A Portable SIMD Primitive in Kokkos for Heterogeneous Architectures

Damodar Sahasrabudhe†, Eric Phipps*, Sivasankaran Rajamanickam*, Martin Berzins†.

†Scientific Computing and Imaging Institute, University of Utah

*Center for Computing Research, Sandia National Laboratories

The work is funded by:

1. Sandia National Laboratories
2. The Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375

Performance Portability

- Wide range of architectures are developed CPUs, GPUs, Many-Core Processors, ARM, FPGAs, memory centric …
- Developing/tuning code for every architecture causes development and maintenance overheads
- Answer: Performance Portability
  - run the user code *without any changes* across diverse architectures
  - Same (or nearly same) performance as architecture-specific code
- OpenMP 4.5, OpenACC, Kokkos, RAJA, OCCA, …
- Kokkos portable construct:

```
Kokkos::parallel_for(…, KOKKOS_LAMBDA(int i){
            …
});
```

Vectorization support in Kokkos

Kokkos uses compiler directives to achieve auto-vectorization

Programmers have to compromise between:

- Rely Kokkos's support with in-built directives
  - Pros: Get a portable code
  - Cons: Vectorization may not be *always* efficient
- Use SIMD primitive (from a third party library)
  - Pros: Efficient vectorization
  - Cons: No CUDA backend, compilation with nvcc fails

    Maintain a separate version: No portability!

Fix: A Portable SIMD primitive with a CUDA backend

What is SIMD Primitive?

- A SIMD primitive is a wrapper around intrinsics
- Speedup by explicit vectorization
- Works across CPUs, readable, maintainable

Explicit Vectorization

```
//user code using intrinsics:
__m512d A, B, C;
C = _mm512_add_pd(A, B)
```

Using SIMD Primitive

```
//simd library:
struct simd{
  __m512d _data;
  inline simd operator+  (const simd &x){
    return _mm512_add_pd ( _data, x._data );
  }
};

//user code: SIMD primitive performs the same operation:
simd A, B, C;
C = A + B;
```

Wrapper around Intel KNL intrinsics

What is SIMD Primitive?

- A SIMD primitive is a wrapper around intrinsics
- Speedup by explicit vectorization
- Works across CPUs, readable, maintainable

nvcc can not compile this code for cuda

Explicit Vectorization

```
//user code using intrinsics:
__m512d A, B, C;
C = _mm512_add_pd(A, B)
```

Using SIMD Primitive

```
//simd library:
struct simd{
  __m512d _data;
  inline simd operator+  (const simd &x){
    return _mm512_add_pd ( _data, x._data );
  }
};

//user code: SIMD primitive performs the same operation:
simd A, B, C;
C = A + B;
```

← Wrapper around Intel KNL intrinsics

Why SIMD Primitive when compiler can auto-vectorize?

```
for (i over mesh $256^3$ cells){          ← Can this loop be vectorized?
    …
    for (j < 4){
        …
        for(k < 3)                        ← 1. Innermost loop gets auto-vectorized
            …
        …
    }
    while (r < t){                        ← 2. Non-countable iterations
        …
    }
    A[i + offset] = …                     ← 3. Assumed dependencies due to offset
    …
}
```

- Better efficiency possible
- Avoids significant code changes needed for auto-vectorization

Some of many SIMD Primitives libraries

- stk::simd and KokkosKernels with Trilinos package.
- Vc vectorization library
- VCL: C++ vector Class Library
- Unified Multi/Many-Core Environment (UME) framework
- Generic SIMD Library

But no CUDA backend!

Requirements of Portable SIMD Primitive

- Portability
- Heterogeneous execution: backend based on the execution space
- Standard math operations and library functions
- Logical Vector Length (LVL)
- Performance and vectorization
  - No overhead against efficiently auto-vectorized code
  - Performance boost against hard to auto-vectorized code
  - No overhead against efficient CUDA code i.e., improving CPU performance must not hamper GPU performance

Portability: Creating a CUDA backend

KNL backend reused from stk::simd

```
struct simd{

  __m512d _d;

  inline simd
  operator+ (const simd &x){
    simd sum;

    sum = _mm512_add_pd(_d, x._d);

    return sum;
  }
};
```

1. simd length number of doubles

2. Each vector lane adds its elements

New CUDA backend

```
struct simd{

 double _d[BLOCK_DIM_X];

  __device__ inline simd
  operator+ (const simd &x){
    simd sum;
    int tx = threadIdx.x;
    sum._d[tx] =_d[tx] + x._d[tx];

    return sum;
  }
};
```

Heterogeneous execution: template meta-programming

- Multiple execution spaces in the same program
- Execution space passed as a template parameter

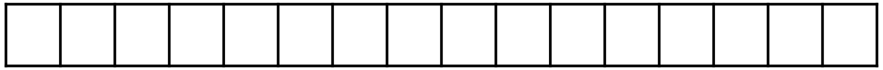
```
simd<double, …, Kokkos::OpenMP> A;
simd<double, …, Kokkos::Cuda> B;
```

Supports execution of tasks on different platforms in a same program.

Logical Vector Length (LVL)

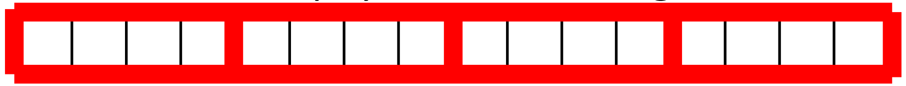
- Passed as a template parameter. Operators iterate over LVL elements. Each vector lane operators on LVL / PVL number of elements
- User code transparent to physical vector length (PVL)
- Arrays can be used as variables: similar to matlab
- Can “unroll and jam” loops automatically: ILP + cache reuse

Scalar:



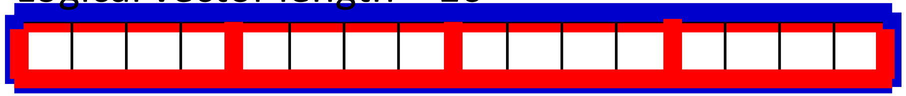
```
for(int i = 0; i < 16; i++)
    c[i] = a[i] + b[i];
```

SIMD Primitive: physical vector length = 4



```
for(int i = 0; i < 4; i++)
    c[i] = a[i] + b[i];
```

Logical vector length = 16



```
c = a + b; //similar to matlab
```

Performance Evaluation

- Case studies:
  - Uintah’s CharOx
  - 2D -Convolution
  - Batched GEMM
  - Ensembled SpMV
- All kernels first written using Kokkos and then the portable SIMD primitive added
- Tested on Intel KNL, Nvidia P100 and Cavium ThunderX2 (ARMv8.1)

Uintah CharOx kernel*

Objective	Baseline	Ideal Speedup
CPU: Vectorization GPU: Find out overhead	CPU: Not vectorized GPU: Ported to cuda	KNL: 8x P100: 1x ARM: 2x

Can cells loop be vectorized?

```
for (i over 32³ patch-cells){
        …
        for (j < 4){
                …
                for(k < 3)
                …
        }
        while (r < t){
                …
        }
        A[i + offset] = …
        …
}
```

- Simulates oxidation of coal in a boiler
- 350 lines of code.
- Complex control flow: triply nested loops, breaks, conditionals
- Array access with offsets
- 300+ iterations for every cell
- Efficient auto-vectorization needs:
  - Rearranging loops
  - Rearranging conditionals
  - Scalar variables to arrays
- #pragma simd generates vgather and gives only 4.3x speedup

*The scalar version was optimized by John Holmen

SP for CharOx Kernel

- No changes in arithmetic operations
- Reduce loop iterations by a factor of the SIMD length
- Cast data structures to SIMD primitive
- Use SIMD conditional operator instead of if else
- Algorithmic change:
  - Newton-Raphson solve runs independently for every cell until convergence
  - With simd type, loop iterates until all cells within simd block converge
- Code change: less than 10% of the kernel. Effort: 2 days

```
const double RHS_v      = RC_RHS_source(i,j,k) * local_RC_scaling_constant * local_weight_scaling_constant; // [kg/s]
const double RHS        = RHS_source(i,j,k) * local_char_scaling_constant * local_weight_scaling_constant;  // [kg/s]

// populate temporary variable vectors
const double delta = 1e-6;

for ( int r = 0; r < reactions_count; r++ ) {
  rh_l_new[r] = old_reaction_rate[r](i,j,k); // [kg/m^3/s]
}
for ( int r = 0; r < reactions_count; r++ ) { // check this
  oxid_mass_frac[r] = species[local_oxidizer_indices[r]](i,j,k); // [mass fraction]
}

for ( int ns = 0; ns < species_count; ns++ ) {
  species_mass_frac[ns] = species[ns](i,j,k); // [mass fraction]
}

const double CO2onCO = 1. / ( 200. * exp( -9000. / ( local_R_cal * p_T ) ) * 44.0 / 28.0 ); // [ kg CO / kg CO2] => [
for ( int r = 0; r < reactions_count; r++ ) {
  if ( local_use_co2co_l[r] ) {
    phi_l[r]  = ( CO2onCO + 1 ) / ( CO2onCO + 0.5 );
    hrxn_l[r] = ( CO2onCO * local_HF_CO2 + local_HF_CO ) / ( 1 + CO2onCO );
  }
  else {
    phi_l[r]  = local_phi_l[r];
    hrxn_l[r] = local__hrxn_l[r];
  }
}

const double Re_p = sqrt( ( CCuVel(i,j,k) - up(i,j,k) ) * ( CCuVel(i,j,k) - up(i,j,k) ) +
                          ( CCvVel(i,j,k) - vp(i,j,k) ) * ( CCvVel(i,j,k) - vp(i,j,k) ) +
                          ( CCwVel(i,j,k) - wp(i,j,k) ) * ( CCwVel(i,j,k) - wp(i,j,k) ) )*
                          p_diam / ( local_dynamic_visc / gas_rho ); // Reynolds number [-]

const double x_org     = (rc + ch) / (rc + ch + local_mass_ash );
const double cg        = local_gasPressure / (local_R * gas_T * 1000.); // [kmoles/m^3] - Gas concentration
```

Not vectorized

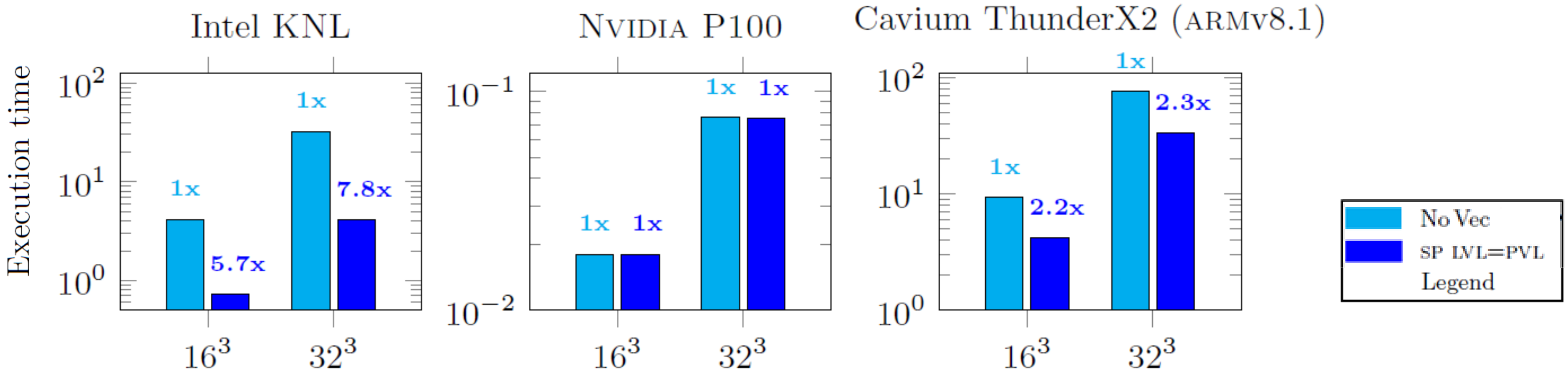
SIMD Primitive

```
const Double RHS_v         = RC_RHS_source(i,j,k) * local_RC_scaling_constant * local_weight_scaling_constant; // [kg/s]
const Double RHS           = RHS_source(i,j,k) * local_char_scaling_constant * local_weight_scaling_constant;  // [kg/s]

// populate temporary variable vectors
const Double delta = 1e-6;

for ( int r = 0; r < reactions_count; r++ ) {
  rh_l_new[r] = old_reaction_rate[r](i,j,k); // [kg/m^3/s]
}
for ( int r = 0; r < reactions_count; r++ ) { // check this
  oxid_mass_frac[r] = species[local_oxidizer_indices[r]](i,j,k); // [mass fraction]
}

for ( int ns = 0; ns < species_count; ns++ ) {
  species_mass_frac[ns] = species[ns](i,j,k); // [mass fraction]
}

const Double CO2onCO = 1. / ( 200. * exp( -9000. / ( local_R_cal * p_T ) ) * 44.0 / 28.0 ); // [ kg CO / kg CO2] => [k
for ( int r = 0; r < reactions_count; r++ ) {
  if ( local_use_co2co_l[r] ) {
    phi_l[r]  = ( CO2onCO + 1 ) / ( CO2onCO + 0.5 );
    hrxn_l[r] = ( CO2onCO * local_HF_CO2 + local_HF_CO ) / ( 1 + CO2onCO );
  }
  else {
    phi_l[r]  = local_phi_l[r];
    hrxn_l[r] = local__hrxn_l[r];
  }
}

const Double Re_p = sqrt( ( CCuVel(i,j,k) - up(i,j,k) ) * ( CCuVel(i,j,k) - up(i,j,k) ) +
                          ( CCvVel(i,j,k) - vp(i,j,k) ) * ( CCvVel(i,j,k) - vp(i,j,k) ) +
                          ( CCwVel(i,j,k) - wp(i,j,k) ) * ( CCwVel(i,j,k) - wp(i,j,k) ) )*
                          p_diam / ( local_dynamic_visc / gas_rho ); // Reynolds number [-]

const Double x_org     = (rc + ch) / (rc + ch + local_mass_ash );
const Double cg        = local_gasPressure / (local_R * gas_T * 1000.); // [kmoles/m^3] - Gas concentration
```

Results – Uintah CharOx

	KNL	ARM
Num of Inst	6x ↓	4.5x ↓
L1 Data Misses	2x ↓	3.8x ↓

- Fewer instructions, fewer L1 cache misses
- Near-ideal speedup on KNL
- No overhead on GPU
- Super linear speedup on ThunderX2



(a) Uintah CharOx: Execution time in **seconds** vs patch size.

Improving CPU performance did not hamper GPU performance!

2D Convolution

Objective	Baseline	Expected Speedup
Evaluate the LVL	CPU: Efficiently Auto-vectorized GPU: Ported to CUDA	Small speedup

Can the 4th loop be vectorized?

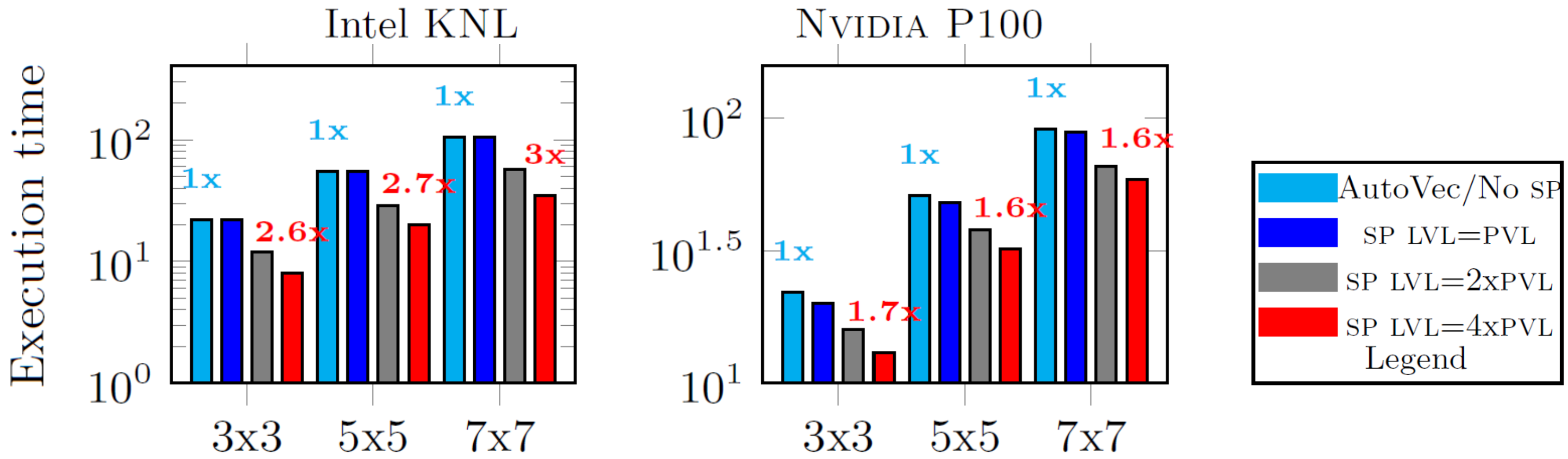
```
1: for b in 0:mini-batches
2:   for co in 0:output filter
3:     for i in 0:M              //image rows
4:       for j in 0:M            //image columns
5:         for ci in 0:channels
6:           for fi in 0:F       //filter rows
7:             for fj in 0:F     //filter columns
8:               out(b, co, i, j) +=
                  in(b, ci, i-F/2+, j-F/2+fj) *
                  filter(co, ci, fi, fj)
```

- Used in deep neural networks, image processing
- Can the 4th loop be vectorized?: Yes, #pragma simd (This forms the baseline)
- data reuse?: Yes, "filter"
- Further improvements?: Yes, unroll-and-jam the 4th loop
- The new primitive used to vectorize the code instead of #pragma simd
- LVL automatically unrolls and jams the loop

Results – 2D Convolution

	KNL	ARM
Num of Inst	2.4x ↓	1.4x ↓
Num of loads	1.5x ↓	1.6x ↓
L2 Hit Rate		3.3x ↑

- LVL increased data reuse – fewer loads, better cache hit rate
- **Improved GPU performance matches that of cuDNN (Nvidia provided cuda library for DNN) *without any hand tuning or platform specific functions***



(b) 2D Convolution: Execution time in **milliseconds** vs filter size.

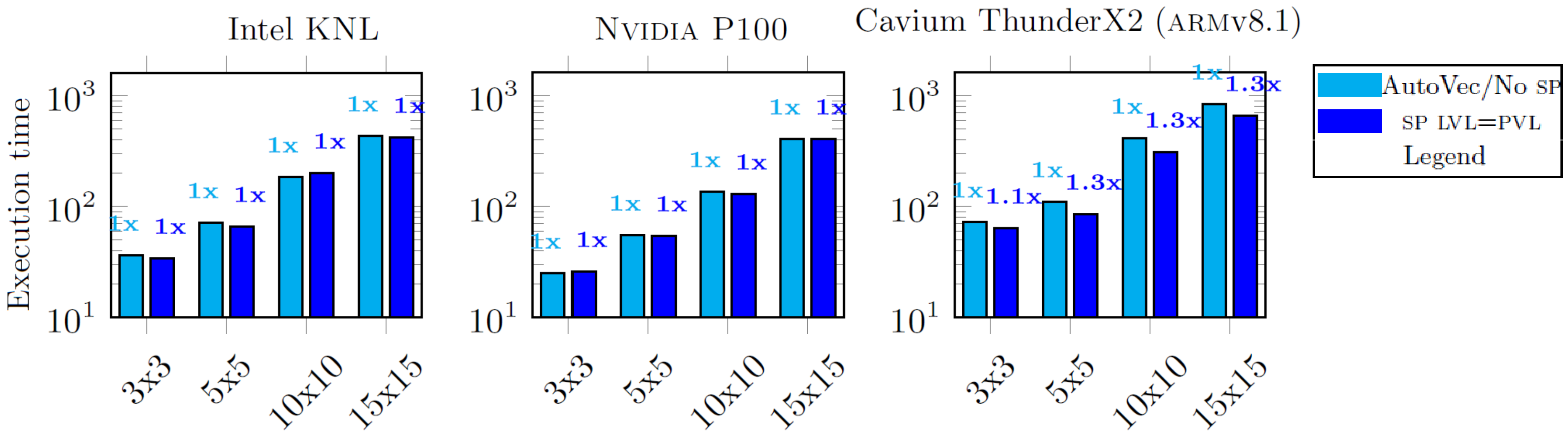
Batched GEMM

Objective	Baseline	Expected Speedup
Find out overhead	CPU: Explicitly vectorized GPU: Ported to CUDA	1x

- GEMM operations over batch of matrices organized in an interleaved fashion*
- KokkosKernels (KK) implementation forms a baseline. Used in CFD code called "SPARC"
- KK kernel vectorized using inbuilt SIMD primitive. Performs as well as (or better than) Intel mkl – a good test to find out the overhead if any
- A same code was re-implemented using the new portable SIMD primitive

Results – Batched GEMM

- No overhead on KNL / GPU.
- Small speedup on ThunderX2 due to KK's SIMD primitive lacks ThunderX2 backend



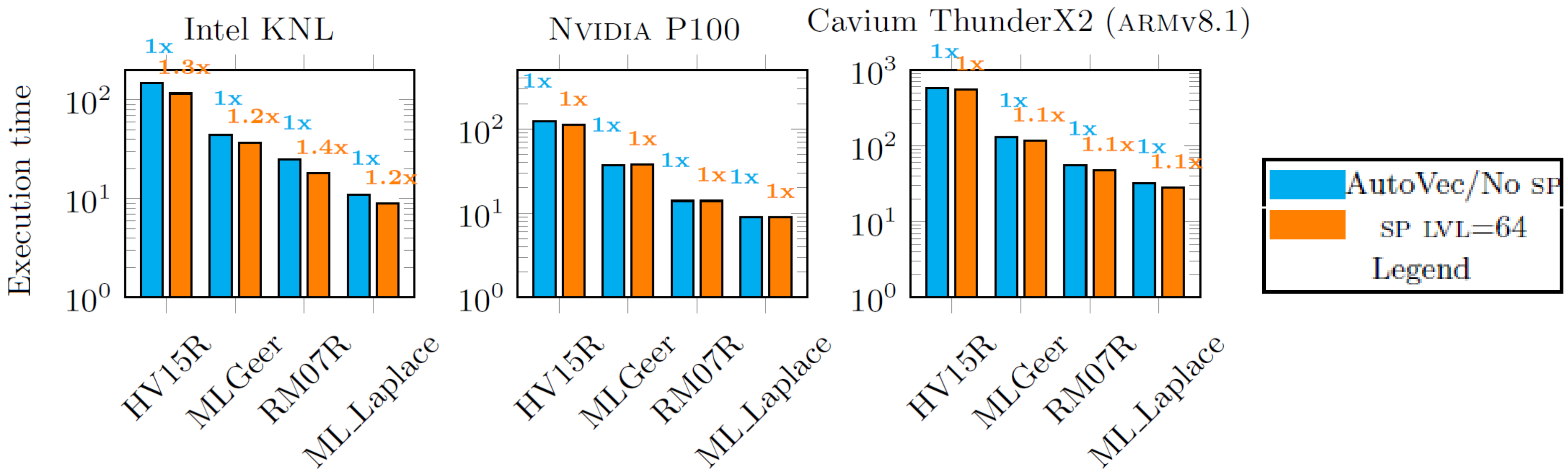
(c) Batched GEMM: Execution time in **microseconds** vs matrix size. (Batch size = 16384)

Ensembled SpMV

Objective	Baseline	Expected Speedup
Find out LVL overhead	CPU: Efficiently auto-vectorized GPU: Ported to CUDA	1x

- A sparse matrix multiplied by an “ensemble” of vectors
- Used in uncertainty quantification of predictive simulations
- Vectors arranged in an interleaved fashion. Each matrix element reused across all vectors. Provides up to 4x speedup over traditional layout*
- Baseline gets auto-vectorized
- SIMD primitive version maps input and resultant vectors to the primitive and maps LVL to length of the vector
- Matrices from University of Florida’s sparse matrix collection.

Results – Ensembled SpMV



(d) Ensemble SpMV: Execution time in **milliseconds** vs data sets (Ensemble size = 64)

- Up to 1.4x and 1.1x speedups on KNL and ThunderX2. Improved register usage
- No overhead on GPU.

Conclusions

- Extended SIMD primitive in Kokkos can provide:
  - Efficient CPU vectorization
  - GPU portability
  - GPU performance as fast raw cuda
  - LVL can add extra boost

Future Work

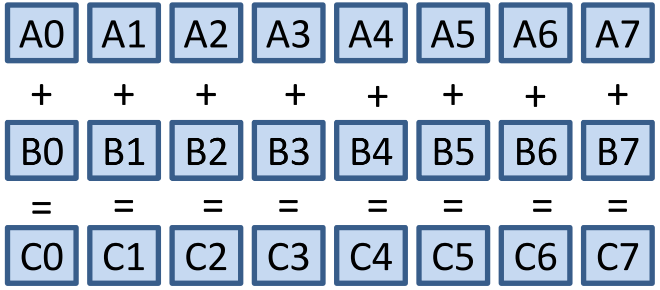
- Vectorize more kernels within Uintah
- Explore use of the SIMD primitive with OpenMP 4.5 and OpenACC

Questions?

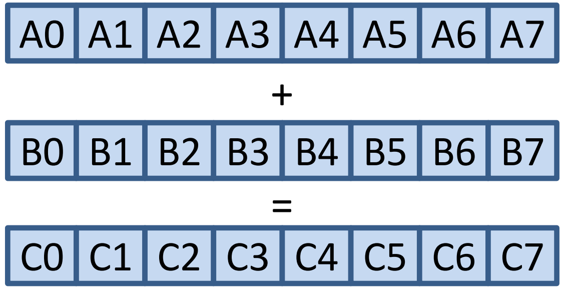
Backup slides

Single Instruction Multiple Data (SIMD) Model

- CPUs with Vector Processing Units (VPUs)
- One "vector operation" processes multiple elements at once
- Intel's Knights Landing's vector length 512 bits, Gen9 GPUs have vector length of 128 bits. Armv8 supports vector length of 2048 bits
- Vectorization is must for the performance



<u>CPU Execution</u>:
16 loads
8 adds
8 stores



<u>SIMD Execution</u>:
2 vector loads
1 vector add
1 vector store
<u>8x faster</u>

For example, on Intel KNL:

```
for(int i = 0;  i < 256;  i++)
      C[i] = A[i] + B[i];
```

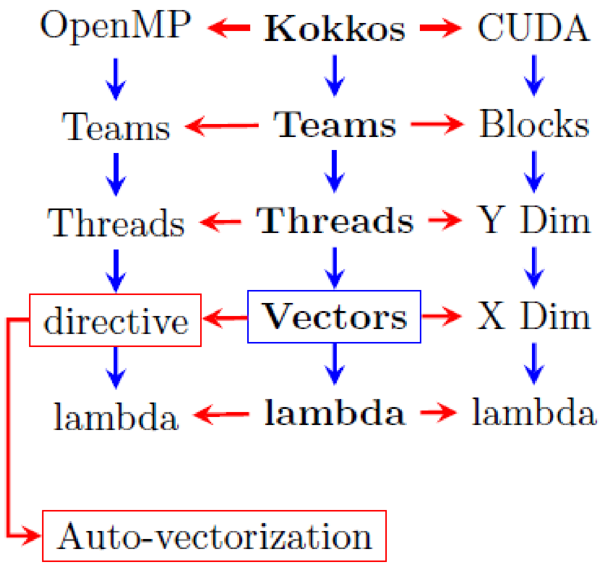
- 256 iterations without vectorization
- 32 iterations with vectorization
- <u>8x faster</u>

Vectorization strategies

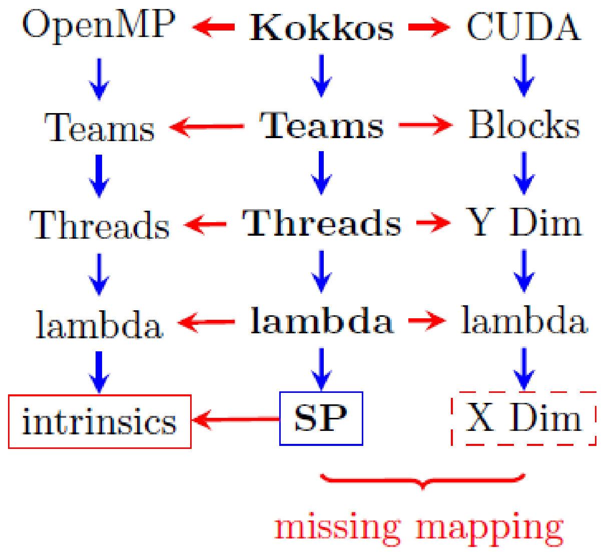
- Compiler auto-vectorization (guided by directives such as simd, vector, and ivdep)
- Explicit vectorization using intrinsics
  - platform dependent
  - consumes significant effort
  - not readable

```
    __m512d A, B, C;    ⟵ KNL simd type holding 8 doubles

    C = _mm512_add_pd(A, B)  ⟵ intrinsic to add 8 doubles
```

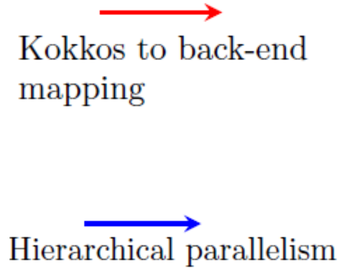
Vectorization support in Kokkos



(a) Existing Vectorization



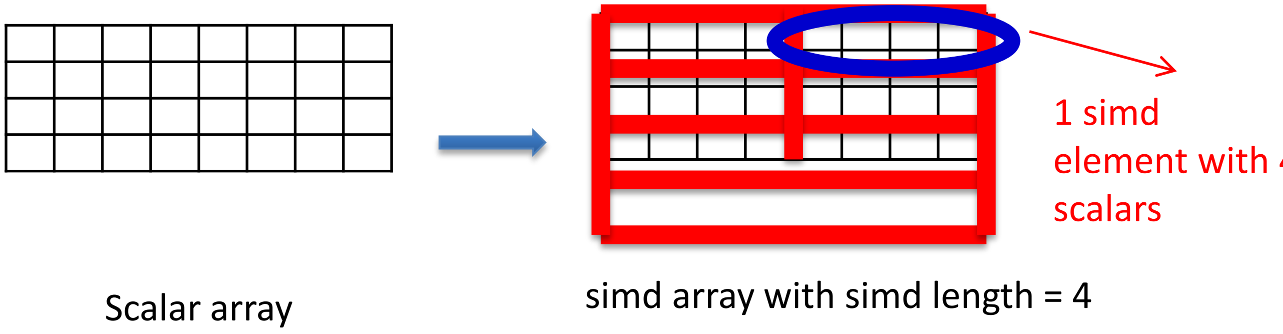
(b) If SIMD primitive (SP) used



Uintah infrastructure APIs for Portable SIMD Primitive

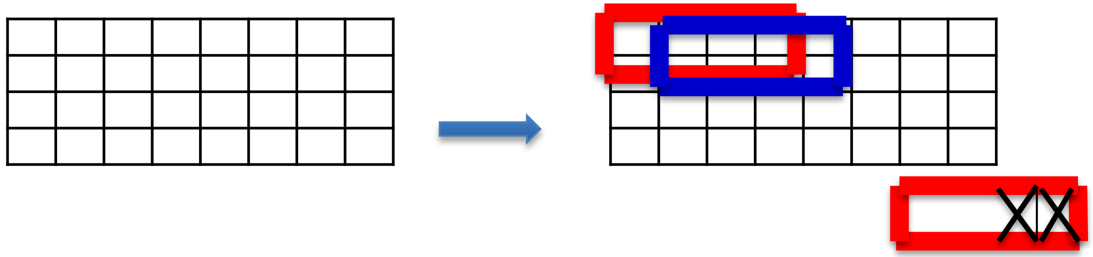
`getSimdView<ExeSpace>()`

- Casts entire data structure.
- Computation independent of neighboring cells .
  e.g. Char Oxidation.


Scalar array
1 simd element with 4 scalars
simd array with simd length = 4

to_simd()

- Casts a scalar pointer to into simd pointer.
- Computation independent of neighboring cells . e.g. stencil operations



check_simd_limit<ExeSpace> (i, i_max):
Under development. Masks simd lanes/ cuda threads going out of the boundary during remainder iterations

Implementation Challenge – temporary type

4 cuda threads declare 4 doubles (i.e. 16 doubles total) rather than declaring 4 doubles shared among 4 vector lanes

CPU equivalent:

```
#pragma omp parallel for
for(int i=0; i<N; i++){

  printf();

 __m512d a;

A = _mm512_load_pd(alpha[i])

}
```

Kokkos:

```
parallel_for( …, [=](int i){

    printf();

    simd a;

    a = alpha [i];

});
```

GPU equivalent:

```
__global__ void kernel(int i){

  printf();

  double a [ 4 ];

  a [ threadIdx.x ] =
  alpha [ i*4 + threadIdx.x ];

}
```

Example

Scalar:

```
using namespace Kokkos;
View<double *, Cuda> A(…), B(…);

//initialize A, B

parallel_for(..., [&](int i){
    B[i] += A[i];
    ...
});
```

SIMD Primitive:

```
using namespace Kokkos;
View<double *, Cuda> As(…), Bs(…);

typedef simd<double, …, Cuda>Double;
typedef View<Double *, Cuda> SimdView;
SimdView A(SimdView(reinterpret_cast<Double *>(As.data()));
SimdView B(SimdView(reinterpret_cast<Double *>(Bs.data()));

parallel_for (..., [&](int i){
    B[i] += A[i];
     ...
});
```

Fix for the temporary type

Use a temporary variable of size one for cuda, rather than SIMD type

```
//cuda version for double, blockDim.x = 32
struct Portable_Temp {
    double _d[1];
};

Kokkos:
parallel_for( …, [=](int i){
    printf();
    Portable_Temp a;
    a = alpha [i];
});
```

Declare “using Portable_Temp=simd” for CPU

Use Case	Objective	Baseline	Ideal Speedup
Uintah’s CharOx	CPU: Vectorization GPU: Find out overhead	CPU: Not vectorized GPU: Ported to cuda	KNL: 8x P100: 1x ARM: 2x
2D - Convolution	Evaluate the LVL	CPU: Efficiently Auto-vectorized GPU: Ported to CUDA	Between 1x to 2x
Batched GEMM	Find out overhead	CPU: Explicitly vectorized GPU: Ported to CUDA	1x
Ensembled SpMV	Find out LVL overhead	CPU: Efficiently auto-vectorized GPU: Ported to CUDA	1x