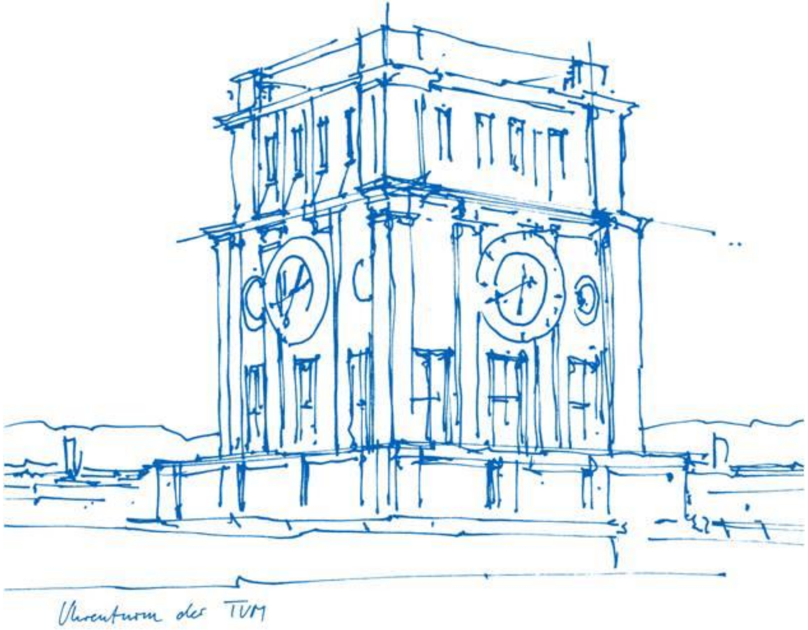
Kokkos: Portable Performance

Alex Hocks

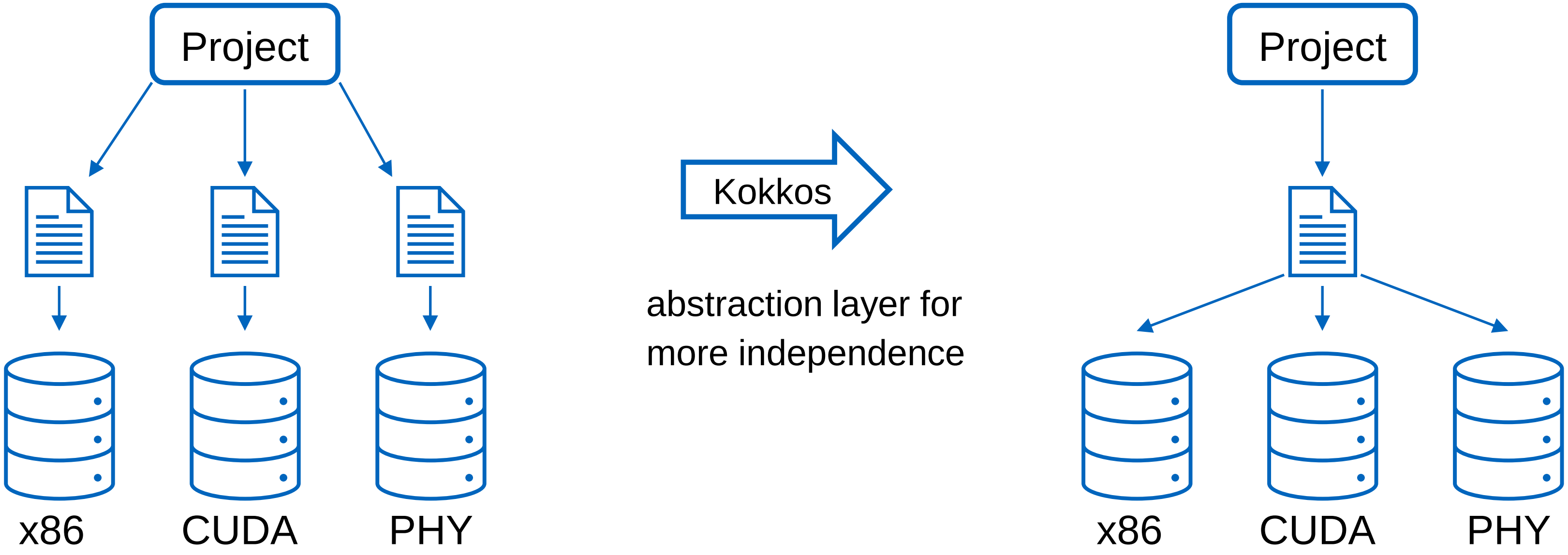
Technical University of Munich

TUM School of Computation, Information and Technology

Garching, 5th July 2022



Introduction



Core Programming Model

Abstractions for:

Execution:

- Execution Spaces
- Execution Policies
- Execution Patterns

Memory:

- Memory Spaces
- Memory Layouts
- Memory Traits

Core Programming Model

Execution Spaces

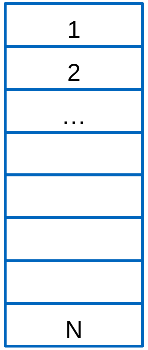
- Execution environment linked to device
- Queue for operations
- Instantiable (defaults)

E.g. Kokkos::Cuda, Kokkos::Thread

Core Programming Model

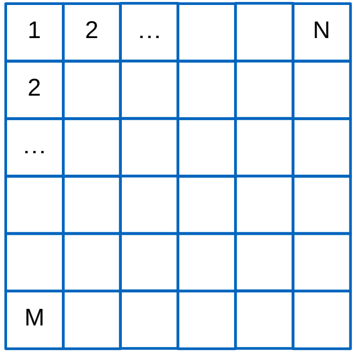
Execution Policies

RangePolicy



RangePolicy(0, N)

MDRangePolicy

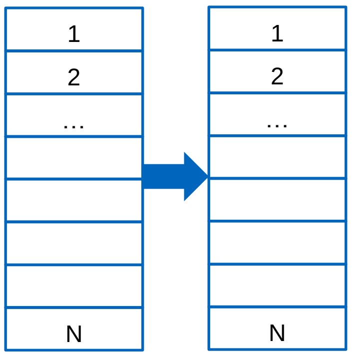


MDRangePolicy<Rank<2>>
({{0,0}}, {N,M}})

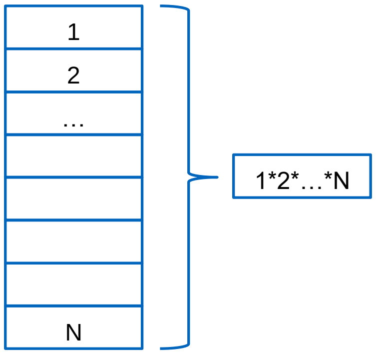
Core Programming Model

Execution Patterns

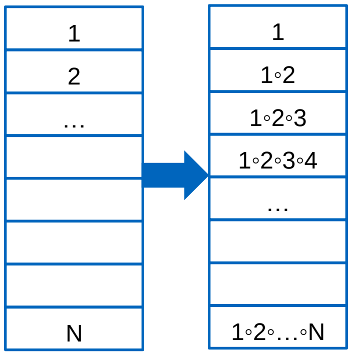
Parallel_for



Parallel_reduce



Parallel_scan



Core Programming Model

Memory Spaces

Defines:

- Memory Location
- Access Restrictions

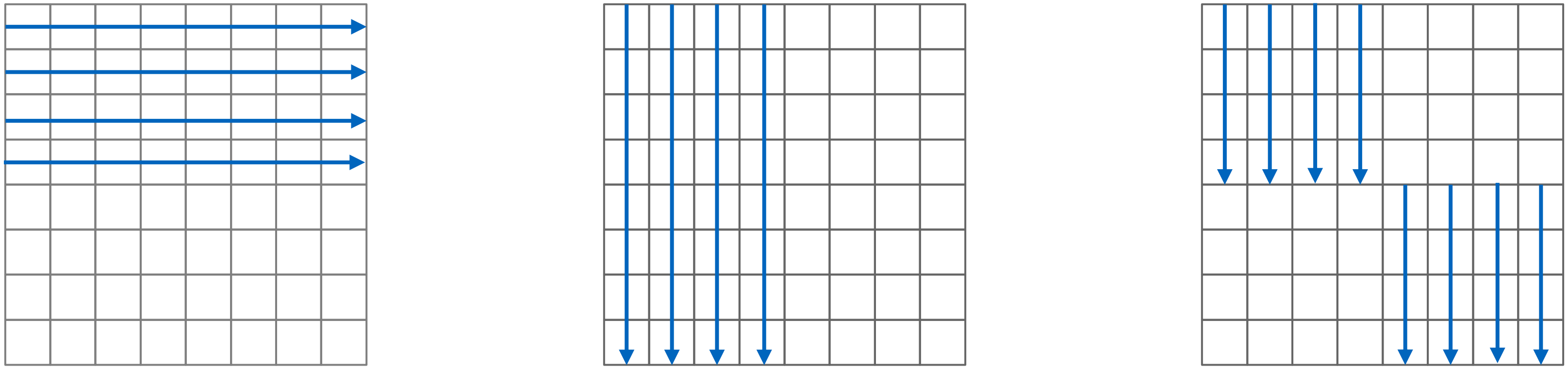
E.g.

- CudaSpace, CudaUVMSpace, CudaHostPinnedSpace (GPU)
- CudaUVMSpace, HostSpace (CPU)

Core Programming Model

Memory Layouts

- Defines indexing method



Core Programming Model

Memory Traits

- Additional behavior when accessing memory

E.g.

- Atomic
- Unmanaged
- …

Core Programming Model

Data Structures

View

- Primary Data structure

ScatterView

- Allows parallel write access to data
- Prevents race conditions
- Concepts:
  - Data replication
  - Atomic Access

Core Programming Model

Examples

```
View<int*, CudaSpace>
    a("a", N), b("b", N), c("c", N);

parallel_for(
    RangePolicy<CUDA>(0, N),
    KOKKOS_LAMBDA(int i) {
        a(i) = b(i) + c(i);
    }
);
```

```
auto v3da =
    View<
            int**[5],
            CudaUVMSpace,
            MemoryTraits<Atomic>
           >
            ("v3da", N, M);
```

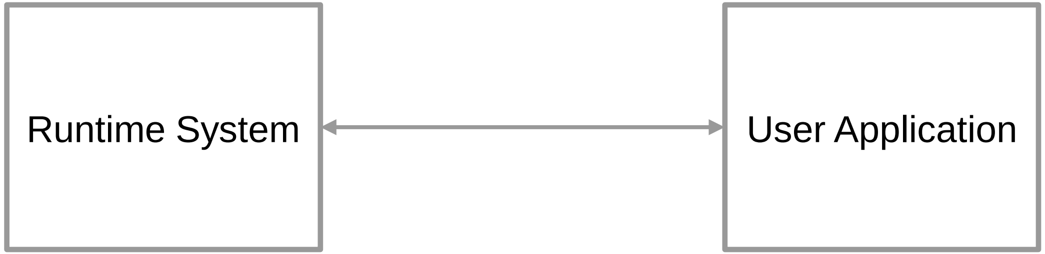
Porting Effort

Steps

1. Find parallelizable sections
2. Refactor into kernels
3. Replace data structures

Uintah

Purpose: Solving gas and fluid dynamics problems



Problem: Many code bases

Uintah

Porting

- Large project → high porting effort
- But: Gradual rewriting (by unmanaged Views)
- Quick initial tests

Uintah

Heat Dissipation Kernel

```
typedef IntVector IV;
for(Iterator itr(low, high); !itr.done(); ++itr) {
    IV c = *itr;
    IV xp = c+IV(1,0,0), xm = c+IV(−1,0,0);
    IV yp = c+IV(0,1,0), ym = c+IV(0,−1,0);
    IV zp = c+IV(0,0,1), zm = c+IV(0,0,−1);
    rhs[c] +=
      ax*(X[xp]*(D[xp]+D[c])*(phi[xp]−phi[c])
          −X[c]*(D[c]+D[xm])*(phi[c]−phi[xm]))
     +ay*(Y[yp]*(D[yp]+D[c])*(phi[yp]−phi[c])
          −Y[c]*(D[c]+D[ym])*(phi[c]−phi[ym]))
     +az*(Z[zp]*(D[zp]+D[c])*(phi[zp]−phi[c])
          −Z[c]*(D[c]+D[zm])*(phi[c]−phi[zm]));
}
```

```
parallel_for(range, [=] (int i, int j, int k) {
    rhs(i, j, k) +=
        ax*(X(i+1,j,k)*(D(i+1,j,k)+D(i,j,k))*(phi(i+1,j,k)−phi(i,j,k))
            −X(i,j,k)*(D(i,j,k)+D(i−1,j,k))*(phi(i,j,k)−phi(i−1,j,k)))
       +ay*(Y(i,j+1,k)*(D(i,j+1,k)+D(i,j,k))*(phi(i,j+1,k)−phi(i,j,k))
            −Y(i,j,k)*(D(i,j,k)+D(i,j−1,k))*(phi(i,j,k)−phi(i,j−1,k)))
       +az*(Z(i,j,k+1)*(D(i,j,k+1)+D(i,j,k))*(phi(i,j,k+1)−phi(i,j,k))
            −Z(i,j,k)*(D(i,j,k)+D(i,j,k−1))*(phi(i,j,k)−phi(i,j,k−1)));
});
```

Conclusion

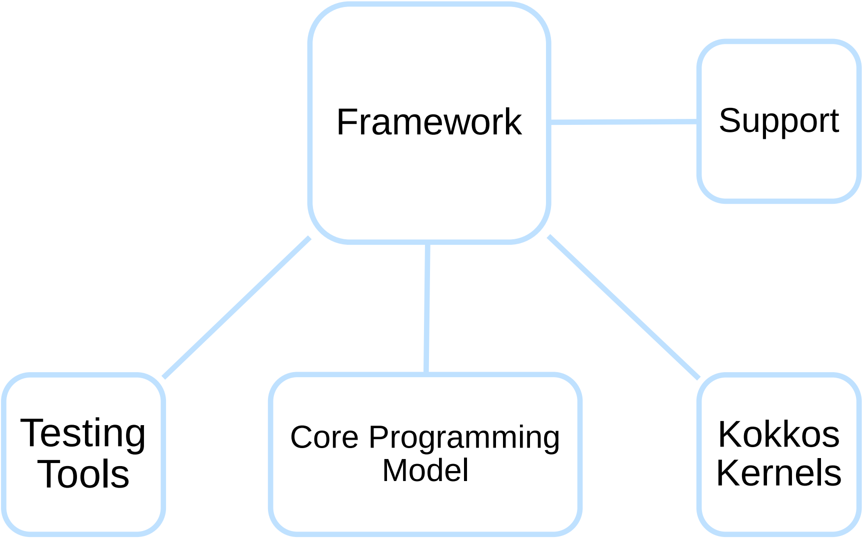
- Acceptable performance
- Vast functionality (data structures, defaults)
- High portability

→ Compelling option for porting with portability as highest priority

Summary

- Explanation of core programming model
  - Execution: Space, Policy, Pattern
  - Memory: Space, Layout, Traits (+ Data Structures)
- Porting Steps
- Uintah

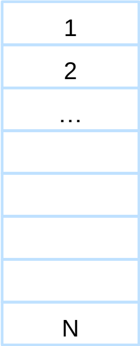
What is Kokkos?



Core Programming Model

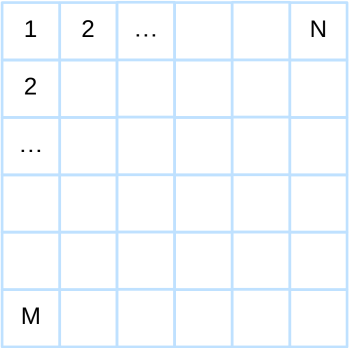
Execution Policies

RangePolicy



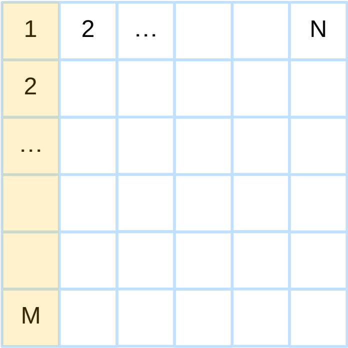

```
RangePolicy(
0, N)
```

MDRangePolicy




```
MDRangePolicy<Ran
k<2>>({{0,0}},
{N,M}})
```

TeamPolicy + TeamThreadRange




```
TeamPolicy<>(N,
AUTO)
TeamThreadRange(team
, M)
```

Porting Effort

Steps

1. Find parallelizable sections
2. Refactor into kernels
3. Replace data structures

```
int a[10]; int b[10]; int
c[10];
for(int i = 0; i < 10; i++) {
    a[i] = b[i] + c[i];
}
```

```
int a[10]; int b[10]; int
c[10];
parallel_for(RangePolicy(0,10),
KOKKOS_LAMBDA(int i) {
    a[i] = b[i] + c[i];
});
```

```
View<int [10]> (“a”);View<int [10]>(“b”);
View<int [10]>(“c”);
parallel_for(RangePolicy(0,10),
KOKKOS_LAMBDA(int i) {
    a[i] = b[i] + c[i];
});
```

Case Studies

Overview

- Uintah (gas and fluid dynamics)
- Comparison Kokkos vs. others
- High Energy Physics
- Deep Neural Networks

Uintah

Porting

- Large project → high porting effort
- But: Gradual rewriting (by unmanaged Views)
- Quick initial tests

Patch size	8^3	16^3	32^3	64^3	128^3
Upwind	4.6	10.0	10.7	12.9	12.7
Van Leer	2.76	4.05	4.04	5.01	6.37

Uintah

Heat Dissipation Kernel

```
parallel_for(range, [=] (int i, int j, pair k_range) {
    auto r = subview(rhs,i,j,ALL());
    /*generate other subviews*/
    parallel_for(krange, [&] (int k) {
      r(k) +=
        ax*(xp(k)*(dp0(k)+d00(k))*(pp0(k)−p00(k))
          −x0(k)*(d00(k)+dm0(k))*(p00(k)−pm0(k)))
       +ay*(yp(k)*(d0p(k)+d00(k))*(p0p(k)−p00(k))
          −y0(k)*(d00(k)+d0m(k))*(p00(k)−p0m(k)))
       +az*(z(k+1)*(d00(k+1)+d00(k))*(p00(k+1)−p00(k))
          −z(k)*(d00(k)+d00(k−1))*(p00(k)−p00(k−1)));
    });
});
```

```
parallel_for(range, [=] (int i, int j, int k) {
    rhs(i, j, k) +=
      ax*(X(i+1,j,k)*(D(i+1,j,k)+D(i,j,k))*(phi(i+1,j,k)−phi(i,j,k))
        −X(i,j,k)*(D(i,j,k)+D(i−1,j,k))*(phi(i,j,k)−phi(i−1,j,k)))
     +ay*(Y(i,j+1,k)*(D(i,j+1,k)+D(i,j,k))*(phi(i,j+1,k)−phi(i,j,k))
        −Y(i,j,k)*(D(i,j,k)+D(i,j−1,k))*(phi(i,j,k)−phi(i,j−1,k)))
     +az*(Z(i,j,k+1)*(D(i,j,k+1)+D(i,j,k))*(phi(i,j,k+1)−phi(i,j,k))
        −Z(i,j,k)*(D(i,j,k)+D(i,j,k−1))*(phi(i,j,k)−phi(i,j,k−1)));
});
```

		32^3		64^3		128^3	
		ms	x	ms	x	ms	x
Serial Uintah		1.06	1.0	8.04	1.0	64.9	1.0
Kokkos Serial	Naive	0.65	1.6	4.30	1.9	36.1	1.8
	SIMD	0.31	3.4	2.47	3.3	20.2	3.2
Kokkos 4 Threads	Naive	0.17	6.4	1.16	6.9	8.94	7.3
	SIMD	0.08	13	0.58	14	5.27	12
Kokkos 16 Threads	Naive	0.07	16	0.54	15	4.51	14
	SIMD	0.04	24	0.31	26	2.54	25
Kokkos 32 Threads	Naive	0.04	29	0.28	29	3.52	18
	SIMD	0.02	43	0.16	49	3.42	19

Comparison Kokkos vs. others

Alternatives: OpenMP, OpenACC, CUDA and RAJA

Evaluation of these libraries in the categories:

- Code clarity (+code overhead)
- Productivity (necessary time)
- Portability
- Performance

Comparison Kokkos vs. others

Example for OpenACC and OpenMP

```
//using OpenACC

#pragma acc parallel loop
for(unsigned int i = 0; i < nCount; i++)
    //do something in each thread
```

```
//using OpenMP

#pragma omp parallel for
for(int i = 1; i < n; i++)
    c[i] = a[i] + b[i];
```

Comparison Kokkos vs. others

Results

Criterion	OpenMP	OpenACC	CUDA	Kokkos	RAJA
Code clarity	High	High	Low	Medium	Medium
Productivity	High	Medium	Low	Medium	Medium
Portability	Low	Medium	Low	High	High
Performance	High	High	High	High	Medium

High Energy Physics

Focus on GPU capabilities of Kokkos

- Implementation in native CUDA → ported to Kokkos
- Interoperability of Kokkos
- Gradual porting to Kokkos

High Energy Physics

Results

		Kokkos time (relative to CUDA)				CUDA time in µs
	Backend	CUDA	Serial	pThread	OpenMP	
Kernel	Clean	2.83	7.67	1.61	2.26	14.0
	Sim_a	1.17	36.9	11.2	11.5	49.2
	Sim_ct	1.35	12.9	1.66	1.99	15.2
	Copy d→h	1.30	0.02	0.06	0.04	25.3
	Event loop	1.16	7.26	2.61	2.79	3.21

Deep Neural Networks

Kokkos Kernels (Linear Algebra Extension)

- Neural Network represented by matrices (sparse)
- Tests with Kokkos Kernels and LAGraph
  - Speedup by 300x to 500x compared to serial code
  - Smaller → Kokkos faster
  - Larger → LAGraph faster

- Node scaling problems

Summary

- Overview of Kokkos framework
- Explanation of core programming model
- Uintah
  - Vectorization capabilities
  - Basic Performance
- Comparison
  - Performance depends on use case
  - High Portability
- HEP
  - Acceptable GPU performance
  - Interoperable with CUDA code
- DNN → high speedups