A Portable SIMD Primitive using Kokkos for Heterogeneous Architectures *

Damodar Sahasrabudhe[1], Eric T. Phipps[2], Sivasankaran Rajamanickam[2], and Martin Berzins[1]

[1] Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, UT, U.S.A.
{damodars,mb}@sci.utah.edu

[2] Center for Computing Research, Sandia National Laboratories, Albuquerque, NM, U.S.A.
{etphipp,srajama}@sandia.gov

Abstract. As computer architectures are rapidly evolving (e.g. those designed for exascale), multiple portability frameworks have been developed to avoid new architecture-specific development and tuning. However, portability frameworks depend on compilers for auto-vectorization and may lack support for explicit vectorization on heterogeneous platforms. Alternatively, programmers can use intrinsics-based primitives to achieve more efficient vectorization, but the lack of a GPU back-end for these primitives makes such code non-portable. A unified, portable, Single Instruction Multiple Data (SIMD) primitive proposed in this work, allows intrinsics-based vectorization on CPUs and many-core architectures such as Intel Knights Landing (KNL), and also facilitates Single Instruction Multiple Threads (SIMT) based execution on GPUs. This unified primitive, coupled with the Kokkos portability ecosystem, makes it possible to develop explicitly vectorized code, which is portable across heterogeneous platforms. The new SIMD primitive is used on different architectures to test the performance boost against hard-to-auto-vectorize baseline, to measure the overhead against efficiently vectroized baseline, and to evaluate the new feature called the "logical vector length" (LVL). The SIMD primitive provides portability across CPUs and GPUs without any performance degradation being observed experimentally.

Keywords: Portability, Vectorization, CUDA, SIMD Primitive, KNL, ARM

* The authors thank Sandia National Lab and Department of Energy, National Nuclear Security Administration (under Award Number(s) DE-NA0002375), for funding this work. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. The authors are grateful to Sandia and also Center for High Performance Computing, University of Utah for extending the resources to run the experiments. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

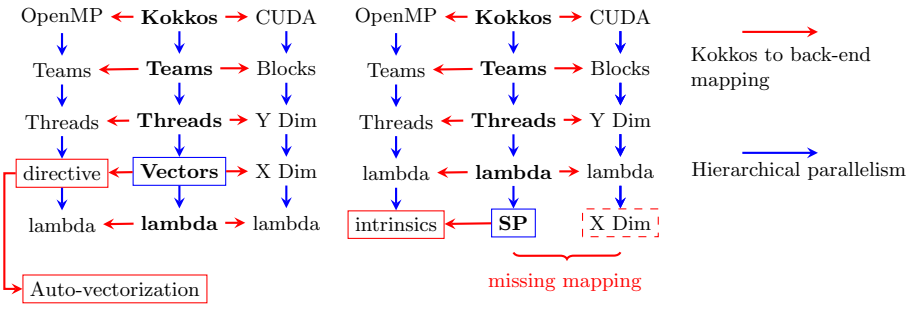
1 Introduction

Many different new computer architectures are being developed to potentially improve floating point performance such as those being developed for exascale. For example, Intel Haswell, Knights Landing (KNL), and Skylake processors support vector processing with a vector length of 512 bits. ARMv8.2-A processors have a vector length of 2048 bits [29]. Nvidia, Intel and AMD GPUs may be part of upcoming supercomputers [6, 23]. Multiple performance portability frameworks are being developed to avoid architecture-specific tuning of programs for every new architecture. Such portability frameworks as Kokkos [5] and RAJA [12] provide uniform APIs to shield a programmer from architectural details and provide a new performant back-end for every new architecture to achieve the performance portability. The Kokkos [5] library achieves performance portability across CPUs and GPUs through the use of C++ template meta-programming.

Many pre-exascale and proposed exascale CPU and many-core architectures increasingly rely on Vector Processing Units (VPUs) to provide faster performance. VPUs are designed with Single Instruction Multiple Data (SIMD) capabilities (Vector capabilities) that execute a single instruction on multiple data elements of an array in parallel. SIMD constructs can enhance the performance by amortizing costs of instruction fetch, decode, memory reads and writes [7]. The process of converting a scalar code (which processes one element at a time) into a vector code (which can handle multiple elements of an array in parallel) is known as the "Vectorization" or "SIMD transformation". Thus, effective vectorization becomes very important for any performance portability tool, including Kokkos, to extract the best possible performance on CPUs.

Another important class of supercomputers uses GPUs as accelerators (e.g., Summit, Sierra). The Single Instruction Multiple Threads (SIMT) execution model of NVIDIA's CUDA divides iterations of a data parallel kernel among multiple CUDA blocks and threads. A warp, a group of 32 CUDA threads, runs in the SIMD mode *similar* to the VPU (an exception: the latest Volta GPUs allows out of sync execution of warp threads). Any portable solution to vectorization should allow both styles of vectorization without considerable effort from application programmers. Furthermore, it is essential to distinguish between the physical vector length (PVL) in the hardware and the logical vector length (LVL) as needed by the application usage. For example, Figure 1 (a) shows how Kokkos' uniform APIs, Team, Thread and Vector [5], provide three levels of parallelism, and how they are mapped to CPUs and Nvidia GPUs. At the third level, user-provided C++11 lambda is called and loop indexes are passed to the lambda.

On Nvidia GPUS, the CUDA threads can be arranged in a three-dimensional grid. Each thread is identified by a triplet of ids in three dimensions which are accessed using "threadId.<x or y or z>". Consider the number of teams, threads and vectors requested by a user are L, T and V, respectively. In this case, GPUs, "L" Kokkos Teams are mapped to "L" CUDA blocks. The CUDA block id is mapped to the Kokkos team id. Each CUDA block is of the size VxTx1. The CUDA threads within a block can be logically divided among T partitions of size V (not to be confused with CUDA-provided Cooperative Groups). Each partition



(a) Existing Vectorization (b) If SIMD primitive (SP) used

Fig. 1: **Kokkos APIs** and mapping to CPU and CUDA

is assigned with a unique threadIdx.y ranging from 0 to T-1. Kokkos maps these partitions to Kokkos threads and the Kokkos thread id to CUDA threadIdx.y. The threads within a partition are assigned a unique threadIdx.x ranging from 0 to V-1. Kokkos Vectors get mapped to V threads within each partition. On a CPU, the Kokkos Teams and Kokkos Threads are mapped to OpenMP thread teams and OpenMP threads. Using the Kokkos Vector augments the user code with the compiler directives, which helps the compiler in auto-vectorization. This Kokkos design enables efficient SIMT execution on GPUs. However, successful automatic vectorization on CPUs depends on a lack of loop dependencies, minimal execution path divergence in the code, and a countable number of iterations, i.e., the number of iterations should be known before the loop begins execution [14]. Traditionally, compilers auto-vectorize loops that meet these criteria but fail to auto-vectorize outer loops or codes having complex control flows, such as nested if conditions or break statements. This problem can be addressed by using SIMD primitive libraries that encapsulate architecture-specific intrinsic data types and operators to achieve explicit vectorization without compromising portability across CPUs. Several such libraries exist for CPUs [17,20,27,32]. However, using SIMD primitive libraries would break the portability model as shown in Figure 1 (b). Instead of calling the Kokkos-provided **Vector**, programmers directly call the **lambda** from a **Thread**, and in turn invoke any SIMD primitive libraries, which would map user data types and functions to platform specific intrinsics. This explicit vectorization can generate more efficient code where compilers do a poor job. However, as far as the authors are aware no portable SIMD primitive library provides a GPU back-end, except OpenCL, which supports vector data types on all devices [24]. Using such primitive libraries with Kokkos, however, leads to compilation errors due to missing CUDA back-end for the primitive. *As a result, programmers are forced to make a compromise - either achieve portability at the cost of non-optimal* CPU *performance through the compiler auto-vectorization or achieve the optimal* CPU *performance using* SIMD *primitives, but maintain a separate version of code for* GPU *without using* SIMD *primitive, thereby compromising portability. Maintaining a different code for* GPUs *defeats the purpose of using Kokkos i.e., "performance portability"*. In order to remedy the situation, this work makes the following contributions:

Heterogeneous Performance Portability: The primary contribution of this work is to add a new CUDA back-end to the existing SIMD primitive in Kokkos and make the SIMD primitive portable across heterogeneous platforms with Nvidia GPUs, for the first time. (More back-ends can be added to Kokkos and to the primitive to support a wider range of heterogeneous platforms.) The CUDA back-end is developed with the exact front-end interfaces as those built for the CPU back-end. Using these uniform interfaces, the application programmers can now achieve efficient vectorization on the CPU *without maintaining a separate* GPU *version of the code*, which was not possible before. Thus, the new SIMD primitive provides GPU portability and requires only a few hundred lines of new code for the GPU back-end.

Using the new portable SIMD primitive gives a speedup up to 7.8x on Intel KNL and 2.2x on Cavium ThunderX2 (ARMv8.1) for kernels that are hard to auto-vectorize. A comparison of the primitive with *existing SIMD code* (either auto-vectorized CPU code or equivalent CUDA code) shows no overhead due to the primitive. The portable primitive provides explicit vectorization capabilities, without the need to maintain a separate GPU code. As the outer loop may now be easily vectorized using the new primitive, more efficient code can be generated than auto-vectorization of the inner loop.

Logical Vector Length (LVL): Another feature of the new primitive is the Logical Vector Length (LVL). Application developers can pass the desired vector length as a template parameter (LVL) without considering underlying physical vector length. The LVL can be used to write codes agnostic of physical vector length (PVL), as explained in Section 3. Vectorizing the outer loop coupled with the LVL automatically introduces the "unroll and jam [3]" transformations, without any burden on programmers. These transformations can exploit instruction level parallelism and data locality to provide speedups up-to 3x on KNL and 1.6x on CUDA than the auto-vectorized / SIMT code.

Easy adoption: Introducing the portable SIMD type needs less than a 10% change in user's loop. Once the primitive is introduced, the code can be explicitly vectorized on CPUS and also ported to GPUs without any further changes.

Applicability to use cases: The new portable SIMD data type supports a wide variety of computational science use cases, such as PDE assembly for complex applications, 2D convolution, batched linear algebra, and ensemble sparse matrix-vector multiplication as will be shown below.

2 Related Work

Vectorization has been studied from multiple perspectives: tools to identify vectorization opportunities [9]; portability frameworks using intermediate representations (IR) [26]; data parallel programming models [21] and data layout transformations [8]. Existing methods for improving vectorization include compiler directives, framework-based methods, tools to assist compilers [9], and language extensions [21]. Compilers provide directives that help auto-vectorization, e.g., the Intel compiler's `#pragma vector` directive instructs the compiler to override

efficiency heuristics. Intel's `#pragma simd` can be used to force vectorization (although it has been deprecated in the 2018 version). `#pragma ivdep` instructs the compiler to ignore assumed loop dependencies. OpenMP provides `#pragma omp simd`, which is similar to `#pragma simd`. Even after specifying these directives, complex control structures in a loop may prevent auto-vectorization. The LLVM community is gradually developing more advanced vectorization capabilities such as outer loop vectorization [30]. OpenCL [24], a portable parallel programming standard, provides vector data types on all the supported devices. The maximum length of a vector data type in OpenCL is limited to 16, which may be problematic for architectures with larger PVLs. On the other hand, the LVL implemented in this work can be passed as a template argument and offers more flexibility to users. In addition to Kokkos, there are other recently developed performance portable libraries such as RAJA [12], Alpaka [33] and OCCA [22]. The SIMD vectorization support in RAJA is limited to using the execution policy `RAJA::simd_exec`, which adds `#pragma omp simd` to the code and relies on compiler auto-vectorization [15]. Alpaka refactors user code to help the compiler in auto-vectorization. OCCA also provides hints to enable auto-vectorization but lacks any explicit SIMD support at present.

Multiple implementations of a SIMD primitive for CPUs such as the Vc vectorization library [20], the Unified Multi/Many-Core Environment (UME) framework [17], and the Generic SIMD Library [32] enable an explicit vectorization using architecture-specific SIMD intrinsics and operator overloading. KokkosKernels [19] is a library that implements computational kernels for linear algebra and graph operations using Kokkos. KokkosKernels uses a SIMD data type for its batched linear algebra kernels [18]. Embedded ensemble propagation [27] using the Stokhos package in Trilinos [28] for uncertainty quantification uses another version of SIMD primitives that allows flexible vector lengths. Furthermore, Phipps [27] addressed portability of this "ensemble type" to SIMT architectures. Pai [26] addressed SIMT portability using Intermediate Representations (IR).

While all these efforts are successful, they do not yet provide the full range of portability shown in this work.

3 Portable simd primitive

The portable SIMD primitive developed here sits on top of Kokkos, which provides basic performance portability across a range of architectures. Figures 2, 3 and 4 present pseudo code of the portable SIMD primitive.

Common declarations: Figure 2 shows some common declarations used to achieve portability. The PVL macro definition derives platform-specific vector length, i.e., physical vector length (PVL). "simd_cpu" and "simd_gpu" are forward declarations for CPU and CUDA primitives, respectively. They need a data type and the logical vector length (LVL) as the template parameters. An alias "simd" is created using std::conditional, which assigns "simd_cpu" to "simd" if the targeted architecture (or the execution space in the Kokkos nomenclature) is OpenMP and "simd_gpu" if the execution space is CUDA. The simd template

```
// PVL: physical vector length
// LVL: logical vector length
// EL: element per vector lane

#define PVL ... //detect architecture-specific PVL

using namespace std;

// advanced declarations
template<typename T, int LVL, int EL=LVL/PVL>
struct simd_cpu; //for cpu

template<typename T, int LVL, int EL=LVL/PVL>
struct simd_gpu; //for gpu

template<typename T, int LVL, int EL=LVL/PVL>
struct gpu_temp;

// conditional aliases for Primitive and Temp
template<typename exe_space, typename T, int LVL>
using simd = typename conditional<
  is_same<exe_space, OpenMP>::value,
  simd_cpu<T, LVL>, simd_gpu<T, LVL> >::type;

template <typename exe_space, typename T, int LVL>
using Portable_Temp = typename std::conditional<
  is_same<exe_space, OpenMP>::value,
  simd_cpu<T, LVL>, gpu_temp<T, LVL>>::type;
```

Fig. 2: Common declarations used in SIMD primitive

```
template<int LVL, int EL>
struct simd_cpu<double, LVL, EL>{

  __m512d _d[EL];  // knl instrinsic for 8 doubles

  Portable_Temp<exe_space, double, LVL> operator+ (const simd &x){
    Portable_Temp<exe_space, double, LVL> y;
#pragma unroll(EL)
    for(int i=0; i<EL; i++)
       y._d[i] = _mm512_add_pd( _d[i], x._d[i]);
    return y;
  }
  //more operators and overloads ...
};
```

Fig. 3: SIMD primitive: KNL specialization for double

expands into respective definitions at compile time depending upon the execution space. As a result, both execution spaces can be used simultaneously, thus giving portable and heterogeneous execution. The "Portable_Temp" alias and "gpu_temp" type are used as a return type and are explained later.

CPU back-end: The CPU back-ends containing architecture-specific SIMD intrinsics are developed for Intel's KNL and Cavium ThunderX2. Template specializations are used to create different definitions specific to a data type as shown in Figure 3 (which is a specialization for double on KNL). Overloaded operators invoke architecture-specific intrinsics to facilitate standard arithmetic

```
template <typename T, int LVL, int EL>
struct gpu_temp{
    T a[EL];
    //more operators and overloads ...
};

template<typename T, int LVL, int EL>
struct simd{

  T _d[LVL];

  Portable_Temp<exe_space, T, LVL> operator+ (const simd &x){
    Portable_Temp<exe_space, T, LVL> y;
#pragma unroll(EL)
    for(int i=0; i<EL; i++){
        int tid = i * blockDim.x + threadIdx.x;
        y._d[i] = _d[tid] + x._d[tid]
    }
    return y;
  }
  //more operators and overloads ...
};
```

Fig. 4: SIMD primitive: CUDA definition

```
using namespace Kokkos;
typedef View<double*> dView;
void add_scalar(dView &A, dView&B, int n){
 parallel_for(..., [&](team_member t){//team loop
  parallel_for(..., [&](int tid){//thread loop
  //calculate "start" and "end" for the thread
  for(int i=start; i<end; i++)
   for(int j=0; j<n; j++)
    if(B[i] < 1.0)
     B[i] += A[j];
  });
 });
}

typedef simd<exe_space, double, SIMD_LVL> Double;
typedef Kokkos::View<Double*, KernelSpace> SimdView;

void add_vector(dView &A, dView&B_s, int n){
 SimdView B(reinterpret_cast<Double *>(B_s.data()));
 parallel_for(..., [&](team_member t){//team loop
  parallel_for(..., [&](int tid){//thread loop
  //calculate "start" and "end" for the thread
  for(int i=start; i<end/SIMD_LVL; i++)
   for(int j=0; j<n; j++)
    B[i] = if_else( (B[i]<1.0), (B[i]+A[j]), B[i]);
  });
 });
}
```

Fig. 5: Example usage of the SIMD primitive: Conditional addition of arrays without (top) and with SIMD primitive.

operations, math library functions, if_else condition (as shown in the example later). The new primitive can support bitwise permutation operations such as

shuffle and has been verified with some preliminary experiments. One such example of an overloaded operator is shown in Figure 3. The operator+ calls the intrinsic function "_mm512_add_pd", which performs the addition of eight doubles stored in the intrinsic data type __mm512 in a simd manner. The return data type of the operator+ is "Portable_Temp". When the execution space is OpenMP, Portable_Temp is set to "simd_cpu" itself, which simply returns an intrinsic data type wrapped in the primitive. The KNL specific back-end from Kokkos::Batched::Vector is reused, and the functionalities of the LVL and alias definition based on the execution space are added on top of it. A new back-end was added for ThunderX2 using ARMv8.1 intrinsics.

Logical vector length: Users can pass the desired vector length as the template parameter "LVL". The LVL iterations are evenly distributed among the physical vector lanes by the primitive. As shown in operator+ (Figure 3), each vector lane iterates over EL iterations, where "EL=LVL/PVL", e.g., if PVL=8 and LVL=16, then EL=2, i.e. each vector lane will process two elements. Thus LVL, allows users to write vector length agnostic code. In use cases, such as the 2D convolution kernel presented in this work later, using LVL improved performance up to 3x.

CUDA back-end and Portable_Temp: Finally, a new CUDA back-end is added with the same front-end APIs as used in the CPU back-end, making the primitive portable, as shown in Figure 4. The common front-end APIs present a unified user interface across heterogeneous platforms, which allows users to maintain a single portable version of the code and yet achieve effective vectorization. The portability of the primitive avoids the development of two different versions as required prior to this work. The common front-end APIs include structures "simd" and "Portable_Temp", declared in Figure 2, along with their member functions. Whenever a programmer switches to the CUDA execution space, "simd" alias refers to "simd_gpu" and expands into an CUDA definition of the SIMD primitive. To emulate the CPU execution model of SIMD processing, the GPU back-end contains an array of "logical vector length" number of elements (double _d [LVL]). These elements divided among the PVL number of CUDA threads along the x dimension. (The PVL is auto-detected based on a platform.) CUDA assigns unique threadIdx.x to each thread ranging from 0 to PVL-1 (as explained in Section 1.) Each CUDA thread within operator+ (Figure 4) adds different elements the array _d indexed by "tid = i * blockDim.x + threadIdx.x". (In this case blockDim.x represents the number CUDA threads along x dimension, which is set to PVL.) Together, the PVL number of CUDA threads process a chunk of PVL number of elements in a SIMT manner. Each CUDA thread execute EL number of iterations (loop variable i). Thus, the primitive processes LVL=PVL*EL number of elements within array _d. Offsetting by threadIdx.x allows coalesced access and improves the memory bandwidth utilization.

However, the CUDA back-end needed an additional development of gpu_temp to be used as a return type. Consider a temporary variable of a type "SIMD" used in the CPU code. The declaration is executed by the scalar CPU thread and the elements of the variable are automatically divided among CPU vector lanes by the intrinsic function. Thus each vector lane is assigned with only "EL=LVL/PVL"

number of elements. However, when used inside a CUDA kernel, each CUDA thread i.e., each vector lane, declares its own instance of the SIMD variable. Each instance contains LVL elements and results into allocating PVLxLVL elements. The problem can be fixed by setting the alias Portable_Temp to the type "gpu_temp". "gpu_temp" holds only EL elements - exactly those needed by the vector lane. Thus, the total number of elements is still LVL. As a result, the CUDA implementation of the SIMD primitive needs combinations of operands: (SIMD, SIMD), (SIMD, Portable_Temp), (Portable_Temp, SIMD) and (Portable_Temp, Portable_Temp).

Two alternatives to avoid Portable_Temp were considered. The PVL can be set to 1 (or EL). One can even use CUDA-supported vector types such as float2 and float4. Both options will solve the return type problem mentioned earlier as each vector lanes processes 1 / 2 / 4 elements and returns the same number of elements as opposed to elements getting shared by vector lanes. Using CUDA vector types can slightly improve the bandwidth utilization due to vectorized load and store. CUDA, however, lacks any vectorized instructions for floating point operations, and the computations on these vector types get serialized. Thus, using either of these options will remove the third level of parallelism (i.e., Kokkos Vector). Hence, the Portable_Temp construct was chosen.

Example usage: Figure 5 shows an example of vectorization using the portable SIMD primitive and Kokkos, but without showing Kokkos-specific details. Kokkos View is a portable data structure used to allocate two arrays, A and B. Elements of A are added into each element of B until B reaches 1. The scalar code (add_scalar function) does not get auto-vectorized due to a dependency between if(B[i]<1.0) condition and addition. (Of course, adding #pragma simd or interchanging loops helps in this example, but may not always work.) The add_vector function, a vectorized version of add_scalar, shows how the SIMD primitive can vectorize the outer loop. Array A is cast from double to simd<double>, the number of iterations of the outer loop is factored by the LVL, and the "if" condition is replaced by an if_else operator. The statement calls four overloaded operators, namely, <, +, if_else and =. Vectorizing across the outer loop works because the outer loop iterations are not dependent on each other. If the LVL is increased to 2*PVL, the loop gets unrolled by a factor of two and each vector lane processes on two iterations consecutively. As the main computations usually take place in the innermost loop, the unrolled outer loop automatically gets jammed with the inner loop. Users can simply set LVL=nxPVL and the primitive unrolls the outer loop by a factor of n. Because the iterations of the outer loop are independent of each other, the transformation can exploit instruction level parallelism.

4 Experiments

Experimental Platforms: A node of Intel KNL with 64 cores, 16GB of High Bandwidth Memory (or MCDRAM) configured in flat quadrant mode and 192GB RAM was used to test the CPU version. Each KNL core consists of two VPUs with a vector length of 512 bits. Thus, using the double precision floating point numbers allows a vector length of 8. The codes were compiled with the Intel compiler suite 2018 with the optimization flags -O3 -xMIC-AVX512 -std=c++11 -fopenmp.

Tests were also run on a single node of the Astra cluster at Sandia National Laboratories. Each Astra node provides 128 GB of high bandwidth memory and two Cavium ThunderX2 CN99xx processors with 28 cores each. Cavium ThunderX2 is an ARMv8.1-based processor with a vector length of 128 bits. It can execute two double precision operations with a single SIMD instruction. The GNU 7.2.0 compiler suite was used to compile applications with the flags -O3 -std=c++11 -mtune=thunderx2t99 -mcpu=thunderx2t99 -fopenmp.

The NVIDIA P100 GPU with Compute Capability 6.0, 3584 CUDA cores, 16GB of High Bandwidth Memory and 48 KB of shared Memory per SM was used to test the GPU performance. The applications were compiled using gcc v4.9.2 and nvcc (from CUDA v 9.1) with the optimization flags -O3 -std=c++11 –expt-extended-lambda –expt-relaxed-constexpr.

Use case	**Goal**	**Baseline**	**Expected Performance**	
			CPU	**GPU**
PDE	CPU: Achieve effective Vectorization for the complex, hard to vectorize code; GPU: Find out the overhead for a performance sensitive portable kernel.	CPU: not vectorized; GPU: Ported to CUDA.	Near ideal speedup.	No extra speedup. No new overhead.
2dConv	Evaluate the benefit of LVL by comparing it with a baseline already running in SIMD mode.	CPU: auto-vectorized; GPU: Ported to CUDA.	Small extra speedup due to LVL.	Small extra speedup due to LVL.
GEMM	Find out the overhead of the primitive by comparing it with a baseline already running in SIMD mode efficiently.	CPU: auto vectorized; GPU: Ported to CUDA.	No extra speedup. No new overhead.	No extra speedup. No new overhead.
SpMV				

Table 1: Summary of use cases, goals and expectations.

†Baselines are written using Kokkos for two reasons: first to make the code portable and second to measure the overhead of the primitive *only*. If the baseline is written using raw CUDA or OpenMP, then the performance measurements will include the overhead of both Kokkos and the primitive.

Use cases and Experimental setup: The primary aim of the portability libraries such as Kokkos is to enable "performance portable" programming. The portable code gets compiled and executed on the heterogeneous platforms without making any platform-specific changes and also provides performance close to the native implementations (such as using raw CUDA or using vector intrinsics). However, it is essential to understand that Kokkos or the SIMD primitive is not a magic construct to provide an extra performance boost. *When the baseline itself is efficiently vectorized or has an efficient* CUDA *implementation, using Kokkos or the primitive can provide portability, but will not provide extra speedup.* Considering these factors, different use cases are chosen to test the per-

Algorithm 1 CharOx Loop Structure, Holmen [10]

```
1: for all patches
2:  for all Gaussian quadrature nodes
3:  Kokkos::parallel_for cells in a patch //Can cells loop be vectorized?
4:      Compute reaction constants.
5:      Nested loops over reactions and species.
6:      Multiple loops over reactions and species.
7:      Nested loops over reactions and species.
8:      while residual < threshold do //indefinite number of iterations
9:          Multiple loops over reactions.
10:         Nested loops over reactions and species.
11:         Compute a matrix inverse.
12:         Multiple loop over reactions.
13:     end while
14:     Loop over reactions.
15:  end Kokkos::parallel_for
```

formance of the SIMD primitive for different scenarios. Table 1 summarizes the four different kernels used in the evaluation. Of the four use cases, first two are not efficiently vectorized and are chosen to demonstrate the effectiveness of the primitive, whereas last two are efficiently vectorized and are chosen to measure the overhead of the primitive.

Use cases were implemented using Kokkos - first without using the SIMD primitive and then using it. A typical transformation from scalar code to vectorized code using the portable SIMD primitive needs casting of legacy data structures and variables and updating any conditional assignments. Some algorithm specific use cases need a special handling, e.g., the while loop discussed in Section 4.1. All the arithmetic operations and math library functions remain untouched. *For all four kernels, less than 10% of the lines of code were modified to introduce the* SIMD *primitive* and this did not require any complex code transformations or new data structures, which is typically needed to auto-vectorize a complex code. The use of Kokkos and the SIMD primitive allows the same code to be compiled on the different target platforms. Various combinations of the number of threads were tested and the best timing was chosen. Each experiment was repeated at least 100 times, and the averages timings were used to compute the speedups.

4.1 PDE Assembly

Uintah [2] is a massively parallel asynchronous multi-task runtime that can be used to solve complex multi-physics problems. It is being used for the multi-scale and multi-physics combustion simulation of a coal boiler under DoE's PSAAP II project and has been successfully scaled to 256k cores on Titan and 512k cores on MIRA [2]. One of the longest running kernels within Uintah is CharOx. It simulates the char oxidation of coal particles by modeling multiple chemical reactions and physical phenomenon involved in the process [1,11,25]. The CharOx kernel consists over of 350 lines of code, reads around 30 arrays, updates five arrays, and performs compute-intensive double precision floating point arithmetic operations, such as exponentials, trigonometric functions, and divisions in nested

loops about 300 to 500 times for every cell. As shown in Algorithm 1, the main cell iterator loop contains different loops over reactions and species, each with multiple levels of nesting. The Newton Raphson Solve loop has an undetermined number of iterations and contains more nested loops. The iterations of the cell iterator loop are not dependent on each other. Hence, vectorizing the cell iterator loop can potentially give a maximum speedup.

Auto-vectorization: On the KNL platform, the Intel compiler auto-vectorizes some of the innermost loops only. Vectorization of the cell iterator loop can be forced by adding the "#pragma simd" directive. However, *this provides only 4.3x speedup,* whereas the ideal speedup for the double precision on KNL is 8x, assuming the majority of the code is scalar. (Unfortunately, the pragma is deprecated in Intel compilers from 2018 onwards, and its replacement "#pragma vector" fails to "force" vectorize the cell loop.) An inspection of the vectorization report and assembly code shows gather/scatter instructions generated for every read/write to the global arrays. These gather and scatter instructions are the reason for the speedup of 4.3x. To maintain the halo region, Uintah internally offsets *all* elements in its data structures with a *constant* value. All cells have the *same* offset. Thus the stride between elements is always one, but the compiler cannot deduce this and generates gather instructions. However, using the SIMD primitive calls SIMD intrinsics that explicitly generate move instructions rather than gather and makes vectorization efficient.

The GNU compilers used on the ThunderX2 platform did not vectorize the cell iterator loop even after adding vectorization hint directives.

On the GPU, the size and the complexity of the kernel substantially increases register usage. Profiling shows that 255 registers are used by every thread within a block, thereby preventing the simultaneous execution of multiple blocks on a single Streaming Multiprocessor (SM). This results in poor occupancy of the SMs (only up to 12%). Hence, the SIMD primitive must not add additional overhead in terms of registers, memory, or execution dependency, and the GPU performance must not be compromised to gain CPU performance. Apart from casting data structures and variables to those based on the portable SIMD primitive, the Newton-Raphson solver [10] used to solve oxidation equations for every cell needed special handling. The solver iterates until the equations converge. In the vectorized version, the vector of cells iterates until all cells within the vector converge. Although the technique needs extra iterations for a few cells, it works faster than executing solver iterations sequentially in a scalar mode.

The experiments were carried out using a total of 64 patches with two patch sizes - 16^3 and 32^3. The CharOx kernel is invoked five times for every patch. With 64 patches, the kernel is executed 320 times in every timestep. The simulation was run for 10 timesteps, and the average loop execution time over 3200 calls was recorded. The shear complexity of this loop appears to provide a distinctive and unusual challenge for performance portability.

Goal: This use case shows a particular instance where the compiler does a poor job in auto-vetorizing the code on the CPU, but the CUDA code works efficiently on the GPU. *It is thus important to ensure that improving* CPU *performance us-*

ing the primitive does not degrade GPU *performance.* The kernel is large and complex enough to cause a register spill on the GPU even without using the SIMD primitive. Thus, adding SIMD will help us to understand the performance of sensitive kernels on GPU and associated overhead, if any.
Expectation: The code performs double precision floating point operations. Hence, the speedups close to 8x and 2x are expected on KNL and ThunderX2, respectively. These are the ideal speedups for double precision computations on these platforms considering the respective vector lengths of 512 bits and 128 bits. The GPU code already runs in a SIMT mode, and hence the new primitive will provide portability without a performance boost. However, portability should not cause any significant overhead either. Ideally, GPU performance should remain the same with and without SIMD.

4.2 2D Convolution

Algorithm 2 Algorithm for a 2D Convolution Kernel

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

2D convolution [4] (as shown in Algorithm 2) is a heavily used operation in deep neural networks. The algorithm multiplies a batch of images (*in*) with a filter (*filter*) by sliding the filter over the image to accumulate the result (*out*). The operation is repeated for multiple filters. This algorithm has a high arithmetic intensity. Using the SIMD primitive provides an opportunity to exploit the spatial locality for all three variables: When the "i" loop is parallelized across the Kokkos threads and the "j" loop across the SIMD lanes, every "filter" element is reused for the "LVL" number of "j" iterations. Also, two levels of parallelism help reusing elements in different rows "in" and "out" (similar to a stencil block). Of course, these improvements can be obtained manually without using the primitive. However, using the primitive introduces these transformations implicitly and improves the programmability, portability and the maintenance of the code.

The mini-batch loop in the original code is parallelized across OpenMP threads/CUDA blocks. The code is then auto-vectorized across the j loop using the directive #pragma simd on CPU and mapping the x dimension CUDA threads across the j loop on a GPU. #pragma unroll was used to unroll the full lengths of the fi and fj loops. The code was then converted into a SIMD primitive code instead of using #pragma simd. Different combinations of mini batch sizes (3584 and 7168), filter sizes (3x3, 5x5 and 7x7), number of input (3, 5 and 10) and output channels (3, 5 and 10) were tested for different values of the LVL.
Goal: Evaluate the effectiveness of the LVL against the vectorized baseline.
Expectation: As the baseline is efficiently auto-vectorized, using SIMD primitive with LVL=PVL will not perform any better. However, setting LVL=2*PVL

Name	Rows	Columns	Nonzeros	Execution time (ms)			
			Nonzeros	mkl KNL	baseline KNL	cusparse P100	baseline P100
HV15R	2017169	2017169	283073458	275	147	160	123
ML_Geer	1504002	1504002	110686677	105	44	54	37
RM07R	381689	381689	37464962	46	25	23	14
ML_Laplace	377002	377002	27582698	34	11	13	9

Table 2: sparse matrices used for ensemble SpMV and comparison of the baseline Kokkos version with Intel's mkl and Nvidia cusparse libraries for ensemble size = 64.

or 4*PVL should give speedups on both CPU and GPU due to instruction level parallelism and data reuse.

4.3 Compact gemm

The general matrix-matrix multiplication (GEMM) on a batch of small, dense, matrices is widely used within scientific computing and deep learning. Thread-parallel GEMM operations over collections of matrices organized in an interleaved fashion can be made efficient and portable using the SIMD primitive [18]. This approach is implemented within KokkosKernels, and used in a large-scale CFD code called SPARC [13]. The KokkosKernels' batched GEMM kernel achieves performance comparable or sometimes better than vendor provided libraries such as Intel's math kernel library (mkl) and Nvidia's cuBLAS [18, 31]. KokkosKernels maintains two versions of batched GEMM - the CPU version which uses an intrinsics-based SIMD primitive, and a CUDA version, which does not have a SIMD primitive. The only change needed in the kernel to utilize the portable SIMD primitive was to map the matrix dimension to the SIMD dimension by casting matrices from Kokkos views of doubles to Kokkos views of the SIMD primitive. Thus, each CPU thread (or a section of a CUDA warp) carried out each operation on the LVL number of matrices in SIMD fashion. Both kernels had the same tiling optimizations with tile sizes of 3 x 3 and 5 x 5 to extract spatial and temporal locality among matrix elements. Experiments were carried out using four matrix sizes: 3 x 3, 5 x 5, 10 x 10 and 15 x 15 using a batch of 16,384 matrices on all three platforms.

Goal: The goal is to compare the performances of the new SIMD primitive and the existing high performance explicitly vectorized code on CPU. Any performance degradation will reveal the associated overheads, if any.

Expectation: The SIMD primitive should perform as well as the code without the SIMD primitive on both CPU and GPU. Neither a performance boost nor any extra overhead is expected.

4.4 Embedded Ensemble Propagation

This kernel is heavily used in the uncertainty quantification of predictive simulations which, involve evaluation of simulation codes on multiple realizations of input parameters. The efficiently auto-vectorized baseline kernel multiplies a sparse

matrix by an ensemble of vectors with matrix rows distributed across threads and vectors distributed across SIMD lanes. Vectors are arranged in an interleaved fashion similar to batched GEMM. This design, introduced by Phipps [27], allows the reuse of matrix values across all vectors and provides up-to 4x speedups over traditional batched sparse matrix-vector multiplication. Instead of repeating Phipps' experiments, the vectorized ensemble version itself is used as a baseline. Compared to the vendor-provided libraries, i.e., Intel's mkl and Nvidia's cusparse, the baseline kernels exhibit 1.8x to 3x speedup on KNL and 1.3x to 1.6x on P100, respectively (See Table 2). These observations are in line with Phipps' experiments. This baseline kernel is converted to use our SIMD type by casting data structures from double to those using the SIMD primitive and setting the LVL equal to the ensemble length. Hence, any performance degradation from baseline can show the shortcomings in the LVL implementation.
Goal: The goal is to find out any overhead associated with the SIMD primitive by evaluating its performance against the highly optimized baseline that implements the same design and parallelism pattern but without using the primitive.
Expectation: The code with the SIMD primitive should perform as well as the baseline on CPU and GPU. No performance boost and no overhead is expected.

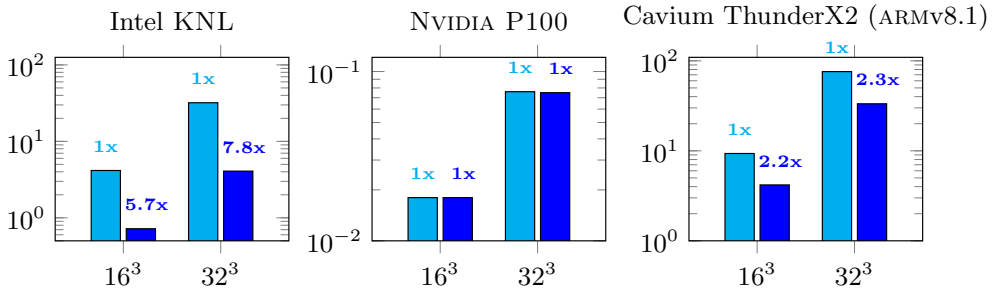
5 Results and Performance Analysis

The rows in Figure 6 show the results of four use cases, and the columns indicate three platforms. Each plot shows execution time along with the speedup compared to the baseline. The baseline is either the auto-vectorized code (AV) or the code with no SIMD primitive (NSP) colored in cyan. Results of using the SIMD primitive with different values of the LVL are represented by "SP". As mentioned earlier, the experiments have three goals: a) Find out performance improvement when the code is not efficiently vectorized (PDE and 2dConv cases), b) Ensure that performance improvement on one platform, does not hamper the performance on another platform (PDE), and c) Measure the overhead of the new primitive against the efficiently vectorized baseline, where the expected speedup is 1x (GEMM and SpMV). The performance of each use case is analyzed below.

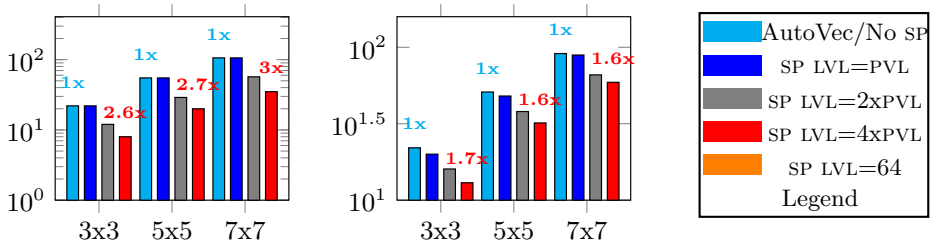
The vectorized code (AV and SP) both executes fewer instructions than the scalar code, but the vector instructions execute more slowly than the scalar counterparts, consuming more cycles. Hence, the Instructions Per Cycle (IPC) count does not reflect the exact speedup. Similarly, KNL hardware counters do not accurately measure floating point operations (FLOPs), and numbers often get skewed while measuring floating point instructions (FLIPs) [16]. Hence, simple counts such as the total number of instructions and cache hits are used here for performance analysis. The performance metrics and events are collected using Intel vtune amplifier, Nvidia nvprof, and the PAPI library.

5.1 PDE Assembly

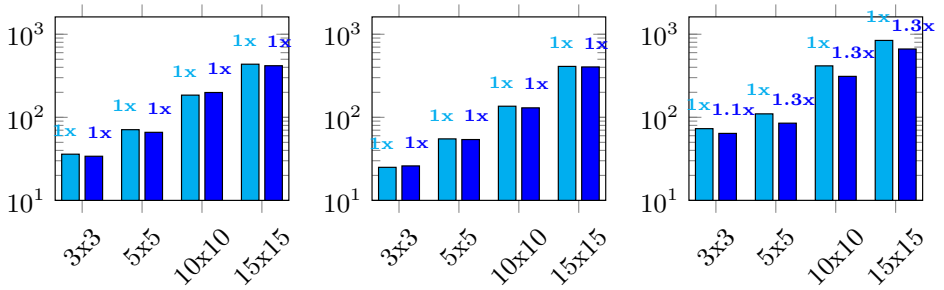
The KNL plot in Figure 6 (a) shows the SP version that achieves 5.7x and 7.8x speedups over AV for mesh patch sizes of 16^3 and 32^3, respectively. Analysis of



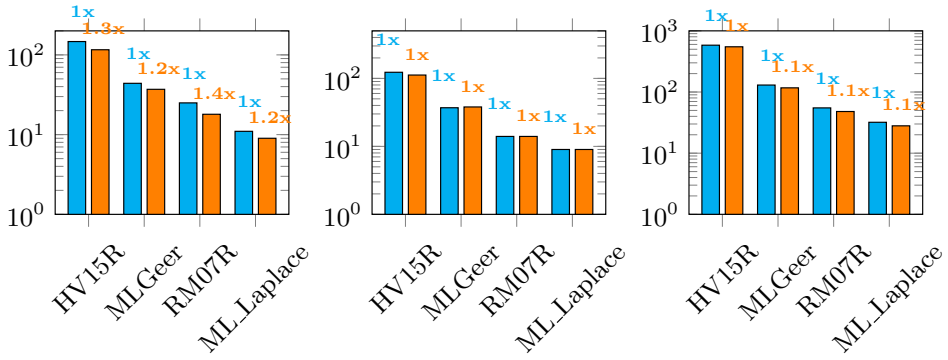
(a) Uintah CharOx: Execution time in **seconds** vs patch size.



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



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



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

Fig. 6: Comparison of execution times along with speedups for different kernels on different architectures. Speedup "1x" indicates zero new overhead due to the new primitive.

the 16^3 patch problem shows the number of instructions (INST_RETIRED.ANY) executed reduced from 1273 million for the AV code to 204 million for the SP code (Table 3). Similarly, L1 cache data misses (PAPI_L1_DCM) decreased from 2.4 million for AV to 1.2 million for the SP. In this case, some of the cache lines are evicted over the course of one iteration due to the complex operations and the 30+ different arrays used in the kernel. When the next iteration starts, at least some of the memory is missing from the L1 cache due to earlier evictions. However, the vectorized code can take advantage of entire cache lines, and all eight double elements from the 64 bytes cache lines can be read by eight vector lanes, thus fully utilizing data fetched in a cache line. The increased cache line efficiency along with vectorization provides near optimal speedup.

The P100 results show the NSP and SP both performing equally well. As the NSP running on the GPU runs in SIMT mode, the SP does not provide any extra level of parallelism and cannot provide an extra boost. These results showing 1x speedup are important in showing that the SP does not create any overhead on a GPU, even when the NSP kernel causes register spilling. All the metrics collected by nvprof showed similar values in this case. Increasing the value of LVL to 2xPVL slowed down the performance by 1.5x, because the increased LVL increased register spilling (evident from increased local memory accesses).

The SP version of CharOx kernel boosted performance by 2.2x and 2.3x for patch sizes of 16^3 and 32^3, respectively, on ThunderX2. The ThunderX2 metrics show a similar trend as that observed on KNL. The total number of instructions executed are reduced from 4253.8 million for the NSP to 1823.1 million for the SP. Again, vectorization reduced the number of cache misses from 5.7 million to 1.5 million, which provided super-linear speedups up to 2.3x where the PVL supported by the hardware for double precision is only 2.

	Intel KNL		Cavium ThunderX2	
	number of instructions	L1 cache data misses	number of instructions	L1 cache data misses
No SIMD primitive	1273	2.4	4253	5.7
SIMD primitive	204	1.2	1823	1.5

Table 3: Performance metrics for CharOx. (counts in millions)

5.2 2D Convolution

Figure 6 (b) shows speedups up to 3x on KNL and 1.6x on P100 for the 2D Convolution kernel shown in Algorithm 2. The input image size, the number of input and output channels were set to 64x64, 3 and 10, respectively. The image was padded by filter size / 2 number of cells. The baseline NSP and the SP with LVL=PVL perform equally well as both get efficiently vectorized. Setting LVL=2xPVL and 4xPVL gives better results on both KNL and P100.

Line number 8 of Algorithm 2 multiplies "in" with "filter" and accumulates the result in "out". Vectorizing the j loop coalesces accesses for "in" and "out". "filter" is independent of the j dimension, and hence the value is reused across all vector lanes. When the LVL is set to 2xPVL (or 4xPVL), the same filter value gets

reused across twice (or four times) the PVL elements. An assembly instruction inspection shows the register containing "filter" was reused across multiple fma operations. All these fma operations are independent of each other and can exploit instruction level parallelism. This "unroll and jam" transformation can be introduced by simply increasing the value of LVL. Using LVL in this case can save developers having to manually perform "unroll and jam" - especially for larger codes - and maintain readability of the code.

	Intel KNL		Nvidia P100		
	number of instructions	number of memory loads	number of instructions	number of memory loads	L2 cache hit rate
No SIMD primitive	6.2	3	9.8	1.3	25%
SIMD primitive	2.6	2	6.8	0.8	83%

Table 4: Performance metrics for 2DCov. (counts in billion)

The reuse of the "filter" values across the j iterations reduced the number of memory loads from 3 billion for the NSP to 2 billion for the SP with LVL=4xPVL on KNL and from 1.3 billion to 0.8 billion on the P100. The number of instructions executed reduced by 2.3x and 1.4x on KNL and P100 platforms, respectively. The L2 cache hit rate improved on the P100 from 25% to 83%. Additionally, the number of control flow instructions executed was reduced by a factor of 3.5 on the P100 due to outer loop unrolling (see Table 4). The effectiveness of the primitive and LVL can be judged from the fact that the naive code in Algorithm 2 with the SIMD primitive and LVL=4xPVL was only 20% slower than the highly tuned cuDNN library by Nvidia as shown in Table 5. A small fix to use GPU's constant memory to store the filter gave additional boost and the naive code performed equally well as the cuDNN library. Thus, the primitive can help application programmers who may focus on the algorithms and applications rather than spending time on specialized performance improvement techniques such as tiling, loop unrolling, using shared memory, etc.

Filter size	cuDNN	simd primitive LVL=4xPVL	simd primitive LVL=4xPVL with constant memory
3x3	11	13	11
5x5	24	31	25
7x7	49	59	47

Table 5: Performance comparison with Nvidia cuDNN. Execution time in milliseconds

Unfortunately, experiments for 2D Convolution could not be conducted on ThunderX2 because the Astra cluster was moved to a restricted domain by Sandia National Laboratories.

5.3 Compact gemm

Figure 6 (c) shows that the NSP and SP versions perform equally well on the KNL and P100 (speedup is 1x) and that the SP does not create any overhead.

These results are as expected because KokkosKernels (the NSP version) contains explicitly vectorized code for KNL and Kokkos code tuned explicitly for GPUs. These observations are confirmed by the same number of instructions executed by the NSP and SP versions - 23 million on KNL and 20 million on P100.

However, the ThunderX2 results show an improvement of up to 1.3x. The architecture-specific intrinsic back-end for ThunderX2 had not yet been updated in the KokkosKernels, and it falls back to an emulated back-end using arrays and "for" loops. Although this NSP version gets auto-vectorized by the compiler, the SP leads to more efficient vectorization. The NSP version executes 146 million instructions whereas the SP version executes 114 million instructions on ThunderX2.

5.4 Embedded Ensemble Propagation

The kernel is evaluated using 13 matrices from the University of Florida sparse matrix collection. However, results from only four matrices (listed in Table 2) that represent the general trend are presented for the sake of brevity. Sparse matrix-vector ensemble multiplication results on the GPU shown in Figure 6 (d) indicate both versions, NSP and SP, perform equally on the P100 GPU. Both the versions are efficiently ported to the SIMT model and use the same ensemble logic for data reuse. Therefore, the matching GPU performance for both versions meets our expectation and indicates that the primitive does not cause any overhead.

More surprising were speedups up to 1.3x on KNL and and 1.1x ThunderX2. Profiling showed about 10% to 20% reduction in the number of instructions executed for different sparse matrices and different ensemble sizes. While the FLOPs were, of course, the same for both versions, an assembly code inspection revealed the reason behind the speedups. The result of matrix - vector ensemble multiplication is also a vector ensemble. The design by Phipps et al. [27] fetches a matrix element and multiplies all vectors with it to avoid repeated accesses to matrix elements, which are costly when the sparse matrix is stored in the "compressed row storage" format. Although the Phipps design performs faster than the traditional batched multiplication, it has to repeatedly fetch elements from the resultant vector ensemble to do the accumulation. In the NSP version, the compiler generates three vector instructions for every vector operation: (i) a fetch of the result ensemble from memory to a vector register, (ii) a vectored fused-multiply-add (fma) on the result stored in a vector register with a vector from memory and a matrix element stored in vector register, and (iii) a store of the result from the vector register in the memory. When the SP is used, the ensemble length is mapped to the LVL. This mapping helps the compiler to deduce the array length and number of registers. Hence, for N=64, all result elements get loaded into eight vector registers only once, and fma operations are repeated on these registers. Hence, using the SP eliminates the need to transfer the result back and forth from the memory and takes only one store to move the accumulated result from the vector registers to the memory. Thus, one load and one store are saved for every fma operation, resulting in a more efficient code.

6 Assigning the optimal LVL value

The LVL value depends on register availability and levels of parallelism, both dictated by the algorithm and hardware. If the LVL is set to 2xPVL or 4xPVL, the compiler can usually allocate the structure into registers. Then the code can take advantage of instruction level parallelism, if supported by the hardware, as observed in the cases of 2DCov and SpMV. When, however, the LVL was set to 8xPVL, the compiler allocated the structure into memory instead of registers and so hampered the performance of 2DCov with extra loads and stores. The CharOx kernel is very complicated and register spilling happens even in the NSP. Therefore, setting LVL=2xPVL, increased the register pressure further and resulted in slower execution in contrast to other use cases. The second factor in choosing the right LVL is the number of levels of parallelism an algorithm can offer. If the both levels of parallelism, thread level and SIMD level, are applied to the same loop (as in GEMM or CharOx), then increasing the LVL effectively increases the workload per thread and decreases the degree of parallelism available, which can cause a load imbalance among cores. The GEMM kernels were hand-tuned to unroll and jam along matrix rows and columns. The optimization gave enough workload to fully exploit available instruction level parallelism. As a result, increasing the LVL did not provide any further advantage.

7 Conclusion and Future Work

This study describes a portable SIMD data type whose primary benefit is to achieve vectorization in a portable manner on architectures with VPUs and GPUs. This capability has a potential to be useful for massive applications that use Kokkos to extract performance from future architectures (including exascale architectures), without explicitly tuning the user code for every new architecture. The largest benefits the SIMD primitive were observed in the most complex kernel, which was hard to auto-vectorize. Performance boosts of up to 7.8x on KNL and 2.2x on Cavium ThunderX2 can be observed for double precision kernels (PDE). For the kernels which are vectorized/ported to GPUs, the new SIMD primitive results in the speedups up-to 3x on KNL, 1.6x on P100 and 1.1x on ThunderX2 due to more efficient vectorization (SpMV), cache reuse (2dConv), instruction level parallelism (2dConv) and loop unrolling (2dConv and SpMV). The comparison with efficiently vectorized kernels showed minimal overhead for PDE and zero overhead for GEMM and SpMV kernels. The new primitive makes outer loop vectorization easier (as shown with CharOx, SpMV and 2dConv). The PDE example proved that performance on one platform can be improved without compromising the performance on another platform.

The Kokkos-based design will make it easier to port this SIMD primitive to future GPU exascale architectures such as A21, and Frontier. The Kokkos profiling interface can possibly be extended to profile the primitive-based code in the future. Preliminary experiments showed that the new primitive can be easily extended to both OpenACC and OpenMP 4.5. It will be interesting to compare the performance of OpenACC, OpenMP 4.5 / 5.0 (in the future), and Kokkos.

References

1. Adamczyk, W., I., B., Parra-Alvarez, J., Smith, S., H., D., Thornock, J., Z., M., Smith, P., Ż., R.: Application of LES-CFD for predicting pulverized-coal working conditions after installation of NOx control system. Energy **160**, 693–709 (10 2018)
2. Berzins, M., Beckvermit, J., Harman, T., Bezdjian, A., Humphrey, A., Meng, Q., Schmidt, J., , Wight, C.: Extending the Uintah Framework through the Petascale Modeling of Detonation in Arrays of High Explosive Devices. SIAM Journal on Scientific Computing (Accepted) (2016), http://www.sci.utah.edu/publications/Ber2015a/detonationsiam16-2.pdf
3. Carr, S.: Combining optimization for cache and instruction-level parallelism. In: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Technique. pp. 238–247. IEEE (1996)
4. Cope, B., et al.: Implementation of 2D Convolution on FPGA, GPU and CPU. Imperial College Report pp. 2–5 (2006)
5. Edwards, H., Trott, C., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing **74**(12), 3202–3216 (2014)
6. of Energy, D.: U.S. Department of Energy and Cray to Deliver Record-Setting Frontier Supercomputer at ORNL. https://www.energy.gov/articles/us-department-energy-and-cray-deliver-record-setting-frontier-supercomputer-ornl (2019)
7. Espasa, R., Valero, M.: Exploiting instruction-and data-level parallelism. IEEE micro **17**(5), 20–27 (1997)
8. Henretty, T., Stock, K., Pouchet, L., Franchetti, F., Ramanujam, J., Sadayappan, P.: Data layout transformation for stencil computations on short-vector simd architectures. In: International Conference on Compiler Construction. pp. 225–245. Springer (2011)
9. Holewinski, J., Ramamurthi, R., Ravishankar, M., Fauzia, N., Pouchet, L., Rountev, A., Sadayappan, P.: Dynamic trace-based analysis of vectorization potential of applications. ACM SIGPLAN Notices **47**(6), 371–382 (2012)
10. Holmen, J.: Private communication (2018)
11. Holmen, J., Peterson, B., Humphrey, A., Sunderland, D., Daz-Ibarra, O., Thornock, J., Berzins, M.: Improving Uintah's Readiness for Exascale Systems Through the Use of Kokkos and Nested OpenMP. Journal of Parallel and Distributed Computing (in preparation) (2018)
12. Hornung, R., Keasler, J.: The RAJA portability layer: overview and status. Tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States) (2014)
13. Howard, M., Fisher, T., Hoemmen, M., Dinzl, D., Overfelt, J., Bradley, A., Kim, K., Rajamanickam, S.: Employing Multiple Levels of Parallelism for CFD at Large Scales on Next Generation High-Performance Computing Platforms (2018), proceedings of the Tenth International Conference on Computational Fluid Dynamics, ICCFD$_1$0, Barcelona, 9–13 July 2018
14. (Intel), M.C.: Requirements for Vectorizable Loops (2012), https://software.intel.com/en-us/articles/requirements-for-vectorizable-loops
15. Jacob, A., Antao, S., Sung, H., Eichenberger, A., Bertolli, C., Bercea, G., Chen, T., Sura, Z., Rokos, G., O'Brien, K.: Towards performance portable gpu programming with raja. In: Workshop on Portability Among HPC Architectures for Scientific Applications (2015)

16. Jeffers, J., Reinders, J., Sodani, A.: Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition. Morgan Kaufmann (2016)
17. Karpiński, P., McDonald, J.: A high-performance portable abstract interface for explicit SIMD vectorization. In: Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores. ACM (2017)
18. Kim, K., Costa, T., Deveci, M., Bradley, A., Hammond, S., Guney, M., Knepper, S., Story, S., Rajamanickam, S.: Designing vector-friendly compact blas and lapack kernels. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. p. 55. ACM (2017)
19. Kim, K., Hammond, S., Boman, E., Bradley, A.M.and Rajamanickam, S., Deveci, M., Hoemmen, M., Trott, C.: KokkosKernels v. 0.9, Version 00 (2 2017), https://www.osti.gov//servlets/purl/1349511
20. Kretz, M., Lindenstruth, V.: Vc: A C++ library for explicit vectorization. Software: Practice and Experience **42**(11), 1409–1430 (2012)
21. Leißa, R., Hack, S., Wald, I.: Extending a C-like language for portable SIMD programming. ACM SIGPLAN Notices **47**(8), 65–74 (2012)
22. Medina, D., St-Cyr, A., Warburton, T.: OCCA: A unified approach to multithreading languages. arXiv preprint arXiv:1403.0968 (2014)
23. Network, I.P.: Think Exponential: Intel's Xe Architecture. https://itpeernetwork.intel.com/intel-xe-compute/#gs.emsehp (2019)
24. Opencl, K., Munshi, A.: The opencl specification version: 1.0 document revision: 48, 2008. Cited on p. 23
25. P., J., Thornock, J., Smith, S., Smith, P.: Large eddy simulation of polydisperse particles in turbulent coaxial jets using the direct quadrature method of moments. International Journal of Multiphase Flow **63**, 23–38 (2014). https://doi.org/10.1016/j.ijmultiphaseflow.2014.03.002
26. Pai, S., Govindarajan, R., Thazhuthaveetil, M.: PLASMA: Portable programming for SIMD heterogeneous accelerators. In: Workshop on Language, Compiler, and Architecture Support for GPGPU, held in conjunction with HPCA/PPoPP (2010)
27. Phipps, E., D'Elia, M., Edwards, H., Hoemmen, M., Hu, J., Rajamanickam, S.: Embedded ensemble propagation for improving performance, portability, and scalability of uncertainty quantification on emerging computational architectures. SIAM Journal on Scientific Computing **39**(2), C162–C193 (2017)
28. Phipps, E., Tuminaro, R., Miller, C.: Stokhos: Trilinos Tools for Embedded Stochastic-Galerkin Uncertainty Quantification Methods. Tech. rep., Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States) (2008)
29. Stephens, N., Biles, S., Boettcher, M., Eapen, J., Eyole, M., Gabrielli, G., Horsnell, M., Magklis, G., Martinez, A., Premillieu, N., et al.: The ARM scalable vector extension. IEEE Micro **37**(2), 26–39 (2017)
30. Tian, X., Saito, H., Su, E., Lin, J., Guggilla, S., Caballero, D., Masten, M., Savonichev, A., Rice, M., Demikhovsky, E., et al.: LLVM Compiler Implementation for Explicit Parallelization and SIMD Vectorization. In: Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC. p. 4. ACM (2017)
31. Trott, C.R.: Kokkos: The C++ Performance Portability Programming Model. Tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States) (2017)
32. Wang, H., Wu, P., Tanase, I., Serrano, M., Moreira, J.: Simple, portable and fast SIMD intrinsic programming: generic simd library. In: Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing. ACM (2014)

33. Zenker, E., Worpitz, B., Widera, R., Huebl, A., Juckeland, G., Knüpfer, A., Nagel, W.E., Bussmann, M.: Alpaka–An Abstraction Library for Parallel Kernel Acceleration. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 631–640. IEEE (2016)