A Programming System for Xeon Phis with Runtime SIMD Parallelization

Xin Huo, Bin Ren, and Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
huox@cse.ohio-state.edu, ren@cse.ohio-state.edu, agrawal@cse.ohio-state.edu

ABSTRACT

The Intel Xeon Phi offers a promising solution to coprocessing, since it is based on the popular x86 instruction set. However, to fully utilize its potential, applications must be vectorized to leverage the wide SIMD lanes, in addition to effective large-scale shared memory parallelism. Compared to the SIMT execution model on GPGPUs with CUDA or OpenCL, SIMD parallelism with a SSE-like instruction set imposes many restrictions, and has generally not benefitted applications involving branches, irregular accesses, or even reductions in the past. In this paper, we consider the problem of accelerating applications involving different communication patterns on Xeon Phis, with an emphasis on effectively using available SIMD parallelism. We offer an API for both shared memory and SIMD parallelization, and demonstrate its implementation. We use implementations of overloaded functions as a mechanism for providing SIMD code, which is assisted by runtime data reordering and our methods to effectively manage control flow. Our extensive evaluation with 6 popular applications shows large gains over the SIMD parallelization achieved by the production (ICC) compiler, and we even outperform OpenMP for MIMD parallelism.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Single-instruction-stream, multiple-data-stream processors (SIMD)*

Keywords

Xeon Phi; SIMD; API; Communication Patterns

1. INTRODUCTION

Over the last 6-7 years, high-end computing systems have changed significantly with respect to the *intra-node* architectures, with popularity of coprocessors. Over the last 3 years, as many as three of the five fastest supercomputers (at any time, based on the bi-annual top 500 list) in the world involved coprocessors on each node, as they offered excellent performance-price and performance-power ratios.

A recent development along the same lines has been the emergence of Xeon Phi chips, based on the Intel MIC architecture. Xeon Phi is a promising system, because it allows x86 compatible software to be used. Thus, users could potentially continue to use their MPI and/or OpenMP applications, and not have to program in (and learn a) new language like OpenCL or CUDA for the use of accelerators. At the same time, there are many similarities between GPUs and Xeon Phis. Both of these systems have a small amount of memory per thread/core, and moreover, both of them extensively employ a form of SIMD parallelism. NVIDIA GPUs have relied on SIMT (Single Thread Multiple Threads) model. Xeon Phi is built on top of the long-existing Intel SSE (Streaming SIMD Instructions), and particularly, supports IMCI (Initial Many Core Instructions) instruction set for use of SIMD. The SIMD width has been extended to 512 bits (16 floats), potentially offering large benefits for applications.

Use of SSE-like instruction sets has always been a hard problem, and it turns out that such parallelism has not been consistently used for applications outside dense matrix or imaging kernels. Moreover, there are significant programming differences between CUDA and SSE-like instruction sets, since they target SIMT and SIMD models, respectively. Specifically, while coalesced memory accesses are important for performance in SIMT programming, parallelism is still available, whereas programmers need to explicitly create aligned and contiguous accesses in the case of SSE or IMCI. Similarly, while branches are automatically managed in SIMT, with masks internally implemented, programmers or compilers must identify instructions executed by all threads with SSE/IMCI.

Effectively exploiting the power of a coprocessor like Xeon Phi requires that we exploit both MIMD and SIMD parallelism. While the former can be done through Pthreads or OpenMP, it is much harder to extract SIMD performance. This is because the restrictions on the model make hand-parallelization very hard. At the same time, productions compilers are unable to exploit SIMD parallelism for many of the cases.

This paper focuses on the problem of application development on any system that supports both shared memory parallelism and SSE-like SIMD parallelism, with a specific emphasis on the Intel Xeon Phi system. We describe an API and a runtime system that helps extract both shared memory and SIMD parallelism. One of the key ideas in our approach is to exploit the information about underlying communication patterns, to both partition and schedule the computation for MIMD parallelism, and reorganize the data for achieving better SIMD parallelism. While our approach is general, we currently focus on stencil computations, generalized reductions, and irregular reductions.

In the context of SIMD parallelization, though there is large volume of existing work on compiler-based code generation [7, 20, 9, 14], our work is driven by three observations. First, advanced features, like *scatter* and *gather* operations and *masks* need to be exploited for supporting different types of applications with the IMCI instruction sets. Second, increasing width requires that new approaches be exploited, for example considering aggressive inter-iteration parallelism for irregular reductions, unlike the existing work on this topic [14]. Finally, we observe that some of the advances in research prototype compilers have not made it to production-level compilers (as evidenced by our experiments with ICC compiler), and alternative approaches to simplifying SIMD code generation may be needed. Overall, with our approach, it is possible to use SIMD lanes for code

ICS'14, June 10–13 2014, Munich, Germany.

http://dx.doi.org/10.1145/2597652.2597682.

involving irregular accesses, reductions, and control flow, unlike the previous work in this area.

Our work is significant in multiple ways: 1) We provide an *end-to-end* application development system for the Xeon Phi architecture, or more broadly, any system with both shared memory and SIMD parallelism, 2) Our work can be viewed as providing a CUDA or OpenCL-like programming API for SSE-like instructions, where the responsibility for determining contiguous vs. non-contiguous accesses or managing conditionals is the responsibility of the underlying library, 3) we offer potential intermediate language which may be generated by a compiler (for example, systems similar to the ones that generate CUDA code), and subsequently, runtime transformations and libraries be used for SIMD parallelization. Compared to the existing code generation approaches, we can simplify SIMD compilation process and make it more portable.

We have extensively evaluated our framework using six applications, which involve generalized reductions, stencil computations, and irregular reductions. Our evaluations shows: 1) on larger of the two datasets used for each application, the SIMD parallelization speedup from our system ranges from 1.6 to 7.8 (average of 2.8) whereas the corresponding gain from production compiler (ICC) is between 0.95 and 3.5 (average of 1.5), 2) as compared to hand-written IMCI code, the overheads of our framework is negligible, 3) by combining MIMD and SIMD parallelism on Xeon Phi, we achieve a speedup between 33 and 580 over single thread execution, outperforming Pthreads with ICC based vectorization by an average of 1.9x, and 4) we outperform parallelism with OpenMP at both MIMD and SIMD levels, even though we offer a comparable programming API.

2. PARALLELIZATION AND PERFORMANCE ISSUES IN INTEL XEON PHI

2.1 Intel Xeon Phi Architecture

The x86-compatible Intel Xeon Phi coprocessor, which is a latest commercial release of the Intel Many Integrated Core (MIC) architecture, has already been incorporated in 9 of the top 100 supercomputers at the time of writing this paper [1]. MIC is designed to leverage existing x86 experience and benefit from traditional multi-core parallelization programming models, libraries, and tools.

In the available MIC systems, there are 60 or 61 x86 cores organized with shared memory. These cores are low frequency in-order ones, and each supports as many as 4 hardware threads. Additionally, there are 32 512-bit vector registers on each core for SIMD operations. The main memory sizes vary from 8 GB to 16 GB, and the memory is shared by all cores. The L1 cache is 32 KB, entirely local to each core, whereas each core has a coherent L2 cache, 512 KB, where cache for different cores are interconnected in a ring.

Our work focuses on three important features of Intel MIC architecture, which need to be exploited for obtaining high performance:

Wide SIMD Registers and Vector Processing Units (VPU): VPU has been treated as the most significant feature of Xeon Phi by many previous studies [16, 27, 21, 6]. The reason is that the Intel Xeon Phi coprocessor has doubled the SIMD lane width compared to Intel Xeon processor, i.e., 256-bit to 512-bit, which means that it is possible to process 16 (8) identical floating point (double precision) operations at the same time. In addition, we have a new 512-bit SIMD instruction set called Intel Initial Many Core Instructions (Intel IMCI), which has built-in *gather* and *scatter* operations that allow irregular memory accesses, a hardware supported *mask* data type, and *write-mask* operations that allow operating on some specific elements within the same SIMD register. Even though all of these new instructions could potentially be simulated by the programers in the SIMD Streaming Extension (SSE) model, explicit new instructions allow easier implementation of more irregular parallelism. Note that SIMD instructions can be generated by the ICC compiler through the *auto-vectorization* option, or the programmers could use IMCI instruction set directly. The former needs low programming effort, though current compilation systems have several limitations and do not always obtain high performance. In comparison, the latter option can achieve the best performance, however, is tedious and error prone, and creates non-portable code.

Large Number of Concurrent Threads: Each Xeon Phi core allows up to 4 hyper-threads, in another word, we can have as many as 240/244 hardware threads sharing the same memory on Xeon Phi. This provides us with massive Multiple Instruction Multiple Data (MIMD) parallelism with shared memory, which has not been common in the past.

Coherent Distributed L2 Cache: Intel Xeon Phi architecture uses coherent L2 Cache with ring interconnection. When a L2 cache miss occurs for a specific core, an address request is sent to the ring. If the address is found in another core's L2 cache, the corresponding data is forwarded back along the ring. In worst case, the entire process may take hundreds of clock cycles. Thus, Xeon Phi reduces the number of L2 cache misses, but even an L2 cache hit can be very expensive. Thus, data locality is crucial for the overall performance.

2.2 Our Approach

Our approach for providing a solution for application development on Xeon Phi systems, including SIMD parallelization, is based on the observation that most applications follow a small number of *patterns* or *dwarfs* (e.g. as summarized by Collela and also described in Berkeley landscape on parallel computing [3]). By exploiting knowledge of individual patterns, needed data transformations and partitioning approaches can be used. Indeed, many previous efforts on SIMD (and SIMT) parallelization have focused specifically on particular patterns, like stencil computations [5, 9, 8] or irregular reductions [11, 14, 29].

We focus on a more general framework for specifying the computations, but where underlying patterns are explicitly known and exploited. Though the idea can be applied to a variety of patterns, we focus on stencil computations, generalized reductions, and irregular reductions in this paper. Among these, stencil computations and generalized reductions are well understood. As a background for our presentation, we show an example of an irregular reduction here.

```
Real    X(num_nodes), Y(num_edges) ;   {* data arrays *}
Integer IA(num_edges,2) ;              {* indirection array *}

for(i = 0; i < num_edges; i++) {
      X(IA(i,1)) = X(IA(i,1)) + Y(i) ;
      X(IA(i,2)) = X(IA(i,2)) - Y(i) ;
}
```

Figure 1: A simple loop involving indirection

A typical irregular reduction is shown in Figure 1. In iteration i of the loop, the code makes two indirect references to the array X using $IA(i, 1)$ and $IA(i, 2)$. Codes from many important scientific and engineering domains contain loops with such indirection array sections. When a problem is modeled using an unstructured grid, a list of edges (with the nodes they connect) is explicitly stored. A computation that iterates over all edges in the grid and updates the attributes associated with the two end-points of the edge will have structure similar to the code in Figure 1. However, such codes can arise in other contexts - for instance, molecular dynamics contains similar loops, as the nodes represent molecules, and the edges denote the interaction between a pair of molecules.

2.3 Challenges and Opportunities

There are two levels of parallelism one can seek on the Xeon Phi: MIMD parallelism supported by large number of hyper-threads, and SIMD parallelism provided by the wide VPU. There are challenges associated with each of them, as well as opportunities to exploit information from specific communication patterns. The issues for applications with different types of patterns are summarized in Table 1.

2.3.1 MIMD Parallelization Issues

A Xeon Phi can be viewed as a SMP machine, in which all the cores not only share the same memory address, but also a coherent

Com Pattern	MIMD Challenge	SIMD Challenge
Generalized Reduction	job partition	unaligned/non-unit-stride access control flow dependency data dependency/conflicts
Stencil Computation	job partition	unaligned memory access control flow dependency
Irregular Reduction	job partition load balance	unaligned/random memory access control flow dependency data dependency/conflicts

Table 1: Parallelization Challenges of Different Communication Patterns

cache space. Thus, the traditional MIMD parallelization methods, like OpenMP, can also be applied with the support of the Intel compiler. Yet, there are many opportunities for exploiting information about specific communication patterns.

Particularly, applications with different communication patterns usually have different requirements on task partitioning and scheduling. For stencil computation and generalized reductions, static scheduling could provide better performance, since it can achieve load balance with a small scheduling overhead. For irregular reductions, a technique like the *reduction space partitioning* [11] can be used to avoid conflicts between the threads. Moreover, dynamic, fine-grained, scheduling could achieve better performance over static scheduling by achieving better load balance.

Communication pattern specific information can also help in other ways. Data reorganization is one of the optimizations to support vectorization, but data reordering can also provide better cache locality for irregular reductions. These optimizations are normally not performed by a more general framework, such as an OpenMP implementation.

2.3.2 SIMD Parallelization Issues

In SIMD execution, one memory access operation can load (store) multiple data elements simultaneously from (to) the memory. However, there are strict restrictions on how and when such operations can be applied.

Unaligned/Non-unit Stride Accesses: For using SIMD parallelism, the start of the read or write memory address has to be 64 bytes aligned on Xeon Phi. But, it is difficult to satisfy this requirement for almost any kind of application. For instance, stencil computation usually needs to access one node's neighbors in different directions. In a one dimension matrix, if the address of matrix[i] is aligned by 64 bytes, addresses of its neighbors, matrix[i-1] and matrix[i+1], will not be aligned. Similar problems will also arise for a matrix with more dimensions. In addition, different SIMD lanes can only access continuous memory address. Thus, accesses of elements from an array of structures or data accessed through indirection arrays cannot exploit SIMD parallelism directly.

Control Flow Dependencies: At any time, all the SIMD lanes have to execute the same instructions on different data elements. However, in the different branches of an *if-else* clause, different lanes may execute different instructions, which is not supported by SIMD. This kind of control flow arises very commonly in generalized reduction and irregular reductions.

Data Dependencies and Conflicts: When different SIMD lanes try to write to the same location, the behavior is undefined, as there is no locking operation. In the case of both generalized reductions and irregular reductions, such write conflicts arise. Thus, how to solve the data dependencies and conflicts for SIMD effectively and efficiently is another challenge.

3. API FOR APPLICATION DEVELOPMENT ON XEON PHI

Our parallelization framework provides a set of user API. Next, we introduce our MIMD and SIMD API, and then show how to use it in a variety of sample kernels.

User Interface API (class Task)	
API	**Descriptions**
struct Configuration	Configuration of the Task size, offset, and accessing stride.
enum Patterns	Declare the communication patter(Generalized Reduction, Stencil, and Irregular Reduction).
tuple</*Parameter Lists*/>Parameters	Define the input parameters for a specific application.
void Kernel(vector<int> &index)	The kernel function provides the computing logic for a single data, given by the index vector.
MIMD Parallel Framework API (class MIMD_Framework)	
API	**Descriptions**
void run(Task &task)	The run function has the capability of register the user defined task to MIMD framework, invoking runtime optimizations, task partitioning, and scheduling on MIC architecture.
void join()	It will block, until the execution on MIC is finished.

Table 2: User Interface and MIMD Parallel Framework API

3.1 Shared Memory (MIMD) API

MIMD parallelization API is shown in Table 2. The first four parameters correspond to a *Task class*, which has four attributes, *Configuration*, *Pattern*, *Parameters*, and the *Kernel* function. The *Configuration* comprises three vector type variables, representing the size, offset, and stride of the computation space across different dimensions. *Pattern* is used to indicate which communication pattern the given task belongs to. Based on the pattern information, MIMD parallelization framework applies different partitioning methods, and this information is used by the SIMD parallel framework as well. In addition, users need to define the *Parameters* types, which includes the input and output parameters for a specific application. The most important part in the user interface is the *Kernel* function, which gives the smallest computation logic on one data element. It has only one input parameter representing the index of the target data element. Moreover, users need to guarantee that the kernel function is independent between different input indices. The independency can be achieved by either replicating the shared writing data or using locks while updating.

The last two API are related to the execution and optimization of the applications. The *run* function receives a user defined *task*, with a specification of the four set of parameters, and automatically invokes runtime optimizations, including partitioning and scheduling methods, for parallel execution on the Xeon Phi. The strategies employed in partitioning, scheduling, and optimizations are based on the parameters from the user interface. We will elaborate it in detail in Section 4. After these preprocessing, run function will launch a group of threads, each of which executes the kernel function with different input indices. The *run* function is a non-blocking function, which will return immediately after launching a job. Next, the users can call the *join* function to wait, until the execution of the *task* finishes.

Overall, our MIMD API provides a way to port applications to the Xeon Phi architecture with a very small efforts on part of the users. After giving a *task* definition, users can call *run(Task)* directly to execute the target applications.

3.2 SIMD API

The main idea of our SIMD API is to express collections of data elements on which parallel operations can be applied. The actual layout and scheduling of the operations is left up to the runtime system.

Before introducing the API for operations, we first introduce the definition of the new data types. We introduce three data types in SIMD API, which are shown in Table 3. *Scalar Type* is the basic data type, which only contain one data element - the implication is that if this variable is involved in a SIMD operation, it will be shared

Data Type	Name	Description
Scalar Type	int, float, double, ...	Data is shared by all the SIMD lanes. All the basic data types or temporary variables are belonged to shared type.
Vector Type	vint, vfloat, vdouble, ...	It includes multiple data scaling to all the SIMD lanes.
Mask Type	mask	It helps handling control flow in vectorization

Table 3: The data types defined in SIMD API

vint v1, v2; int s; mask m; **op represents the supported mathematic or logic operations;**	
API	**Examples**
Assignment API	
Support assignment between vector types, and scalar type to vector type.	v1 = v2; v1 = s; v1 op= v2; v1 op= s;
Mathematic API	
Support most mathematic operations, including +, -, *, /, %, between vector types and scalar types.	v1 = v2 op v1; v1 = v2 op s;
Logic API	
Support most logic operations, including ==, !=, <, >, <=, >=, between vector types and scalar types. Return type is mask type.	m = v1 op v2; m = v1 op s;
Load/Store API	
void load(void *src);	v1.load(addr);
void store(void *dst);	v1.store(addr);
void load(void *src, const vint &index, int scale)	v1.load(addr,index,scale);
void store(void *dst, const vint &index, int scale)	v1.store(addr,index,scale);
Generalized Reduction API	
template<class ReducComp = reducAdd > void reduction(int *update, int scale, int offset, vint *index, type value, [mask m])	reduction(update, scale, offset, index, v1);
Mask API	
mask()	v1.mask()
Mask_State Object	
Members	**Descriptions**
mask m	mask type variable
type old_val	the default value for unset vector lanes
set_mask(const mask &m, type &old_val);	set mask and default value
void clear_mask();	clear default value and set all vector lanes to active

Table 4: SIMD API

by all the SIMD lanes. In contrast, *Vector Type*, which is represented as *vint or vfloat*, includes an array of data elements. Thus, if we declare one array as *Vector Type*, each time SIMD lanes will access a group of contiguous data elements. However, when SIMD lanes access a *Scalar Type*, the same data element will be automatically duplicated for all the lanes. This automatic duplication is supported by the implicit conversion from *Scalar Type* to *Vector Type* in our implementation.

The last data type is the *Mask Type*. Because SIMD vectorization does not support control flow, we require use of a mask variable to express what computations are applied on which elements. The mask type is implemented as a bit set, in which each bit represents one vector lane. Two values, 1 and 0, represent set and unset, respectively.

The supported operations on different data types are shown in Table 4. The main idea is to *overload* most operators on vector types, or even operations involving one vector type and a scalar type. Thus, the difference between the serial codes and the vectorized codes by using our API is quite small, as we will show through several examples. As shown in Table 4, for assignments and mathematical operations, users can use the same operator in serial codes for vector types and a combination of vector and scalar types. The overloaded operator implementation will automatically perform vectorization on the input parameters. For logic operations, the difference from the traditional logic operators is with respect to the return type. Because there is no support for control flow in SIMD, in the logic operation API, the return type is the mask type, which is then used to express the conditional clause that will be applied for a particular element.

Moving onto the rest of the API, there are two types of load and store functions, which are for reading and writing contiguous and non-contiguous addresses, respectively. A load (store) with a single source or destination parameter provide the function of read and write between the vector type and a contiguous memory address space. On the other hand, a function with the extra *index* and *scale* parameters helps exploit *gather* and *scatter* operations in the IMCI instruction set for non-contiguous memory accessing. For the applications, which data reorganization can be applied, such as generalized reductions and stencil computations, there is no need for non-contiguous load and store API. However, for irregular applications, in which data reorganization cannot eliminate indirect memory accessing, non-contiguous load and store API can provide an alternative way.

One specific feature is a reduction function. As we had stated before, multiple SIMD lanes cannot update the same element, and as a result, implementation of a reduction function using SIMD instructions is more complex. The specific reduction function is given as a parameter in the template. Our runtime system ensures that SIMD lanes are correctly used for such computation.

The goal of the mask function is the conversion of a unmask vector type to the mask vector type. After this conversion, all the operations on this collection start using the *mask_state object* to determine which elements an operation is applied to. Function *set_mask* is used to setup the mask for current *mask_state object* on one thread, which is then used till it is cleared or updated.

3.3 Sample Kernels

We now illustrate the API using functions involving different communication patterns. We establish how code using our API is similar to sequential code, and much simpler than a hand-written vectorized code.

3.3.1 Stencil Application

Listing 1: Sobel: Stencil Computation with serial codes

```
void kernel(int i, int j){
    float Dx = 0.0, Dy = 0.0;
    for(int p = -1; p <= 1; p++){
        for(int q = -1; q <= 1; q++){
            Dx += weight_H[p+1][q+1]*b[i+p][j+q];
            Dy += weight_V[p+1][q+1]*b[i+p][j+q];
        }
    }
    float z = sqrt(Dx*Dx + Dy*Dy);
    a[i][j] = z;
}
```

Listing 2: Sobel: Stencil Computation with SIMD API

```
void kernel(int i, int j){
    vfloat Dx = 0.0, Dy = 0.0;
    //Compute the weight for a node in a 3x3 area
    for(int p = -1; p <= 1; p++){
        for(int q = -1; q <= 1; q++){
            Dx += weight_H[p+1][q+1]*b[X(i,p,q)][Y(j,p,q)];
            Dy += weight_V[p+1][q+1]*b[X(i,p,q)][Y(j,p,q)];
        }
    }
    vfloat z = sqrt(Dx*Dx + Dy*Dy);
    z.store(&a[i][j]);
}
```

Listing 3: Sobel: Stencil Computation with manual vectorization

```
void kernel(int i, int j){
    __m512 Dx = _mm_set1_ps(0.0), Dy = _mm_set1_ps(0.0);
    //Compute the weight for a node in a 3x3 area
    for(int p = -1; p <=1; ++p){
        for(int q = -1; q <=1; ++q){
            __m512 *tmp = (__m512*)&b[i+q][j+p*vec_width];
            __m512 tmpx = _mm512_mul_ps(*tmp,weight_H[p+1][q+1]);
            Dx = _mm512_add_ps(Dx, tmpx);
            __m512 tmpy = _mm512_mul_ps(*tmp,weight_V[p+1][q+1]);
            Dy = _mm512_add_ps(Dy, tmpy);
        }
    }
    __m512 sqDX = _mm512_mul_ps(Dx, Dx);
    __m512 sqDy = _mm512_mul_ps(Dy, Dy);
    __m512 ret = _mm512_add_ps(sqDx, sqDy);
    ret = _mm512_sqrt_ps(ret);
    _mm512_store_ps(&a[i][j], ret);
}
```

In Listing 1, 2, and 3, we take a simple stencil computation, the sobel filter, and compare serial, vectorized using our API, and manually vectorized versions.

Comparing between Listing 1 and 2, the vectorized codes in our API are almost as same as the serial version, except new vector types (*vfloat*) are introduced to replace the original scalar types (*float*). Another difference is that the assignment from vector type to scalar type is achieved through the *store* API, because it needs to involve multiple data copies from the vector variable to the target memory locations. Also, to facilitate a possible data reorganization at runtime, a function *Dim(idx, offset1, offset2, . . .)* is provided to calculate the transformed index in each dimension by applying offsets on different dimensions. For example, in Listing 2, $X(i, p, q)$ calculates the transformed index in the X-dimension when applying p and q offsets on the original X and Y dimensions, respectively.

To summarize, our API provide a convenient way to achieve vectorization with very little modification on the serial code. It is also clear that the manual vectorization codes, shown in Listing 3, introduces more new Intel IMCI API, and is much more complicated compared to serial and our SIMD API versions, as about 40% extra lines are added.

3.3.2 Generalized Reduction

Listing 4: Kmeans: Generalized Reduction with SIMD API

```
void kernel(vfloat *data, int i){
    vfloat min = FLT_MAX;
    vint min_index = 0;
    for(int j = 0; j < k; ++j){
        //step 1 (Computation): compute the distance
        vfloat dis = 0.0;
        for(int m = 0; m < 3; ++m){
            dis += (data[i+m*n]-cluster[j*3+m])*
            (data[i+m*n]-cluster[j*3+m]);
        }
        dis = sqrt(dis);
        //step 2 (Control flow): update index
        mask m = dis < min;
        set_mask(m, min);
        min.mask() = dis.mask();
        set_mask(m, min_index);
        min_index.mask() = j;
    }
    //step 3 (Reduction): reduction
    reduction(update, 5, min_index, 0, data[i]);
    reduction(update, 5, min_index, 1, data[i+n]);
    reduction(update, 5, min_index, 2, data[i+n*2]);
    reduction(update, 5, min_index, 3, 1.0);
    reduction(update, 5, min_index, 4, min);
}
```

In Listing 4, we show the main function of Kmeans, a simple data mining kernel, with vectorization by using our API. The procedures of Kmeans can be divided into three steps: 1) compute the distance between one node and the candidate clusters, 2) update index to the cluster with the minimum distance, 3) do reduction on the cluster found in the step 2. Thus, the step 1 is only simple arithmetic operations, whereas steps 2 and 3 involve control flow and generalized reduction, respectively.

In the step 1, similar to the stencil computation, the only modification is the data types of corresponding variables are changed from scalar type to the vector type. As a result, the computation is automatically vectorized by loading values from the *data* array to all vector lanes, and computing the distance between the data in each lane and the clusters. The step 2 introduces a branch, specifically, if the distance is smaller than the current minimum distance (min), we update the min and min_index, otherwise, min and min_index are not changed. Using our mask API, we represent this computation as *if-else* branch, in which the *else* branch just assigns its own value to itself. As we can see in step 2 of the Listing 4, a mask variable m is returned by the logic computation. Then, m and the default value for *else* branch are set by *set_mask* function. Next, the *mask()* function will do the conversion from unmask vector type to the mask vector type. In the step 3, reduction is performed on the array *update* with the add operation. It is not safe to perform the reduction using the general arithmetic and assignment operations, due to the potential written conflict between different vector lane. Thus, we use the API for reduction. Here, *add* operation, which is the default reduce operation for reduction function, is used to reduce values to the array *update*.

To summarize, in our API, the code with arithmetic operations is almost as same as the original (serial) code. The reduction in our API is provided through a function interface, which allows us to vectorize these codes, whereas most compile-time solutions fail to do this. The most complicated part of our API is handling of control flow, where branches are replaced by mask operations. However, we note that existing vectorizing compilers do not handle control flow at all (as we will show through experimental results), and manual vectorization in presence of control flow is very complicated (please see an example in Figure 4).

4. RUNTIME SCHEDULING FRAMEWORK

We now describe the implementation of the framework, and particularly, how runtime scheduling that is applied for both MIMD and SIMD parallelization.

4.1 MIMD Parallelization

Though MIMD parallelization is performed by a number of existing frameworks, our focus is on providing automatic or guided task partitioning and scheduling for three different communication patters (generalized reduction, stencil computation and irregular reduction) on the Xeon Phi. In each of these patterns, the computation is an iteration over a set of *indices*, where the following two steps are applied on each index: Step 1 - Loading the index of the targeted data and other auxiliary data for computation, and Step 2 - Executing the computation logic, including both computation and writing results, for the target data. In MIMD parallelization, each thread will load a different index in the Step 1, and execute Step 2 simultaneously and independently, except for handling possible race conditions on the output elements. Our API allows the user to provide a *task* function, which is the serial code for computations associated with a single target data element, which is used for Step 2.

Our runtime system has two major components, task partitioning and runtime scheduling, to parallelize the target applications on the Xeon Phi. Task partitioning can potentially be applied on the *computation space* or the *reduction space*. *Computation space* refers to the space of the computation loop. For example, the num_edges loop in Figure 1 is belonged to *computation space*. *Reduction space* refers to the space in which a reduction is executed. The X array in Figure 1 is an example of reduction space for this loop. For generalized reductions and stencil computations, it is straightforward to perform task partitioning on the computation space. Particularly, the task partitioning component can just divide the computation loop into a number of blocks with an equal size, and pass these blocks to the dynamic scheduling component to execute.

However, for irregular reduction applications, there is a tradeoff between computation space partitioning and reduction space partitioning [11]. Briefly, computation space can introduce significant overhead on locking operations when different threads trying to update the results on the same reduction index, whereas, reduction space partitioning can completely avoid competition between threads

by assigning different reduction space to different threads. Thus, all threads can execute independently by updating non-overlapping parts of the reduction space. Thus, in our framework, different task partitioning strategies will be launched based on the types of the applications, provided by the users.

In the runtime scheduling component, we include three scheduling methods. The first is the static scheduling, in which all the tasks from task partitioning module will be equally distributed to the all the available threads. The static method introduces the smallest scheduling overhead, and for stencil computations, this scheduling method achieves better performance, because it can still achieve good load balance. However, for generalized reductions and irregular reductions, the workload in each task partition may be different. Especially, for an irregular reduction, after reduction space partitioning, the workload in each partition may be quite different, and depends on the number of edges associated with each node in the reduction space. Thus, a dynamic scheduling method based on factoring is provided in our framework, which assigns large number of tasks to the threads at first, and reduces the number of assigned tasks as execution progresses. The third and the final scheduling method is the *user-defined* method, where a user can define the number of tasks in each partition.

4.2 SIMD Parallelization Support

Our SIMD parallelization support has three components: implementation of overloaded functions which supports SIMD execution, runtime data reorganization, and handling of control flow.

4.2.1 SIMD Parallelization Through Implementation of Overloaded Functions

```
int func(vfloat *a, vfloat *b, vfloat *c){
    for(int i = 0; i < n; ++i)
        c[i] = a[i] + b[i];
}
```

(a) An vectorized function by using overloaded functions

```
int func(float *a, float *b, float *c){
    for(int i = 0; i < n; i+=16){
        __mm512 *s_a = (__mm512*)a[i];
        __mm512 *s_b = (__mm512*)b[i];
        __mm512 *s_c = (__mm512*)c[i];
        *s_c = _mm512_add_ps(*s_a, *s_b);
    }
}
```

(b) The expansion of the overloaded functions in (a)

Figure 2: An example of vectorization in overloaded functions

Our primary method for auto vectorization is based on the implementation of the overloaded functions we had listed in Table 4. The basic idea is as follows - overloaded functions are used inside definition of a *task*, which applies computation to a particular point. Since these computations can be applied in parallel, an overloaded function's implementation uses SIMD instructions to achieve parallel execution.

An example is shown in Figure 2. The initial function performs an add operation between arrays a and b, and writes the results to the array c. The sub-figure (a) shows the code with our API, which is the sequential code, except for *vector-type* declarations. SIMD parallelization is now automatically applied based on the overloaded add operator on the vector types. Sub-figure (b) shows the expansion of the overloaded function. First, it applies a translation between the scalar type and the vector types on the arrays involved. Then, a SIMD add function is called on the translated arrays to perform 16 add and write operations in the SIMD manner. Next, the index is moved to the start of the next 16 operands.

Overall, unlike hand-code SIMD parallelization, our framework uses numerous overloaded functions to provide a convenient way to achieve the same performance. There is no need for application developers to consider address translation, different vectorization instructions for different operand types, operations, and architectures,

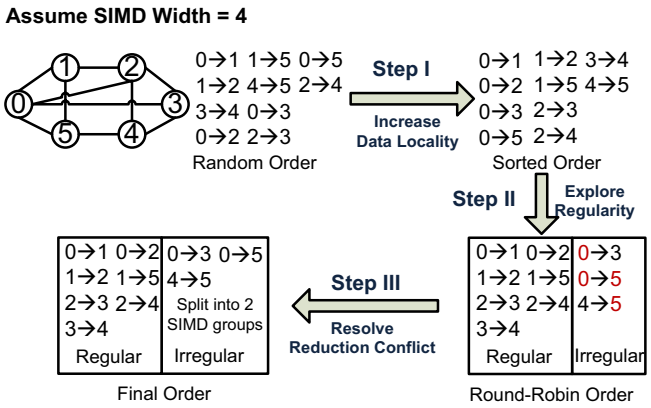

Figure 3: Irregular Reduction Edges Reorder

and the position of the next computing index. However, in practice, there are many complications in the application of overloaded functions, particularly, when data elements are not contiguous, and/or when branches are involved. We discuss these issues in the rest of this section.

4.2.2 Data Reorganization

SIMD operations on Xeon Phis (or any SSE-like instruction set) can only be applied if there are continuous and aligned memory access. Many applications have non-unit stride and unaligned or even random memory accesses. Such kind of accesses impede compiler vectorization. In our framework, we exploit the knowledge about underlying communication patterns to reorganize the data and facilitate SIMD parallelization.

Generalized Reductions: For generalization reductions, data array is usually given as an *Array of Structures* (AoS). For instance, in Kmeans, the input point array comprises x, y, and z dimensional information (for three dimensional-points). During vectorization, the Vector Processing Unit (VPU) will apply the same operation for 16 elements, which means that the VPU needs to access 16 continuous values from each dimension. However, with AoS storage, values corresponding to one particular attribute are non-contiguous. Moreover, if the x dimension data is stored aligned, y and z dimension, data is likely to be unaligned. Therefore, both non-continuous and unaligned memory accesses can either prevent vectorization or impact vectorization performance negatively by the compiler or programing having to introduce extra gather and scatter operations.

Our system applies the standard AoS to SoA *Structure of Array* transformation. In the SoA format, instead storing each member of the structure continuously, all values for a particular member are grouped together. Thus, accesses to the same member will be continuous. Moreover, aligned accesses can be ensured by adding padding at the end of each member array. Because AoS can be viewed as a matrix, in which columns represent different members of the structure, our framework employs a parallel matrix transpose to apply the transformation efficiently.

Stencil Computations: Unaligned memory accesses is the major problem for vectorization on stencil computations. When computing the value for a target point, we need to access all its neighbors. Thus, in the original format, it is impossible to ensure that the target point and its neighbors are both aligned. In the literature [8], a non-linear data layout transformation has been proposed to make the target point and its neighbors aligned at the same time, achieved by *dimension lifting* and a *matrix transposition*. In our framework, we invoke this data reorganization to be able to achieve aligned memory accesses.

Irregular Reductions: In an irregular reduction kernel, indirect data references can cause very random memory accesses. If we want to vectorize these operations, a large number of *gather* and *scatter* operations must be invoked. There are many existing efforts trying to solve or alleviate this problem from different perspectives. Kim and

Han [14] design an algorithm to replace unnecessary gather and scatter operations by scalar operations. Wu *et al.* [28] try to resolve a very similar problem, coalesced memory access, within the context of the GPU architecture. Focusing on inter-iteration parallelism on an irregular reduction for a SSE-like instruction set, we address this problem by a novel computation (edges data) reordering method, which we describe below.

Our method is explained with the help of an example shown in Figure 3. First, the motivation for our method is as follows. The gather and scatter operations incur a very long latency when the data locality is poor, because each gather and scatter operation works at the unit of the entire cache line. For example, when the required data is split across multiple cache lines, we need multiple gather operations to load them. So, the first objective of the our data reorganization method is to reorder the edge data, and increase data locality. To achieve this objective, based on the partitioning algorithm that is used for task partitioning at the MIMD level, we further reorder the *edge* data according to their first nodes (Step I in Figure 3). As a result, at least for one of the end-points of the edge, data is likely to be in the same cache line.

The second objective is to replace *gather* and *scatter* operations by normal SIMD *load* and *store* operations to the extent possible. To achieve this objective, we partition the edges into *regular partitions* and *irregular partitions*, as explained below. First, we further reorder the *edges* data (Step II in Figure 3), so that the edges are ordered in a round-robin manner according to their first nodes, and we have a consecutive set of first node for the set of edges that will be processed in one SIMD step (a *regular partition*). Now, clearly, given a set of edges, we cannot ensure that we can simply reorganize them as a set of regular partitions. A set of edges that will be processed in one SIMD step but whose first nodes do not form a consecutive set is an *irregular partition*. Thus, we will likely have a set of regular partitions and irregular partitions. After this, we can further apply AoS to SoA to duplicate all the first nodes of edges in the regular partition. In such case, we can apply normal SIMD *load* and *store* operations for the first nodes of edges in the regular partition, and only apply *gather* and *scatter* operations for the remaining nodes.

The third objective is to resolve write conflicts within the same SIMD register for the second nodes of edges in a regular partition and for all the nodes in an irregular partition. Note that this issue arises for generalized reductions as well. The problem is that unless we are careful, different SIMD lanes may update the same element of the SIMD register, causing a race condition. A larger SIMD width increases this possibility, and moreover, indirect accesses can make it hard to avoid such situations. In order to resolve this problem, we have two options: a) serialized reduction; and b) further data/-computation reorder. For serialized reduction, we provides a way to automatically serialize all the reduction operations to eliminate the possible conflicts. Alternatively, we can further reorder the elements into blocks according to the SIMD width, even introducing *bubble* elements. For irregular reductions, we can further reorder the edges (computation order) as shown in Step III of Figure 3, by which, we can make sure there is no write conflict within the same SIMD register.

4.2.3 Resolving Control Flow

Control flow (presence of branches) has been a severe impediment to SIMD parallelization, from the time of the initial version of SSE released decades ago. Without hardware support, in SSE and AVX, one has to *simulate* the *mask* operations, which is cumbersome. We show an example in Figure 4, where sub-figure(b) shows hand-written SIMD parallelization, where a set of tasks are applying the code in sub-figure (a). It is easy to see that a statement with simple control flow leads to very complex and the size of the code is increased dramatically.

Our framework helps manage this complexity, building on top of the *mask* data type and *mask* operations in latest Xeon Phi (and the IMCI instruction set). As shown in Figure 4 (c), logic operations between the vector variables can return a mask type variable, and we can use the mask variable as part of the *mask* arithmetic operations

```
if(a < b) a += b;
else a -= b;
```
(a) An example of control follow

```
__mm128i mask1 = _mm_cmplt_epi32(a, b);
__mm128i mask0 = _mm_andnot_si128(mask1,
_mm_set1_epi32(0xffffffff));
__mm128i res = _mm_and_si128(_mm_add_epi32(a, b), mask1);
__mm128i oldval = _mm_and_si128(a, mask0);
a = _mm_or_si128(res, oldval);
res = _mm_and_si128(_mm_sub_epi32(a, b), mask0);
oldval = _mm_and_si128(a, mask1);
a = _mm_or_si128(res, oldval);
```
(b) The vectorization code of control follow in (a)

```
__mmask16 mask1 = _mm512_cmplt_epi32(a, b);
__mmask16 mask0 = _mm512_cmpge_epi32(a, b);
a = _mm512_mask_add_epi32(a, mask1, a, b);
a = _mm512_mask_sub_epi32(a, mask0, a, b);
```
(c) The vectorization code of control follow with mask type in (a)

Figure 4: An example of control follow (a) without vectorization (b) with vectorization (c) with vectorization and mask type

to get results from different branches. Thus, compared to the code in sub-figure (b), control flow can be handled in a more concise fashion. However, users are still required to be familiar with the new intrinsics, which is still complicated and error prone. This is addressed in our framework.

Initially, we further elaborate on the available *Mask_State Object*, shown in Table 4. There are two members, a mask type *m*, and a scalar or vector type *old_val*. *old_val* is the default value assigned to the *unset* or *inactive* SIMD lanes. The idea is that the inactive SIMD lanes still need to execute the instruction, but they simply produce *old_val*. So, as one can see from Figure 4 (c), we set a as the *old_val*. Thus, when executing $a+ = b$, for the lanes in which a is greater than or equal to b, the *old_val* or a, will be assigned to itself in the end. Similar operations occur when executing $a- = b$.

The API we support is simpler, and was summarized towards the end of Tables 3 and 4. The key thing to note is that interface of the operations that involve a mask is same as the operations that do not involve any mask. Each thread owns a local mask_state object for its entire life. The mask_state object is declared as a global and static variable for each thread. Each mask_state object includes the information about the *mask* and *old_val* for current control flow. After users set a mask_state object by the *set_mask* function (example shown in Listing 4), it is effective, until a new *mask* is set, or the current mask is cleared (using the *clear_mask* function). Each thread's mask information is used to provide two implementations of each overloaded operation: *unmask* (default) and with *mask*. If a mask is set, versions with mask are invoked, and use the thread's local mask_state object as a parameter. Listing 4 includes an example of how to translate unmask vector type to mask vector type.

5. EVALUATION

In this section, we evaluate our framework using various applications that involve the communication patterns we have focused on. The objectives of our experiments were: 1) Comparing the performance of applications developed using framework, over hand-written parallel versions (using Pthreads), and evaluating the SIMD parallelization in our framework, over the ICC compiler generated SIMD code, 2) Quantifying the overheads of our runtime framework, by comparing performance against the hand-written SIMD code for SIMD parallelization, 3) Comparing the performance of MIMD parallelization from our framework against OpenMP, another high-level framework, and further evaluating the SIMD parallelization by our framework against what is achieved by ICC compiler with OpenMP directives. All experiments were conducted on a Xeon Phi SE10P card, which has 61 cores each running at 1.1 GHz, with four hyper-

threads per core, along with a 32 MB L2 cache and 8 GB GDDR5 memory. The compiler that we used is Intel ICC compiler 13.1.0. All benchmarks are compiled with the *-O3* optimization. Compiler vectorization is turned on and off by *-vec* and *-no-vec* options, respectively. *#pragma vector always* was always used with OpenMP. We also attempted SIMD pragmas, such as *#pragma ivdep* and *#pragma simd*, as well as the SIMD Pragmas introduced by OpenMP 4.0. However, none of them could handle irregular and some generalized reductions, due to data dependencies, complicated deferences, and the need for outer loop vectorization for our target class of applications. All experiments are conducted in the *Native Model* with the *-mmic* option.

5.1 Benchmarks

Six benchmarks are selected from various benchmark suites, two each involving generalized reductions, stencil computations, and irregular reductions. **Kmeans** [12] is a very popular data clustering kernel - in each iteration, it processes each point in the dataset, determines the closest center to this point, and computes how this center's location should be updated. Our experiments used two different values of the parameter K, the number of clusters, $K = 10$ and $K = 100$. **Naive Bayes Classifier (NBC)** [25] is a simple classification algorithm based on observed probabilities. We used two datasets, 50 MB and 200 MB. **Sobel** is a stencil computation. For 2D Sobel, two 3×3 *weight templates* are used to compute the weight on the target point. Two matrices, with the size of 8192×8192 and 16384×16384, respectively, were used. **Heat3D** [2] simulates heat transmission in a 3D space, involving a 7-point stencil. The small and large datasets used in the experiments are $512 \times 256 \times 256$ and $512 \times 512 \times 512$, respectively. **Molecular Dynamics (MD)** is an irregular reduction kernel used to study the structural, equilibrium, and dynamic properties of molecules. The simulation iterates over all the edges, and updates the attributes associate with the two end nodes. The small dataset used in the experiments has 16K nodes and 2M edges, while the large one has 256K nodes and 32M edges. **Euler** is another irregular reduction kernel based on Computational Fluid Dynamics (CFD) that takes description of the connectivity of a mesh and calculates quantities like velocities ate each mesh point. The small dataset used in our experiments has 182K nodes and 1.13M edges, while the large one has 1.4M nodes and 8.9M edges.

5.2 Speedups from Our Framework

Our first set of experiments focused on comparing the SIMD parallelization with our framework against compiler generated SIMD code (auto-vectorization), and hand-written SIMD code. Compiler SIMD parallelization was applied on Pthreads code, so as to also allow shared memory parallelization. Pthreads-based shared memory parallel versions used similar style (and thus obtain similar performance) as the shared memory parallelization supported by our framework, though the programmer effort is much smaller with our framework. In Figure 5, we compared the best performance between the pthread versions with and without compiler vectorization, and the vectorization versions with hand-coding and our API, for *small* and *large* datasets described earlier. The performance, shown in Figure 5, is the one with the number of threads that leads to the best performance (which maybe different across versions). The numbers reported are relative speedups, with baseline of Pthreads version without vectorization.

For generalized reductions, both Kmeans and NBC show similar trends. The SIMD-API version achieves better performance compared to the Pthread versions with or without compiler generated SIMD code. Moreover, the runtime overhead introduced by SIMD-API is very small compared to the hand-written SIMD versions. In Kmeans, compiler vectorization can only be applied in the innermost loop, which is the loop calculating the distance between one node and all the cluster centers. The performance is sensitive to the amount of computation in this loop, which depends upon the number of clusters, K. Thus, with $K = 10$, `pthread-vec` version is even slower than the `pthread-novec` version. With $K = 100$, `pthread-vec` gains 3.5x speedup compared to `pthread-novec`. However, with `SIMD-API`, the vectorization is applied on the outermost loop, which is the loop iterating over all input points. In addition, with data reorganization and effective management of branches, we can further improve the performance. Thus, `SIMD-API` gains 2.5 and 7.8 times speedups with $K = 10$ and $K = 100$. Another optimization applicable to Kmeans is the reordering of reduction. When K is smaller than the number of vector lanes, it is impossible to eliminate the write-conflicts, but this optimization is effective with a large K.

Now, considering NBC, large number of branches causes significant overhead on vectorization. So, the available production compiler failed to vectorize the kernel function in NBC, i.e, the difference between `pthread-novec` and `pthread-vec` is negligible. However, with the help of the mask operations introduced in our framework, `SIMD-API` still gains 1.5 and 1.6 times speedups on small and large datasets, respectively.

For stencil computations, one of the major problems for vectorization is unaligned memory accesses. In our framework, we overcome this limitation by reorganizing the datasets. However, the ICC compiler also has the capability to do the automatic vectorization. In Heat3D, we can see that `pthread-vec` achieves the best performance, which is very close to `SIMD-API` and `SIMD-Manual`. But for Sobel, compiler vectorization fails due to the extra inner loop that applies weights on the neighbors of the target node. Thus, the performance of `pthread-vec` is very similar to that of `pthread-novec`, whereas `SIMD-API` can still achieve more than 2x speedup, because vectorization is not limited to the inner loop.

For irregular reductions, the production compiler cannot vectorize a loop with indirection-based memory access at all. In our framework, we use data reordering together with a reduction in the use of gather and scatter operations to vectorize such kind of loops, which turns out to be effective when the datasets are large. We achieve 1.5 and 2.5 times speedup over the pthread versions for Euler and MD, respectively. For small datasets, the performance of the best `SIMD-API` version is comparable to the pthread versions. However, the best configuration with `SIMD-API` involves fewer threads (60 instead of 244). In other words, for the smaller datasets, enough parallelism is not available to exploit both MIMD and SIMD features. Comparing to the best `SIMD-manual` versions, `SIMD-API` incurs neglectable overheads.

5.3 Overall Scalability

In Figure 6, we compare the scalability of `pthread-novec`, `pthread-vec`, `SIMD-API`, and `SIMD-Manual` with an increasing number of threads. Execution with a single thread and no vectorization on Xeon Phi is used as the baseline, and thus, we are evaluating the combined benefits of shared memory parallelization (61 cores), hardware multi-threading (4 threads per core) and SIMD units. The performance scales well for all the versions. `SIMD-API` outperforms both `pthread-vec` and `pthread-novec` in most cases, consistent with what we reported earlier. `SIMD-API` achieves better relative performance when the number of threads is small. For instance, when the number of threads is one, `SIMD-API` is 20 times better than the `Pthreads-novec` version. With small datasets, as the number of threads increases, the vectorization advantage with `SIMD-API` becomes restricted due to limited amount of overall work. The overall speedups obtained range between 580 and 33, depending upon the application. Thus, we can see that our framework is effective in allowing users to exploit the Xeon Phi chip. As an aside, benefits of hardware multi-threading (more than 1 thread per core) seem limited, except for Kmeans (speedup from 2 threads per core, but slowdown from 4 threads per core) and the irregular applications (where latency is masked by hardware multi-threading). In the future, we will examine this issue further and develop a module for automatically choosing the number of threads for a given application.

5.4 Comparison with OpenMP

Our last set of experiments had two distinct goals. First, we wanted to examine how SIMD parallelization with our framework compares

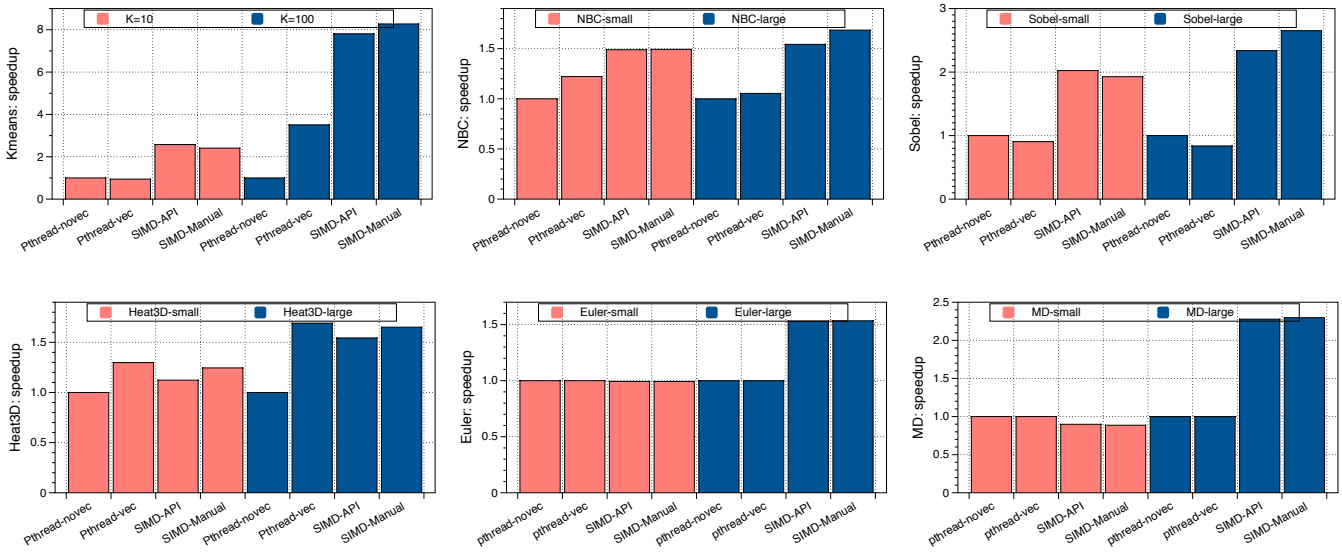

Figure 5: Speedup of Pthread without SIMD (`Pthread-novec`), Pthread with auto-SIMD (`Pthread-vec`), MIC SIMD with our framework (`SIMD-API`), and hand-written SIMD (`SIMD-manual`): Kmeans, NBC, Sobel, Heat3D, Euler, and MD with `small` and `large` datasets each

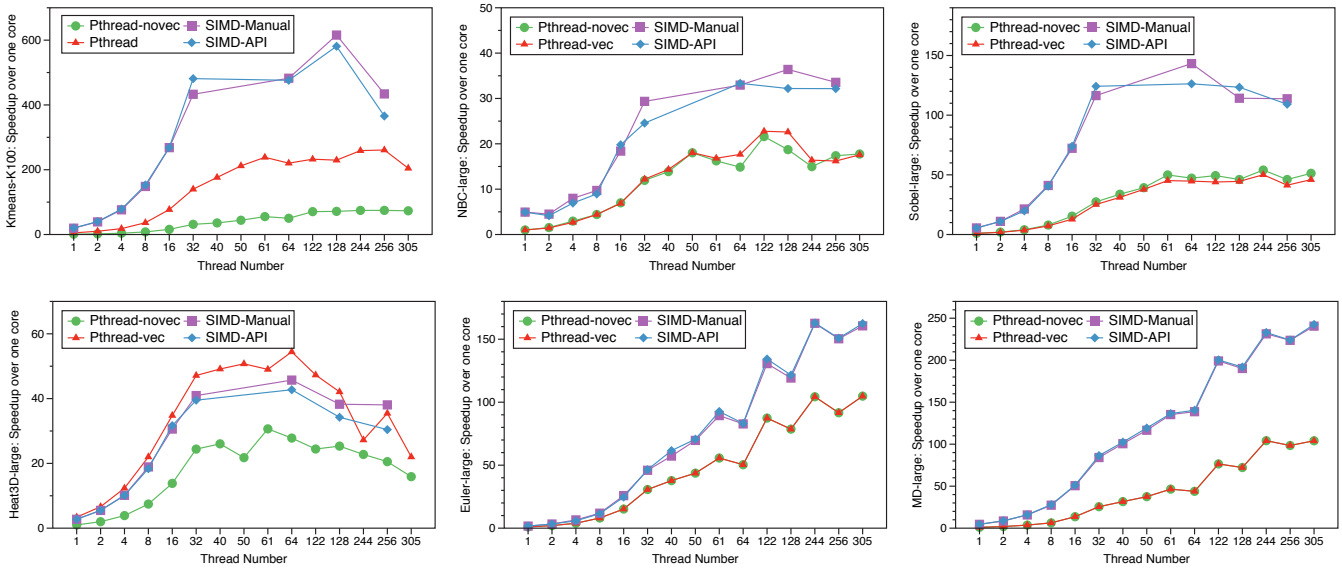

Figure 6: Scalability with Increasing Number of Threads: Pthread without vectorization (`Pthread-novec`), Pthread with auto-vectorization (`Pthread-vec`), SIMD with API (`SIMD-API`), and hand-written SIMD (`SIMD-manual`) with Kmeans, NBC, Sobel, Heat3D, Euler, and MD (large datasets) - Relative Speedups Over 1 Thread Execution on Xeon Phi with no Vectorization

against SIMD parallelization performed by the ICC compiler with OpenMP directives. Second, because both OpenMP and the MIMD API in our framework provide a high-level model for developing shared memory applications, we wanted to examine if our framework offers any performance advantages, possibly because it exploits the knowledge of the underlying communication patterns.

In Figure 7, we compared our MIMD parallel framework with and without SIMD parallelization to the OpenMP MIMD parallelization with and without the compiler vectorization. Comparing `MIMD+SIMD` to `OpenMP-vec`, more than 3 times speedup is achieved in Kmeans and NBC, due to the better SIMD parallelization and efficient MIMD parallelism. For Heat3D, OpenMP with compiler vectorization can provide good performance, but our parallel framework still outperforms the OpenMP version, due to the more efficient MIMD parallelism. Sobel, where SIMD parallelization is not achieved by the compiler, our framework gains significant speedups compared to the OpenMP version.

Now, focusing just on MIMD parallelization, our parallel framework still obtains better performance compared to OpenMP. The benefits of our framework are modest for Kmeans and stencil computations, but more significant for NBC and the two irregular applications. Overall, combining both MIMD and SIMD parallelization, our framework is better for all six applications, has relative speedup of 2.5 or better for five of the six applications, and for the two irregular reductions, it has an improvement by a factor of 4 and 7, respectively. As discussed throughout the paper, these advantages come from a number of factors, e.g., our framework can vectorize an irregular kernel with indirection-based memory accesses, while OpenMP compiler cannot, and pattern-aware MIMD partitioning and scheduling can avoid locking overheads.

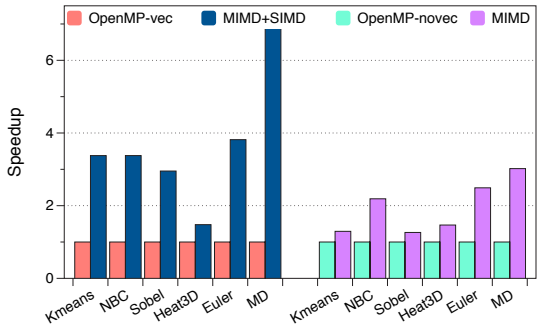

Figure 7: Benefits of MIMD+SIMD Execution in our Framework (Comparison with `OpenMP-vec` - left) and MIMD-only execution (Comparison with `OpenMP-novec` - right)

6. RELATED WORK

Intel SSE has been a part of the x86 since 1999, and there have been many efforts to automatically accelerate various applications using these instructions. For vectorizing stencils, memory alignment is a key problem, which was addressed by Eichenberger *et al.* [7] and Nuzman *et al.* [20] with data reorganization methods. More Recently, Henretty *et al.* [9] propose a system that involved improving data locality and utilizing short-vector SIMD optimizations, and Kong *et al.* [15] designed a Polyhedral compiler to perform loop transformation, optimization and vectorization for imperfectly nested loops.

Vectorizing irregular applications on SSE has also gained considerable interest in recent years. Kim and Han [14] propose a compiler method to generate efficient SIMD code for irregular kernels containing indirection based memory accesses. However, their work is on Cell SPU, with much shorter SIMD unit compared to the Xeon Phi and their method primarily focuses on intra-iteration vectorization. We focus on aggressive inter-iteration parallelism, consistent with presence of wide SIMD lanes. Tian *et al.* [26] provided an extension to the current directive vectorization methods to support function call. ISPC [22] provides a compiler based solution supporting function calls, SOA data structure, and control flow. The focus of our work is different, as we are providing a template based runtime solution for auto-vectorization. It utilizes the knowledge of patterns to automatically conduct data reorganization for different patterns. Moreover, it can also help resolving data dependencies in runtime, which is difficult to be handled for compiler solutions. Ren *et al.* [23] design a *virtual machine* together with domain-specific *bytecodes* method for pointer data traversals. There are also efforts on hand-optimizing irregular applications on SSE and other vector units [24, 13].

Some of the GPU compilation efforts have a similar favor, because SIMT is closely related to SIMD. This includes work on parallelizing stencil applications on GPUs [5, 18, 19, 10, 4]. For irregular applications on GPU, the coalesced memory access problem has also been addressed [28, 29]. However, because of the differences in the architectures (e.g. lack of atomic stores), our data reorganization methods are different. Overall, as compared to the existing work on SIMD compilation, the key distinctive aspects of our work are: 1) handling branches in a general way, 2) exploiting features in the IMCI instruction set, 3) using knowledge of communication patterns for runtime data reorganization, and 4) use of an overloaded function approach, which is unlike all previous efforts on SIMD parallelization, and can simplify the compiler code generation in the future.

There are also many efforts to parallelize various applications on Xeon Phi, which includes the work of Liu *et al.* [16] on Sparse Matrix-Vector Multiplication, Pennycook *et al.* [21] on parallelizing a Molecular Dynamic application, and Lu *et al.* [17] on optimizing the MapReduce framework. We have, to the best of our knowledge, offered the first general and end-to-end system for exploiting both MIMD and SIMD parallelism on the Xeon Phis.

7. CONCLUSIONS

This paper has presented and evaluated a framework for parallelization on the Xeon Phi coprocessors. Two distinct aspects of our work are 1) use of the knowledge of underlying patterns to perform job partitioning and scheduling in MIMD setting and data reorganization for SIMD parallelization, and 2) a very different approach for SIMD code execution, based on the implementation of overloaded functions, with runtime management of masks. Overall, we perform SIMD parallelization in presence of control flow, irregular accesses, and reductions, unlike previous work with SSE-like instruction sets. Moreover, our work can also be seen as providing a CUDA-like language (and its implementation) for using SSE-like instruction sets.

8. REFERENCES

[1] `http://www.top500.org/lists/2013/11/`.

[2] `http://dournac.org/info/parallel_heat3d`.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical Report EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[4] L. Chen, X. Huo, and G. Agrawal. Scheduling methods for accelerating applications on architectures with heterogeneous cores. In *HCW13*, 2013.

[5] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. A. Patterson, J. Shalf, and K. A. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC*, page 4. IEEE/ACM, 2008.

[6] J. Dokulil, E. Bajrovic, S. Benkner, S. Pllana, M. Sandrieser, and B. Bachmayer. Efficient hybrid execution of c++ applications using intel xeon phi coprocessor. *CoRR*, abs/1211.5530, 2012.

[7] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *PLDI*, pages 82–93. ACM, 2004.

[8] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *CC'11/ETAPS'11*, pages 225–245, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *ICS*, pages 13–24, 2013.

[10] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the international conference on Supercomputing*, pages 311–320. ACM, 2012.

[11] X. Huo, V. Ravi, W. Ma, and G. Agrawal. An execution strategy and optimized runtime support for parallelizing irregular reductions on modern gpus. In *Proceedings of the international conference on Supercomputing*, pages 2–11. ACM, 2011.

[12] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[13] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. Nguyen, T. Kaldewey, V. Lee, S. Brandt, and P. Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the International Conference on Management of Data*. ACM, 2010.

[14] S. Kim and H. Han. Efficient simd code generation for irregular kernels. *ACM SIGPLAN Notices*, 47(8):55–64, 2012.

[15] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When polyhedral transformations meet simd code generation. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 127–138. ACM, 2013.

[16] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th international ACM conference on supercomputing*, pages 273–282. ACM, 2013.

[17] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R. S. M. Goh, and R. Huynh. Optimizing the mapreduce framework on intel xeon phi coprocessor. *CoRR*, abs/1309.0215, 2013.

[18] J. Meng and K. Skadron. A performance study for iterative stencil loops on gpus with ghost zone optimizations. *International Journal of Parallel Programming*, 39(1):115–142, 2011.

[19] A. D. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *SC*, pages 1–13. IEEE, 2010.

[20] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In *PLDI*, pages 132–143. ACM, 2006.

[21] S. Pennycook, C. Hughes, M. Smelyanskiy, and S. Jarvis. Exploring simd for molecular dynamics, using intel xeon processors and intel xeon phi coprocessors. In *IPDPS*, 2013.

[22] M. Pharr and W. R. Mark. ispc: A spmd compiler for high-performance cpu programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13. IEEE, 2012.

[23] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte. Simd parallelization of applications that traverse irregular data structures. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10. IEEE, 2013.

[24] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. *PVLDB*, 4(11):795–806, 2011.

[25] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[26] X. Tian, H. Saito, M. Girkar, S. Preis, S. Kozhukhov, A. G. Cherkasov, C. Nelson, N. Panchenko, and R. Geva. Compiling c/c++ simd extensions for function and loop vectorizaion on multicore-simd processors. In *IPDPS Workshops*, pages 2349–2358. IEEE Computer Society, 2012.

[27] S. Williams, D. D. Kalamkar, A. Singh, A. M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker. Optimization of geometric multigrid for emerging multi- and manycore processors. SC '12, 2012.

[28] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. In *Proceedings of the SIGPLAN symposium on Principles and practice of parallel programming*, 2013.

[29] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *ASPLOS*, pages 369–380, 2011.