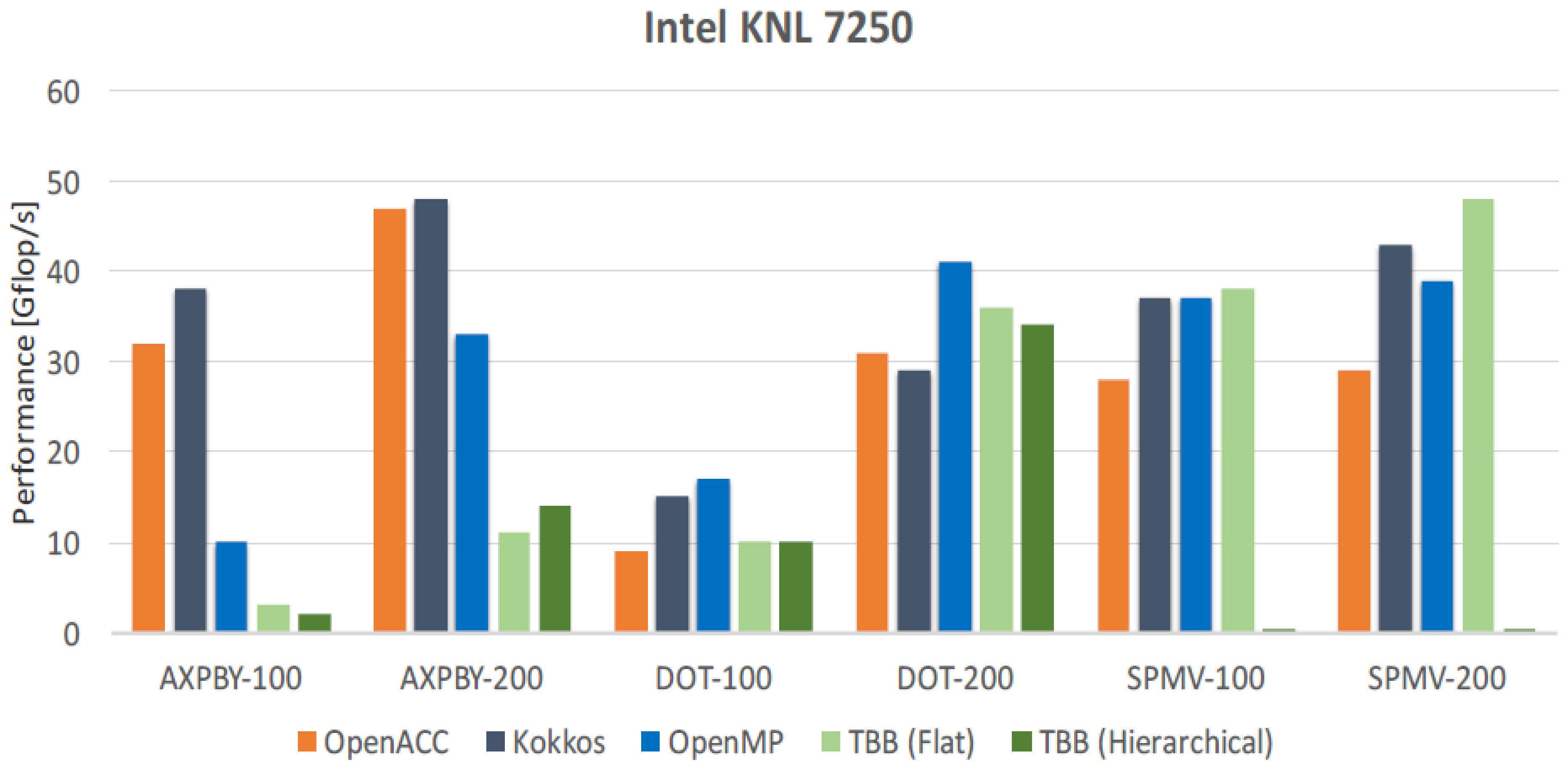


miniFE

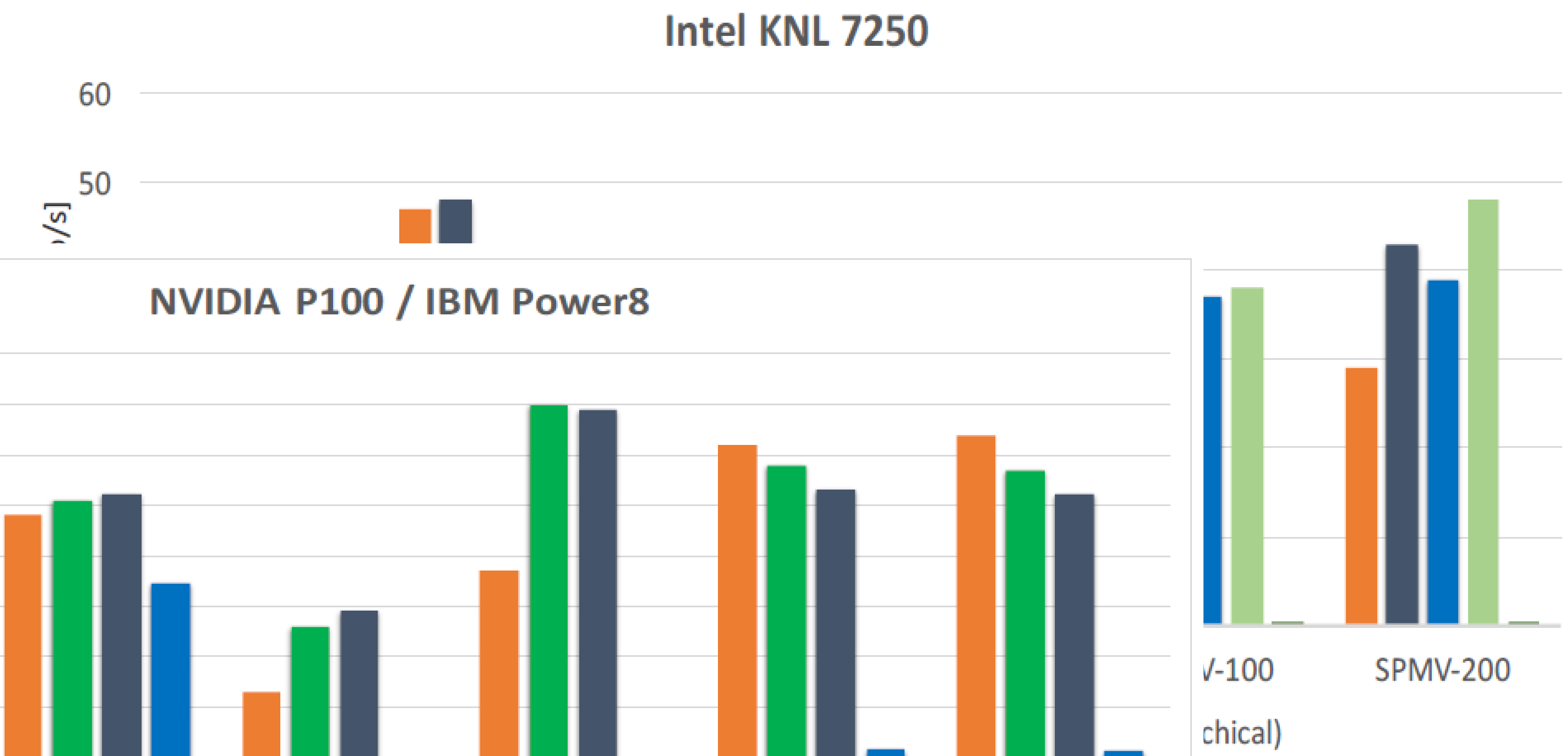


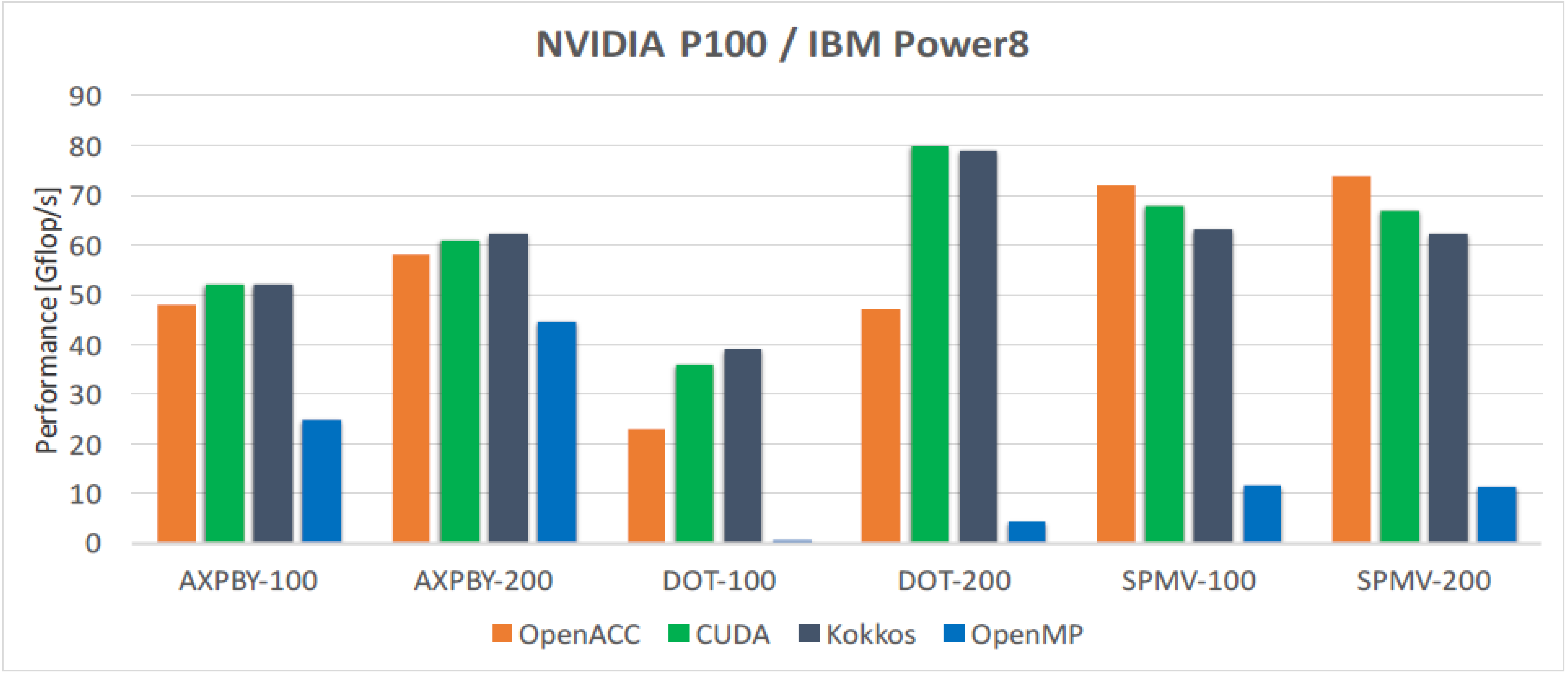


Kokkos: Evaluation

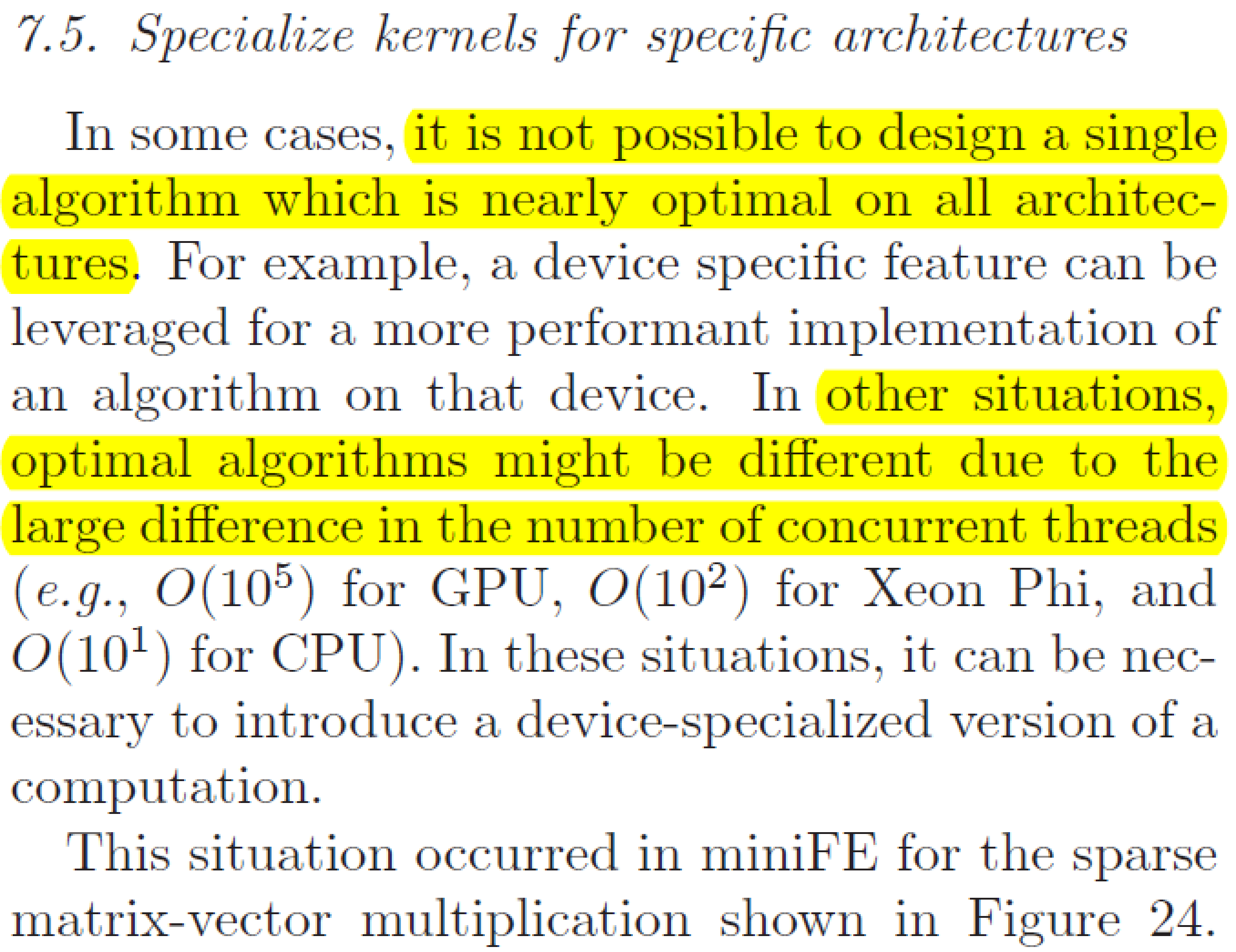


Kokkos: Evaluation





Kokkos: Shh…! Secret…



7.5. Specialize kernels for specific architectures

In some cases, it is not possible to design a single algorithm which is nearly optimal on all architectures. For example, a device specific feature can be leveraged for a more performant implementation of an algorithm on that device. In other situations, optimal algorithms might be different due to the large difference in the number of concurrent threads (*e.g.*, $O(10^5)$ for GPU, $O(10^2)$ for Xeon Phi, and $O(10^1)$ for CPU). In these situations, it can be necessary to introduce a device-specialized version of a computation.

This situation occurred in miniFE for the sparse matrix-vector multiplication shown in Figure 24.

Kokkos: Pros and cons

- Pros:
  - is a C++ library, not a new language or language extension.
  - It is widely used and backbone of Trilinos – Big package for multi-package apps
  - supports clear, concise, thread-scalable parallel patterns.
  - lets you write algorithms once and run on many architectures
  - minimizes the amount of architecture-specific implementation details users must know. (if you need perf – better understand the system)
  - solves the data layout problem by using multi-dimensional arrays with architecture-dependent layouts
- Cons:
  - Rewrite of complete application, need to worry about abstraction, arch, etc
  - kokkos kernels are used by scientist and they don't deal with how to write kokkos kernels! (that's the complexity!)
  - Not a compile time transformation
  - Its TEMPLATE – unreadable.
  - Surprise – some application require to device specific optimization using Kokkos to get performance… Ex: MiniFE.

More details/References

- https://www.sciencedirect.com/science/article/pii/S0743731514001257
- https://github.com/kokkos/kokkos/wiki/The-Kokkos-Programming-Guide (only few contents written)
- https://github.com/kokkos/kokkos
- https://github.com/kokkos/kokkos-tutorials
- http://on-demand.gputechconf.com/gtc/2014/presentations/S4213-kokkos-manycore-device-perf-portability-library-hpc-apps.pdf
- http://extremecomputingtraining.anl.gov/files/2017/08/ATPESC_2017_Track-2_7_8-3_315pm_Edwards-Kokkos.pdf

Kokkos: Questions and answer

1. UVM - https://devblogs.nvidia.com/unified-memory-cuda-beginners/
2. Views: https://github.com/kokkos/kokkos/wiki/View
3. Nested parallelism : https://github.com/kokkos/kokkos/wiki/HierarchicalParallelism
4. Compilation: https://github.com/kokkos/kokkos/wiki/Compiling
5. Sorry – Kokkos View is a potentially reference counted multi dimensional array with **compile time** layouts and memory space. (not runtime as I mentioned)