CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3621

SPECIAL ISSUE PAPER

Vectorizing unstructured mesh computations for many-core architectures

István Z. Reguly[1,2,*,†], Endre László[1,2], Gihan R. Mudalige[1] and Mike B. Giles[1]

[1]*Oxford e-Research Centre, University of Oxford, Oxford, UK*
[2]*Faculty of Information Technology and Bionics, Pázmány Péter Catholic University, Budapest, Hungary*

SUMMARY

Achieving optimal performance on the latest multi-core and many-core architectures increasingly depends on making efficient use of the hardware's vector units. This paper presents results on achieving high performance through vectorization on CPUs and the Xeon-Phi on a key class of irregular applications: unstructured mesh computations. Using single instruction multiple thread (SIMT) and single instruction multiple data (SIMD) programming models, we show how unstructured mesh computations map to OpenCL or vector intrinsics through the use of code generation techniques in the OP2 Domain Specific Library and explore how irregular memory accesses and race conditions can be organized on different hardware. We benchmark Intel Xeon CPUs and the Xeon-Phi, using a tsunami simulation and a representative CFD benchmark. Results are compared with previous work on CPUs and NVIDIA GPUs to provide a comparison of achievable performance on current many-core systems. We show that auto-vectorization and the OpenCL SIMT model do not map efficiently to CPU vector units because of vectorization issues and threading overheads. In contrast, using SIMD vector intrinsics imposes some restrictions and requires more involved programming techniques but results in efficient code and near-optimal performance, two times faster than non-vectorized code. We observe that the Xeon-Phi does not provide good performance for these applications but is still comparable with a pair of mid-range Xeon chips. Copyright © 2015 John Wiley & Sons, Ltd.

Received 15 April 2014; Revised 20 November 2014; Accepted 17 July 2015

KEY WORDS: vectorization; Xeon Phi; AVX; CUDA; unstructured grid; programming abstraction

1. INTRODUCTION

The rapidly changing hardware architectures of microprocessor systems continue to produce highly parallel systems in the form of multi-core, many-core, and heterogeneous systems. Today, most computer system has a complex configuration with many levels of parallelism. At the low end, four processor cores on a single piece of silicon is commonplace, while many emerging architectures such as many-core accelerators/co-processors have hundreds or thousands of cores. At the high-end, systems such as large parallel high performance computing clusters are developed with combinations of these processors and co-processors [1], with heterogeneous configurations propelling them towards capabilities to scale up to a billion threads. These systems have become significantly more difficult to program for performance than systems with traditional single-threaded microprocessors [2].

Parallel programming models, such as distributed memory message passing and shared memory multi-threading, have been the dominant models for programming traditional CPU-based systems. These programming models were largely based on utilizing task level parallelism either through multi-threading (OpenMP, POSIX threads) or message passing (MPI). But the emergence of accel-

*Correspondence to: István Z. Reguly, Oxford e-Research Centre, 7 Keble Road, Oxford, OX1 3QG, UK.
†E-mail: istvan.reguly@oerc.ox.ac.uk

erators such as GPUs (e.g., NVIDIA GPUs [3]), the Intel Xeon Phi [4], and similar co-processors (DSPs [5] and FPGAs [6]) have complicated the way in which parallel programs are written to utilize these systems. Even traditional x86 based processor cores now have increasingly larger vector units (e.g., Advanced Vector Extensions (AVX)) and require special programming techniques in order to acquire full utilization.

Many-core co-processors such as GPUs have brought with them new extensions (e.g., CUDA and OpenCL) to general-purpose programming languages such as C/C++/Fortran. These follow a single instruction multiple thread (SIMT) parallel programming model. NVIDIA's CUDA has gained widespread popularity and has developed a large software ecosystem. However, it is restricted to NVIDIA GPUs. OpenCL, based on a very similar SIMT model, supports a wider range of hardware but has not developed an ecosystem comparable with that of CUDA – in part because it struggles with the lack of proactive support from some hardware vendors. At the same time, larger vector units are being designed into CPUs, and at the lowest level, they require a very explicit single instruction multiple data (SIMD) programming model. In an SIMD programming model, operations are carried out on packed vectors(128–512 bits) of values as opposed to scalar values, and it is more restrictive than the SIMT model. For example, in the case of data-driven random memory access patterns that arise in key classes of applications, computations require explicit handling of aligned, unaligned, scatter, and gather type accesses in the machine code. In contrast, in the SIMT model, this is carried out implicitly by the hardware.

In previous generations of CPUs, vectorization of computations was of lesser importance, because of the shorter vector units; however, the 256-bit and 512-bit long vectors have become key features in the latest architectures; their utilization is increasingly necessary to achieve high performance. This has drawn the auto-vectorization capabilities of modern compilers into focus; such compilers at best could only vectorize a few classes of applications with regular memory access and computation patterns, such as structured grids or multimedia.

The programming techniques that make it possible to utilize vectorization are thus becoming more and more important. Gaining good performance from vector units is not only important for CPUs but also for emerging co-processors such as the Intel Xeon Phi. While the Xeon Phi is designed as an accelerator with processor cores based on a simpler x86 architecture, it has the largest vector lengths currently found on any processor core. However, it requires very explicit programming techniques specific to the hardware to gain maximum performance.

It has been shown that very low level assembly implementations of algorithms can deliver performance on the Xeon Phi, usually as part of software packages such as MKL [7] or Linpack [8]. Several studies have been published, which discuss porting existing applications to the Phi by relying on higher-level language features, usually compiler auto-vectorization, and have shown success in the fields of structured grid computations [9, 10], molecular dynamics [11, 12], and finance [13]. Most of the computations in these applications were engineered to lend themselves easily to auto-vectorization because of the structure of the underlying problems. Even then, the use of low level vector programming was still required in many cases. Irregular computations have been notoriously hard to vectorize, because of dependencies driven by data [14].

The focus of this paper is to present research into gaining higher performance through vectorization on CPUs and the Xeon Phi. Unlike previous work, we target the application domain of unstructured mesh-based applications, a key class of applications that have very irregular access patterns. Our aim is to present the challenges in achieving good vectorizing code for these applications, which goes beyond the simple structured and regular applications that are well documented in many-core computing literature. We make use of the OP2 [15] domain-specific abstraction framework to develop specific vectorizing implementations. OP2 is an 'active' library framework for the solution of unstructured mesh applications. The active library approach uses program transformation tools so that a single application code written using the OP2 API is transformed into the appropriate form that can be linked against a given parallel implementation enabling execution on different back-end hardware platforms. We exploit the multi-layered development strategy of OP2 to work and optimize at the required level of parallelization, namely, targeting SIMD vectorization on CPUs and Intel's Xeon Phi, for unstructured mesh applications. These implementations are then benchmarked and compared with the performance of other (non-vectorized) parallelizations previously developed with OP2 for the same application. We show that it is difficult to achieve efficient vectorization on both CPUs and the Xeon Phi; acquiring high performance often involves coding in very low-level intrinsics, but a high-level code generation framework such as OP2 can hide many of the platform-specific issues from the application developer. More specifically, we make the following contributions:

(1) We explore the SIMT and SIMD execution and parallel programming models and investigate how popular programming language extensions (CUDA and OpenCL) and techniques (e.g., vector intrinsics) available for programming modern CPUs and accelerators/co-processors map to these execution models. Furthermore, we introduce execution schemes that permit compiler auto-vectorization of OpenMP code.
(2) Two CFD applications developed with OP2, the Airfoil [16] benchmark and the previously unpublished tsunami simulation code Volna [17], are used to illustrate the contrasting strategies required to achieve good vectorization. As a result of this work, we develop for Airfoil and Volna (1) a vector intrinsics-based version and (2) an OpenCL-based version, both of which can be executed on Intel CPUs and the Xeon Phi co-processor.
(3) The resulting vectorized codes are benchmarked on a number of CPU and many-core processor systems including two CPUs based on the latest generation of Intel's Sandy Bridge and Ivy Bridge micro-architecture, and an Intel Xeon Phi, where they are also compared with auto-vectorized code. These results are compared against Airfoil's performance on NVIDIA GPUs (K40) developed in the previous work [18] to provide a comparison of what could be achieved with current many-core systems. We use highly optimized code generated through OP2 for all back-ends, using the same application code, allowing for a direct performance comparison.
(4) Finally, we present a performance analysis study of what has been achieved by our vectorization efforts, exposing the underlying reasons for the performance of various implementations on different hardware and performance bottlenecks caused by computational, control, or bandwidth limitations.

The rest of the paper is organized as follows: Section 2 introduces the parallel programming models used, from the perspective of unstructured grids; Section 3 discusses the OP2 abstraction for unstructured grids and parallelization approaches used to guarantee correct execution. Section 4 discusses the approaches and code transformations used to acquire vectorization, and Section 5 discusses the existing back-ends, which will serve as a baseline for performance analysis in the rest of the paper. Section 6 presents our experimental setup, the hardware used, and the problem being solved through OP2 and analyses performance on the test hardware. Finally, Section 7 gives an overview of the results presented in the paper.

2. PARALLEL EXECUTION AND PROGRAMMING MODELS

To make efficient use of today's heterogeneous architectures, a number of parallelization strategies have to be employed, often combined with each other to enable execution in different systems utilizing multiple levels of parallelism. Usually at the topmost level, we see the distributed memory parallelization model, which involves distributing the problem across a number of processes, with explicit message passing between processes for coordinating the solution to the problem. The Message Passing Interface (MPI) has become the default standard in implementing this model. While it is possible to use just distributed memory parallelization, by assigning one process to each CPU core in a multi-core processor, it is often difficult to maintain such a coarse level of parallelism because of the increasing number of cores on a single compute node. As the cores on a single compute node share other resources such as NIC, memory, and caches with a distributed memory model, the contention for these shared resources results in increasing communication and memory storage/access bottlenecks [19].

Therefore, underneath a distributed memory parallelization, a shared memory parallelization approach is often used to reduce the burden of explicit communications between cores. Threads are used as the execution unit at this level, where multiple threads share a block of memory, as opposed to MPI processes maintaining their own private memory block. This model, called simultaneous multi-threading, executes a collection of processing streams operating largely independently, each with its own execution path, working on different chunks of shared data. Currently popular simultaneous multi-threading implementations include OpenMP [20] and P-threads.

Recently, many-core processors and co-processors gained widespread popularity. These consist of a large number of low power, low frequency compute cores and rely on high throughput to achieve performance by allowing to execute a massive number of threads in parallel. This is in contrast to speeding up execution of a few threads on traditional CPUs. For many-core processors and co-processors, a popular programming model is the SIMT model, where a number of lightweight threads execute the same instructions at the same time. From a programming perspective, one implements code for a single thread, where data or control flow usually depends on a thread index, and then this code is executed by different threads concurrently. While it is possible for different threads to have divergent flow paths, in these cases, the execution is serialized because there is only a single instruction being executed at the same time; threads not active in the currently executed branch are deactivated by the hardware. Threads may also access different memory addresses when executing the same instruction; the hardware is responsible for collecting data and feeding it to the threads. The relative position of these memory addresses between adjacent threads has an important impact on performance: addresses packed next to each other may be accessed with a single memory transaction, while gather-type and scatter-type accesses may require multiple transactions. This combined with the comparatively small amount of cache per thread often has a non-trivial effect on performance that is difficult to predict. CUDA and OpenCL are based on the SIMT model, and the latter maps to both CPU vector units and GPUs.

Finally, the SIMD execution and programming model is used by the vector units on Intel CPUs and the Xeon Phi. While some programming models (such as OpenCL) do support an SIMT programming model and compilation for these architectures, the hardware and therefore the generated assembly code has to use SIMD. The most reliable way of obtaining vectorization is by the explicit use of vector registers and instructions; we investigate the AVX on Sandy Bridge processors and the Initial Many Core Instructions (IMCI) specific to the Xeon Phi platform. Most of these instructions have a one-to-one correspondence with assembly instructions but are more readable and do not require explicit management of registers. Vector instructions operate on vector registers that are 256 bits (AVX) or 512 bits (IMCI) long; they can contain 8 or 16 integers/floats or 4 or 8 doubles. There is also support for masked instructions, where specific vector lanes can be excluded from an instruction, thereby facilitating branching. This execution model implies that data have to be explicitly packed into vector registers that can then be passed as arguments to an operation. Explicit management of data movement requires differentiation between loading a contiguous chunk of data from memory that is (1) aligned, (2) not aligned to the vector length, or (3) that has to be collected from different addresses and packed; the same patterns apply for store operations.

Applications for scientific computations often combine these parallel programming models in order to ensure high performance on modern hardware; however, from an abstract parallelization perspective, there are huge differences between different types of computations; take for example a structured grid computation as illustrated in Figure 1(a) that applies a finite difference stencil – parallelism here is almost trivial because different iterations of the loop are completely independent of each other; it is only in a distributed setting that attention has to be paid to exchanging 'halo'

```
int number_of_points=...;
double *input=...; //size number_of_points
double *result=...; //size number_of_points
for ( int n=1; n<number_of_points-1; n++ ){
  result[n] = -2.0*input[n] + input[n-1]
                            + input[n+1];
}
```

(a) Finite difference stencil on structured grids

```
int number_of_nodes=...;
int number_of_edges=...;
int edges2nodes=...; //size 2*number_of_edges
double *edge_weights=...; //size number_of_edges
double *result=...; //size number_of_nodes
double *multiplicand=...; //size number_of_nodes
for ( int n=0; n<number_of_edges; n++ ){
  result[edges2nodes[2*n]] += edge_weights[n] *
         multiplicand[edges2nodes[2*n+1]];
}
```

(b) Sparse matrix-vector product on unstructured grids

Figure 1. Typical operations over structured and unstructured grids. (a) Finite difference stencil on structured grids; (b) sparse matrix–vector product on unstructured grids.

information. Most compilers are capable of automatically vectorizing such computations, and if the alignment of the base pointers is known, even aligned memory accesses can be inserted at compile time. On the other hand, in most unstructured grid computations, data are accessed indirectly. Figure 1(b) shows an example of a sparse matrix–vector product; clearly, different iterations of the loop may not be independent of each other, which may present data races when trying to parallelize this operation. Hence, the compiler cannot automatically parallelize this computation. It is also much more difficult to satisfy data dependencies in a distributed memory environment, because, again, indirections depend on data. In such an unstructured setting, it is necessary to carry out further pre-processing, based on the indirections, to guarantee execution free of data races, and in the case of SIMD vectorization, it is necessary to gather and scatter data to or from vector registers with compile-time unknown memory locations.

While many structured grid computations are amenable to pragma-driven compiler auto-parallelization, most unstructured grid computations require a lot of additional effort to parallelize. Developing and maintaining large-scale codes with multiple different combinations of parallel programming approaches in order to support today's diverse hardware landscape is clearly not feasible, doing so would reduce the productivity of application developers and would also require intimate knowledge of different hardware. Our approach to tackling this issue is to use a domain-specific, high-level abstraction, based on which we can automatically map to different parallel programming models and hardware, managing data dependencies and race conditions.

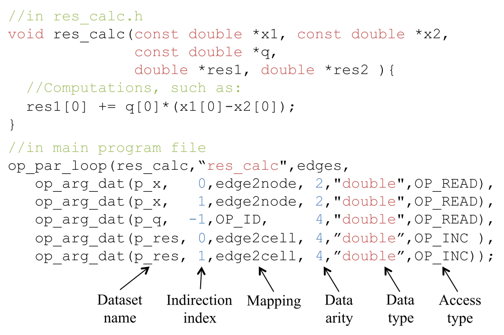
3. THE OP2 LIBRARY FOR UNSTRUCTURED GRIDS

Recently, domain specific languages (DSLs) and similar high-level abstraction frameworks have been utilized to reduce the complexity of programming parallel applications. With DSLs, the application programmer describes the problem at a higher level and leaves the details of the implementation to the library developer. Given the right abstraction, this enables high productivity, easy maintenance, and code longevity for the domain scientist, permitting them to focus on the problem at hand. At the same time, it enables the library developers to apply radical and platform-specific optimizations that help deliver near-optimal performance.

OP2 is such a high-level framework for the solution of unstructured mesh applications [15]. Its abstraction involves breaking up the problem into four distinct parts: (1) sets, such as vertices or edges, (2) data on sets, such as coordinates and flow variables, (3) connectivity between sets, and (4) operations over sets. These form the OP2 API that can be used to fully and abstractly define any unstructured mesh. Unstructured grid algorithms tend to iterate over different sets, accessing and modifying data indirectly on other sets via mappings; for example, flux computations often loop over edges accessing data on edges and neighboring cells, updating flow variables indirectly on these cells. In a parallel setting, this leads to data races, the efficient handling of which is paramount for high performance.

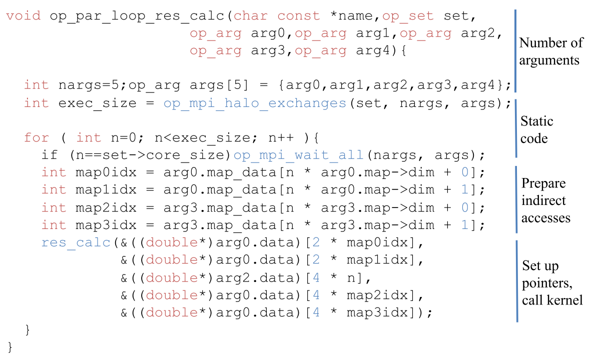
The OP2 abstraction is designed to implicitly describe parallelism; the basic assumption is that the order in which elements are processed does not affect the final result to within the limits of finite precision floating point arithmetic. This allows for parallelization of the execution over different set elements; however, potential data races have to be recognized and dealt with. Therefore, the API is designed to explicitly express access types and patterns, based on the following building blocks:

(1) `op_set`: basic components of the mesh, such as edges and cells.
(2) `op_dat`: data on each element of a set, with a given arity (number of components).
(3) `op_map`: connectivity from one set to another, with a given arity, for example, each edge connects to vertices.
(4) `op_par_loop`: a parallel loop over a given set, executing an elementary kernel function on each element of the set passing in pointers to data based on arguments described as follows:
`op_arg_dat(op_dat,idx,op_map,dim,"typ",access)`,
where a given dataset with `dim` arity and `type` datatype is to be accessed through a specific index in a mapping to another set (or no mapping if it is defined on the same dataset), describing the type of access, which can be either read, write, increment, or read–write.



```
//in res_calc.h
void res_calc(const double *x1, const double *x2,
              const double *q,
              double *res1, double *res2 ){
  //Computations, such as:
  res1[0] += q[0]*(x1[0]-x2[0]);
}
//in main program file
op_par_loop(res_calc,“res_calc",edges,
   op_arg_dat(p_x,   0,edge2node, 2,"double",OP_READ),
   op_arg_dat(p_x,   1,edge2node, 2,"double",OP_READ),
   op_arg_dat(p_q,  -1,OP_ID,     4,"double",OP_READ),
   op_arg_dat(p_res, 0,edge2cell, 4,”double”,OP_INC ),
   op_arg_dat(p_res, 1,edge2cell, 4,”double”,OP_INC));
```

(a) Specification of a parallel loop



```
void op_par_loop_res_calc(char const *name,op_set set,
                          op_arg arg0,op_arg arg1,op_arg arg2,
                          op_arg arg3,op_arg arg4){

  int nargs=5;op_arg args[5] = {arg0,arg1,arg2,arg3,arg4};
  int exec_size = op_mpi_halo_exchanges(set, nargs, args);

  for ( int n=0; n<exec_size; n++ ){
    if (n==set->core_size)op_mpi_wait_all(nargs, args);
    int map0idx = arg0.map_data[n * arg0.map->dim + 0];
    int map1idx = arg0.map_data[n * arg0.map->dim + 1];
    int map2idx = arg3.map_data[n * arg3.map->dim + 0];
    int map3idx = arg3.map_data[n * arg3.map->dim + 1];
    res_calc(&((double*)arg0.data)[2 * map0idx],
             &((double*)arg0.data)[2 * map1idx],
             &((double*)arg2.data)[4 * n],
             &((double*)arg0.data)[4 * map2idx],
             &((double*)arg0.data)[4 * map3idx]);
  }
}
```

(b) Code generated for MPI execution

Figure 2. The OP2 API and a simple example of code generation. (a) Specification of a parallel loop; (b) code generated for MPI execution.

Figure 2(a) gives an example of how a parallel loop is described; iterating over edges, reading data on vertices and edges, and incrementing data on cells, and Figure 2(b) gives a simplified example of code generated to execute this computation using the pure MPI back-end – note how information provided in the API is sufficient to generate this code.

This API allows OP2 to use a combination of code generation and run-time execution planning in order to support a wide range of contrasting hardware platforms, using a number of parallel execution models. At the highest level, we use distributed memory parallelization through message passing (MPI), where the mesh is split up into partitions using standard partitioners such as PT-Scotch [21], and an owner-compute approach is used to be in combination with halo exchanges to ensure correct execution. There are no potential race conditions, but redundant execution of certain set elements by different processes may be necessary. On multi-core CPUs, we use OpenMP threads and, through a pre-processing step, split up the computational domain into mini-partitions, or blocks, which are colored based on potential data races [22]. Blocks of the same color can then be executed by different threads concurrently without any need for synchronization. The same technique is used to assign work to CUDA thread blocks or OpenCL work groups.

It is evident that by working within such a multi-layered abstractions framework there is opportunity to select and optimize the required level of parallelization for the types of unstructured mesh applications solved with OP2. The focus of the paper is therefore to present the previously unexplored SIMD parallelization level for CPUs and the Intel Xeon Phi. We will demonstrate that our work in obtaining good vectorizing code through OP2 is applicable to any application that falls within the domain of OP2.

4. VECTORIZATION

Most unstructured mesh computations are not immediately suitable for parallelization because of data-driven races; therefore, we carry out further pre-processing, and we alter execution patterns to guarantee that no race conditions exist. In order to work towards achieving vectorization in the context of OP2, we draw upon our previous experience by using GPUs [15]. We have used a two-level coloring approach to avoid race conditions when indirectly writing data; first, the problem is broken up into blocks of elements, and these blocks are colored based on potential races, and second, elements within each block are colored based on potential races.

When using an SIMT programming model, we assign adjacent set elements to adjacent threads, gather operations and potential branching is automatically handled by the programming model and the hardware, and in case of indirect writes to data, we serialize access based on the second level of coloring and block-level synchronization constructs. However, in addition to supporting OpenCL, we also wish to evaluate vector intrinsics, which are based on the SIMD programming model, as well as the compiler's abilities to automatically vectorize loops where different iterations are independent of each other. Therefore, we have introduced two additional coloring and execution schemes, which assign independent set elements to different vector lanes. With the ‘full permute’ approach, we have a single level of coloring, and set elements are executed by color in parallel, which avoids all race conditions – however, it also inhibits any data reuse between elements of the same color. The ‘block permute’ approach tries to combine the benefits of not having to serialize because of race conditions with improved temporal locality; in the two-level coloring approach, we compute a permutation map for the execution of elements within each block by color – as long as blocks are small enough so that their data are contained in cache, this permits data reuse – however, we are no longer assigning adjacent set elements to adjacent vector lanes; therefore, data that were formerly accessed directly now has to be gathered. Our goal in introducing these coloring schemes is to enable the race-free parallelization of an innermost loop over a number of independent elements; on one hand, this will reduce the branching necessary for the safe updating of shared variables, and on the other, it will enable compiler auto-vectorization. The performance impact of these approaches is only evaluated on hardware that support gather and scatter instructions – the K40 and the Xeon Phi – in Section 6.5.

Enabling compiler auto-vectorization is simply an issue of inserting the appropriate compiler pragmas, but utilizing an SIMT or an SIMD programming model requires more elaborate code generation and optimization techniques; therefore, we now discuss how OP2 can utilize a combination of back-end logic, C++ classes, and code generation to support vectorization for general unstructured mesh computations through OpenCL and vector intrinsics.

4.1. OpenCL

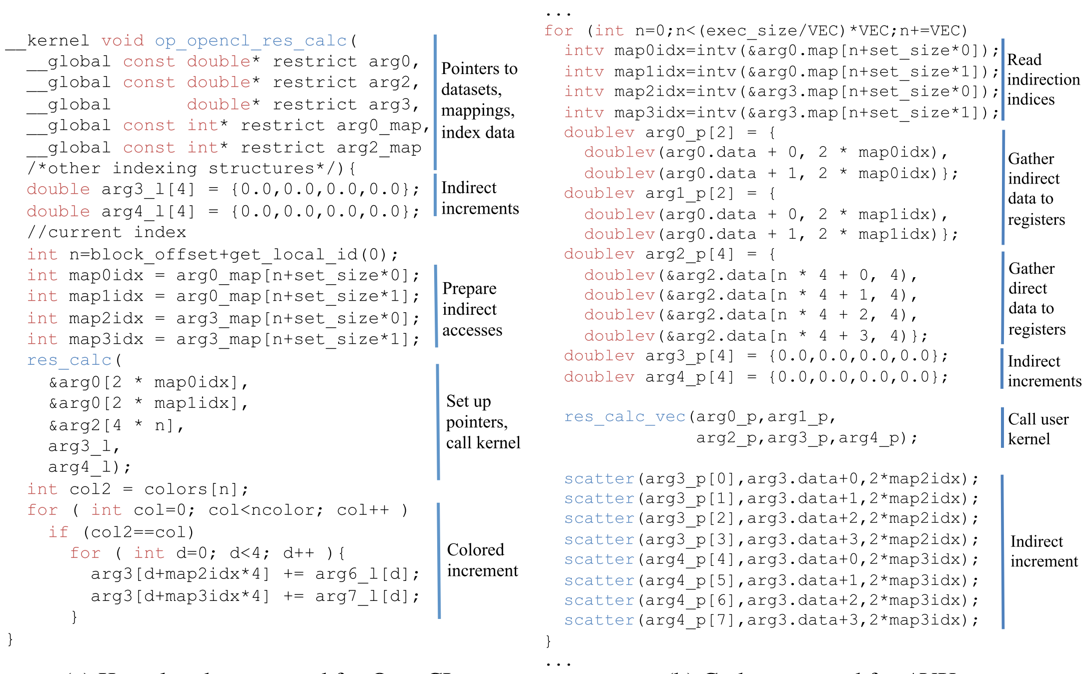
OpenCL (Open Computing Language) is an open standard language based on the SIMT abstraction, targeting a wide variety of heterogeneous computing platforms, including CPUs, GPUs, DSPs, and FPGAs. Because OpenCL is a standard, the implementation of the driver and the run-time environment rely on the support of the hardware manufacturers. Following the SIMT abstraction, the workload is broken up into a number of work groups, each consisting of a number of work items. The exact mapping of these is influenced by the parallel capabilities of the device and the possible optimizations that can be carried out by the compiler or synthesizer. As with most portable programming abstractions, performance portability is an issue; therefore, we first have to understand how it maps to the target hardware.

The OpenCL abstraction fully describes any concurrency issues that may arise; therefore, vectorization of work items would always be valid. However, similar to the case of compiler auto-vectorization of conventional C/C++ code, this can be prevented by a number of factors: control flow branching, irregular memory access patterns, and so on. Because Intel’s IMCI instruction set is more extensive than AVX, it gives more flexibility for the compiler to produce vector code.

Task parallelism at the level of the work groups in Intel OpenCL is implemented using Intel’s TBB (Thread Building Blocks) library [23]. Although TBB offers an efficient solution for multi-threading, the overall overhead of scheduling work groups is larger in OpenCL than that of static OpenMP parallel loops; however, this keeps improving. Because CPUs – as opposed to GPUs – do not have dedicated local and private memory units, using these memory spaces introduces unnecessary data movement and may lead to serious performance degradation. Thus, in the OpenCL-based CPU implementation, OP2 does not use local memory. OpenCL distinguishes between global, local, and private address spaces.

A key observation when optimizing code for the CPU is that work groups are executed sequentially. One can be certain that there are no data-races between work items in a work group if the compiled code is scalar. Even if the code is implicitly vectorized, the bundled work items execute in lock-step; synchronization between them is implicit, and these bundles are still executed sequentially. Recognizing this, it is possible to remove work group-level synchronizations, which would otherwise be very expensive. This of course is a platform-specific optimization that violates the OpenCL programming model, but it is necessary to obtain good performance. It is possible to remove barriers from the following operations:

- Sharing information specific to a block: Because every work group processes one block (mini partition) of the whole domain, some information about the block data structure can be shared



```
__kernel void op_opencl_res_calc(
  __global const double* restrict arg0,
  __global const double* restrict arg2,
  __global       double* restrict arg3,
  __global const int* restrict arg0_map,
  __global const int* restrict arg2_map
  /*other indexing structures*/){
  double arg3_l[4] = {0.0,0.0,0.0,0.0};
  double arg4_l[4] = {0.0,0.0,0.0,0.0};
  //current index
  int n=block_offset+get_local_id(0);
  int map0idx = arg0_map[n+set_size*0];
  int map1idx = arg0_map[n+set_size*1];
  int map2idx = arg3_map[n+set_size*0];
  int map3idx = arg3_map[n+set_size*1];
  res_calc(
    &arg0[2 * map0idx],
    &arg0[2 * map1idx],
    &arg2[4 * n],
    arg3_l,
    arg4_l);
  int col2 = colors[n];
  for ( int col=0; col<ncolor; col++ )
    if (col2==col)
      for ( int d=0; d<4; d++ ){
        arg3[d+map2idx*4] += arg6_l[d];
        arg3[d+map3idx*4] += arg7_l[d];
      }
}
```

Pointers to datasets, mappings, index data
Indirect increments
Prepare indirect accesses
Set up pointers, call kernel
Colored increment

```
...
for (int n=0;n<(exec_size/VEC)*VEC;n+=VEC)
  intv map0idx=intv(&arg0.map[n+set_size*0]);
  intv map1idx=intv(&arg0.map[n+set_size*1]);
  intv map2idx=intv(&arg3.map[n+set_size*0]);
  intv map3idx=intv(&arg3.map[n+set_size*1]);
  doublev arg0_p[2] = {
    doublev(arg0.data + 0, 2 * map0idx),
    doublev(arg0.data + 1, 2 * map0idx)};
  doublev arg1_p[2] = {
    doublev(arg0.data + 0, 2 * map1idx),
    doublev(arg0.data + 1, 2 * map1idx)};
  doublev arg2_p[4] = {
    doublev(&arg2.data[n * 4 + 0, 4),
    doublev(&arg2.data[n * 4 + 1, 4),
    doublev(&arg2.data[n * 4 + 2, 4),
    doublev(&arg2.data[n * 4 + 3, 4)};
  doublev arg3_p[4] = {0.0,0.0,0.0,0.0};
  doublev arg4_p[4] = {0.0,0.0,0.0,0.0};

  res_calc_vec(arg0_p,arg1_p,
               arg2_p,arg3_p,arg4_p);

  scatter(arg3_p[0],arg3.data+0,2*map2idx);
  scatter(arg3_p[1],arg3.data+1,2*map2idx);
  scatter(arg3_p[2],arg3.data+2,2*map2idx);
  scatter(arg3_p[3],arg3.data+3,2*map2idx);
  scatter(arg4_p[4],arg3.data+0,2*map3idx);
  scatter(arg4_p[5],arg3.data+1,2*map3idx);
  scatter(arg4_p[6],arg3.data+2,2*map3idx);
  scatter(arg4_p[7],arg3.data+3,2*map3idx);
}
...
```

(a) Kernel code generated for OpenCL (b) Code generated for AVX

Figure 3. Simplified examples of code generated for vectorized back-ends. (a) Kernel code generated for OpenCL; (b) code generated for AVX.

amongst the work items of the work group. Either these data are read by one work item and shared with other work items through a local variable or every work item reads the same data and then no local memory is necessary. The barrier is not necessary in the first case because work item 0 reads the data and stores it in local memory, and it will be available to every other work item, because work item 0 is executed before any other work item in the group, because of the sequential execution model.

- Reductions: Doing a reduction is straightforward because of the sequential execution even in the case of vectorized kernels; first, the reduction is carried out on vectors, and at the end, values of the accumulator vector are added up.
- Indirect increments (or indirect scatter): Use of barriers is not needed here either, because (1) the work group is executed sequentially and (2) even if the kernel is vectorized, a sequential colored increment is used to handle WAR (Write After Read) conflicts.

Figure 3(a) shows a simplified example of the code generated for the loop in Figure 2(a), the 'host-side' setup of parameters and the kernel launch are omitted for brevity. This code is very similar to the CUDA code generated for GPU execution, except for CPU-specific optimizations as discussed earlier.

4.2. *Vector intrinsics*

Our experiments have shown that the primary obstacle to acquiring auto-vectorization and good performance is the scatter–gather nature of memory accesses; for vectorized execution, the values of adjacent set elements have to be packed next to each other. By using vector intrinsics however, it is possible to explicitly vectorize execution; in this case, data have to be explicitly gathered into vector registers, the user kernel has to be modified to use intrinsic operations, and the results have to be explicitly scattered from the vector registers. While operators are not defined for these vector types, the Intel Compiler includes a header file (`dvec.h` or `micvec.h` [24]) that defines container classes for these types, overloading most operators. By extending these classes, it is possible to

```
#ifdef AVX
#define VEC 4
#typedef doublev F64vec4
#typedef intv I32vec4
...
  //A constructor of the F64vec4 class
  F64vec4(const double *p, I32vec4 &idx) {
    vec[0]=p[idx[0]]; vec[1]=p[idx[1]];
    vec[2]=p[idx[2]]; vec[3]=p[idx[3]];}
  F64vec4 operator *(F64vec4 &a, F64vec4 &b) {
    return _mm256_mul_pd(a,b);}
...
#endif
```

(a) 256 bit AVX

```
#ifdef IMCI
#define VEC 8
#typedef doublev F64vec8
#typedef intv I32vec8
...
  //A constructor of the F64vec8 class
  F64vec8(const double *p, I32vec8 &idx) {
    vec = _mm512_i32logather_pd(idx,p,8);
  }
  F64vec8 operator *(F64vec8 &a, F64vec8 &b) {
    return _mm512_mul_pd(a,b);}
...
#endif
```

(b) 512 bit IMCI

Figure 4. Wrapper classes for vector types. (a) 256 bit AVX; (b) 512 bit IMCI.

maintain the original simple arithmetic expressions in the user kernels, but instead of scalars, they will now operate on vectors.

With pre-processor macros, it is also possible to have a single source code generated by OP2 for different vector lengths and to select one at compile-time; Figure 4 shows an example of this; the possible targets are AVX (on the CPU) and IMCI on the Xeon Phi. An important obstacle is the lack of support for branching in kernels; without a compiler technology, we are not able to support conditionals through operator overloading. Therefore, the user of OP2 has to alter conditional code to use $select()$ instructions instead, which can be supported through function overloading. While this was easy to apply to the Airfoil benchmark, this is by no means a generic and flexible solution.

A range of load and store instructions are implemented that support aligned addresses, strided gather/scatter, or mapping-based gather/scatter operations, which enables us to utilize different instructions on different hardware; for example, the IMCI has a gather intrinsic. In the case of indirect incrementing of data, it is necessary to avoid data races; depending on the coloring approach, discussed in Section 4, we can either use simple scatter operations with the full/block permute approaches, or with the original two-level coloring approach, we would need masked scatter operations – these proved to be slower than just sequentially scattering data out of the vector register; therefore, we do not use them.

Code resulting from the use of vector containers is rather verbose and results in a much higher number of source lines because of explicit packing and unpacking of data, as illustrated in Figure 3(b). Furthermore, the iteration range for any given thread where vectorization can take place must be divisible by the vector length, in order to support aligned load instructions and fully packed vectors, which is not always the case (especially in an MPI+OpenMP setting), therefore there are actually three loops generated; a scalar pre-sweep to get directly accessed data aligned to the vector length, the main vectorized loop, and a scalar post-sweep to compute set elements left over.

5. BASELINE CPU AND GPU IMPLEMENTATIONS

Results presented in previous papers [15, 25] report OP2's pure MPI and hybrid MPI+OpenMP performance on clusters of CPUs as well as MPI+CUDA performance results running on Fermi-generation NVIDIA GPUs. We carried out an in-depth investigation of our MPI and OpenMP back-ends that revealed a few bottlenecks. Pure MPI execution was facilitated by a generic `op_par_loop` function that had to carry out a number of logical operations and called the user kernel via a function pointer. This prevented a number of compiler optimizations; to enable these, we implemented a code generator that produces a pure MPI stub file, which contains more specialized code, eliminating conditionals, substituting literal constants where possible, and with an explicit call to the user kernel – a simplified example of this was shown in Figure 2(b). The generated code for OpenMP execution was similarly optimized and simplified.

To evaluate the compiler's auto-vectorization capabilities on these implementations, we have inserted `#pragma ivdep` statements in front of innermost `for` loops when one of the newly introduced coloring approaches was being used. However, in our experience on the CPU, this rarely led to auto-vectorization of the loops in question, the single exception being the `adt_calc` kernel in Airfoil for the OpenMP implementation.

Because of significant changes in GPU architecture from the Fermi to the Kepler generation, we have also revised our optimization techniques; shared memory staging of indirectly accessed data we used previously does not prove to be efficient on the new hardware, so a new code generator is introduced for Kepler hardware that employs a different set of optimizations. Pointers to read-only data are decorated with `const * __restrict` to make use of caching loads. Furthermore, data on the GPU are transposed to an SoA layout, and directly accessed from global memory, not from shared memory. These changes are facilitated by the code generator, no changes have to be made to the 'user code'.

6. PERFORMANCE ANALYSIS

6.1. *Experimental setup*

It is our goal to benchmark the performance characteristics of different parallel programming approaches on a range of multi-core and many-core platforms, both relative to each other and in absolute terms, calculating computational performance and memory throughput. Details of test hardware can be found in Table I; the list includes a mid-range Intel Xeon machine (CPU 1), a high-end Xeon machine (CPU 2), an Intel Xeon Phi [4], and an NVIDIA K40 GPU [26]. In addition to theoretical performance numbers, as advertised by the vendors, we display results of standard benchmarks (STREAM for memory bandwidth with ECC off on the K40 and generic matrix–matrix multiply, GEMM for arithmetic throughput), which serve to illustrate how much of the theoretical peak can be realistically achieved. It is clear that achieving anything close to the theoretical peak – especially on accelerators – is quite difficult and often requires low-level optimizations and parameter tuning. We also highlight the FLOP/byte figures in Table I, which is essentially a metric for the balance of computational throughput and memory bandwidth on different architectures; the higher it is, the more bandwidth-starved the chip is. This is especially important for generally bandwidth-bound applications, such as the ones being investigated, because if their FLOP/byte ratio is much lower than that of the hardware, then a lot of computational power will be wasted.

For benchmarking, we chose two applications, both implemented in OP2; the first is Airfoil, a non-linear 2D inviscid Airfoil code that uses an unstructured grid [16], implemented in both single precision and double precision, and the second is Volna, a shallow water tsunami simulation code

Table I. Benchmark systems specifications.

System	CPU 1	CPU 2	Xeon Phi	K40
Architecture	2×Xeon E5-2640	2×Xeon E5-2697 v2	Xeon Phi 5110P	Tesla K40
Clock frequency (GHz)	2.4	2.7	1.053	0.87
Core count	2×6	2×12	61 (60 used)	2880
Last level cache (MB)	2×15	2×30	30	1.5
Peak bandwidth (GB/s)	2×42.6	2×59.7	320	288
Peak GFLOPS DP(SP)	2×120 (240)	2×259 (518)	1010 (2020)	1430 (4290)
Stream bandwidth (GB/s)	66.8	98.76	171	244
D/SGEMM GFLOPS	229 (433)	510 (944)	833 (1729)	1420 (3730)
FLOP/byte DP(SP)	3.42 (6.48)	5.43 (9.34)	4.87 (10.1)	6.35 (16.3)

Table II. Properties of Airfoil kernels; number of floating point operations and numbers transfers.

Kernel	Direct read	Direct write	Indirect read	Indirect write	FLOP	FLOP/byte DP (SP)	Description
save_soln	4	4	0	0	4	0.04 (0.08)	Direct copy
adt_calc	4	1	8	0	64	0.57 (1.14)	Gather, direct write
res_calc	0	0	22	8	73	0.3 (0.6)	Gather, colored scatter
bres_calc	1	0	13	4	73	0.5 (1.01)	Coundary
update	9	8	0	0	17	0.1 (0.2)	Direct, reduction

Table III. Properties of Volna kernels; number of floating point operations and numbers transfers.

Kernel	Direct read	Direct write	Indirect read	Indirect write	FLOP	FLOP/byte	Description
RK_1	8	12	0	0	12	0.6	Direct
RK_2	12	8	0	0	16	0.8	Direct
sim_1	4	4	0	0	0	0	Direct copy
compute_flux	4	6	8	0	154	8.5	Gather, direct write
numerical_flux	1	4	6	0	9	0.81	Gather, reduction
space_disc	8	0	10	8	23	0.88	Gather, scatter

Table IV. Test mesh sizes and memory footprint in double(single) precision.

Mesh	Cells	Nodes	Edges	Memory
Airfoil small	720 000	721 801	1 438 600	94 (47) MB
Airfoil large	2 880 000	2 883 601	5 757 200	373 (186) MB
Volna	2392352	1 197 384	3 589 735	n/a (355) MB

[17], implemented in only in single precision. Both simulations use a finite volume numerical algorithm, which is generally considered bandwidth-bound. Tables II and III detail the communication and computational requirements of individual parallel loops in Airfoil and Volna, in terms of useful data transferred (ignoring, for example, mapping tables) as number of floating-point values and useful computations (ignoring, for example, indexing arithmetic) for each grid point during execution. Transcendental operations (sin, cos, exp, and sqrt) are counted as one; these are present in `adt_calc` and `compute_flux`. Note, that these figures do not account for caching; therefore, in case of indirectly accessed data, off-chip transfers are reduced by data reuse, and the FLOP to byte ratio goes up.

Comparing the ratio of floating point operations per bytes transferred with those in Table I, it is clear that most of these kernels are theoretically bandwidth-bound; however, we have to account for the fact that several kernels may not be auto-vectorizing; therefore, for a fair comparison, the FLOP/byte ratios of CPU architectures have to be divided by the vector length (four in double and eight in single precision with 256-bit vectors). This pushes several kernels much closer to being compute-limited, which suggests that by applying vectorization, the code can be potentially further accelerated to the point where its entirely bandwidth-limited, eliminating any computational bottlenecks. In the following analysis, we primarily focus on achieved bandwidth, calculated based on the minimal (useful) amount of data moved; this assumes an infinite cache size for the duration of a single loop, which is of course unrealistic, but it gives a good idea of the efficiency of execution.

For Airfoil, performance is evaluated on two problem sizes; a 720k cell mesh and its quadrupled version, a 2.8M cell mesh, to investigate the sensitivity of the hardware to load balancing issues. For Volna, we use a real-world mesh with 2.5M cells, describing the northwestern coast of North America and the strait leading to Vancouver and Seattle, simulating a hypothetical tsunami originating in the Pacific Ocean. Mesh sizes and memory footprints are detailed in Table IV. These meshes are halved for OpenCL benchmarks on CPU 1 and CPU 2, because execution is restricted to a single socket because of limitations in the run-time discussed in Section 6.3.

6.2. *Baseline performance*

First, we briefly present and discuss the performance of the updated back-ends, described in Section 5, on Airfoil and Volna. This will serve as a baseline for analyzing the performance of vectorized code. We use the versions of computations that were not auto-vectorized by the compiler, which we saw was the case for most parallel loops (with one exception). The data and the computational requirements shown in Tables II and III give an idea of the cost of kernels, but several factors that influence performance are not immediately obvious. Several kernels, such as `save_soln`, `update`, `RK_1`, `RK_2` and `sim_1` are loops that only access data that is defined on the set being operated over; therefore, no indirection or coloring is necessary. In our two applications, these direct loops are computationally very simple, updating data, carrying out reductions, and so on. These kernels are clearly bandwidth-bound, with trivial access patterns, albeit no data reuse, making good use of pre-fetch engines on the CPU. The kernel `bres_calc` only operates over the boundary, indirectly accessing data on the interior set that is connected to the boundary; utilization – due to the small size of the boundary – is generally very low, as well as run-time. Because of its negligible contribution to the total execution time, we drop `bres_calc` from further analysis, as well as `sim_1`, which behaves the same as `RK_1` and `RK_2`.

The kernels `adt_calc` `compute_flux` and `numerical_flux` read and write data directly on their respective iteration sets (cells or edges) as well as reading values from different sets through mappings. While these kernels are indirect, no indirect increment takes place; therefore, coloring is not required. `adt_calc` and `compute_flux` include transcendental operations that are much more expensive than multiplications or additions. The throughput of the double precision square root instruction is 1 instruction per 44 clock cycles on the CPU, making it a very costly instruction compared with the one instruction per clock cycle of a multiplication; this further shifts these kernels towards being compute limited. Finally, `res_calc` and `space_disc` access data indirectly through mappings and have an indirect update that requires coloring – in the case of shared memory parallelism approaches, this means that the mini-partitions executed by OpenMP threads or CUDA thread blocks do not overlap with each other, which reduces data reuse and cache performance.

Figure 5 shows the baseline performance on Airfoil and Volna using the MPI back-end on CPUs 1 and 2, as well as the CUDA back-end on the K40 card. Estimated bandwidth and FLOP values are displayed for each kernel in Table V when running the double precision version of Airfoil and Volna on CPU 1 and the K40. On all hardware, direct kernels achieve a high percentage of peak bandwidth; however, for indirect kernels, both of these metrics are lower, especially on ones that require indirect incrementing of data. Because of expensive square root operations in `adt_calc` and the high number of computations on `compute_flux`, these kernels appear to be compute-bound on CPUs 1 and 2. Note that the overall runtime of Airfoil in double precision compared with

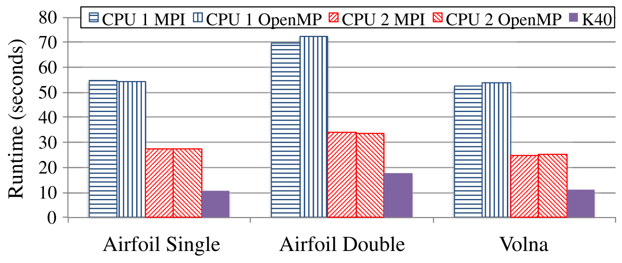


Figure 5. Baseline performance on Airfoil, in single precision and double precision (2.8M cell mesh), and Volna in single precision (2.4M cell mesh).

Table V. Useful bandwidth (BW – GB/s) and computational (Comp – GFLOP/s) throughput baseline implementations on Airfoil (double precision) and Volna (single precision) on CPU 1 and the K40 GPU.

	MPI CPU 1			MPI CPU 2			CUDA		
Kernel	Time	BW	Comp	Time	BW	Comp	Time	BW	Comp
save_soln	4	46	3.2	2.9	63	4	0.20	230	14
adt_calc	24.6	13	14.6	7.6	43	48	0.69	116	133
res_calc	25.2	27	32	13.6	50	61	2.77	62	75
bres_calc	0.09	29	12	0.05	52	16	0.06	32	5
update	14.05	56	8	9.7	81	10	0.83	235	29
RK_1	3.24	53	4	0.72	79	6	0.87	198	15
RK_2	2.88	59	5	0.64	89	9	0.72	242	24
compute_flux	23.34	14	42	4.01	27	82	3.21	101	309
numerical_flux	4.68	29	4	0.96	57	6	1.14	120	17
space_disc	16.86	21	9	1.51	79	33	1.92	73	31

single precision is much less than double, despite the fact that twice the amount of memory has to be moved. This supports our conclusion that without vectorization, some parts of the code are not limited by bandwidth, rather compute or latency. As we apply vectorization in the following sections, we will show that by increasing computational throughput, the performance is increasingly limited by bandwidth and latency.

6.3. *OpenCL performance on CPU and the Xeon Phi*

Because of a limitation on the tested Intel CPUs, neither AMD's nor Intel's OpenCL 1.2 driver (both support compilation for Intel CPUs) is currently able to select – using the device fission feature – a subset of processor cores to execute on a single NUMA socket. Because a fully operational MPI+OpenCL back-end is not yet available, the presented benchmark is limited to single socket performance comparisons. Scheduling threads to a single socket is enforced by the *numactl* utility. Based on the first touch memory allocation policy in the Linux kernel, it is certain that the master thread and the child threads – placed on the same socket – become memory allocated to the same NUMA memory region.

In the presented performance measurements, the one-time cost of run-time compilation is not counted. Only the time spent on effective computation and synchronization is shown. Figure 6 shows that OpenCL execution time in the CPU case is close to the plain OpenMP time. As opposed to conventional auto-vectorization, where segments of a code can be vectorized, OpenCL either vectorizes a whole kernel or none of it. Even though `adt_calc`, `bres_calc`, `compute_flux` and `numerical_flux` kernels are vectorized, the overall performance of the OpenCL implementation is not significantly better.

The Intel Offline Compiler [23] has been used to test whether kernels have been vectorized or not. Results are shown in Table VI. The extended capabilities of the instruction set on the Xeon Phi,

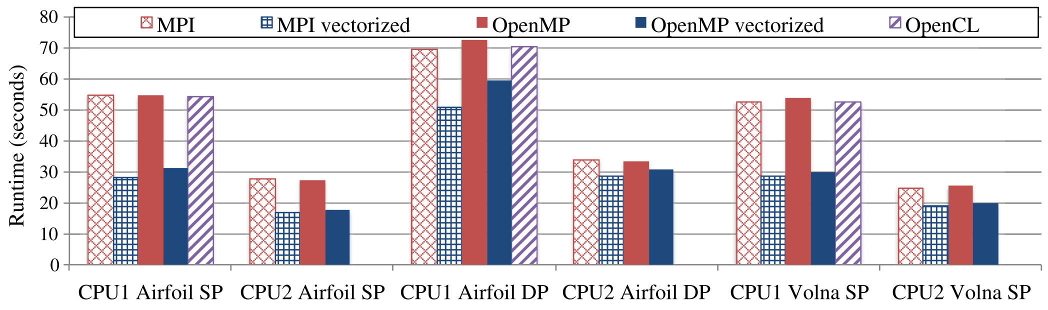


Figure 6. Vectorization with explicit vector intrinsics and OpenCL, performance results from Airfoil in single (SP) and double (DP) precision on the 2.8M cell mesh and from Volna in single precision on CPU 1 and 2.

Table VI. Timing and bandwidth breakdowns for the Airfoil benchmarks in double (single) precision on the 2.8M cell mesh and Volna using the OpenCL back-end on a single socket of CPU 1 and Xeon Phi. Also, kernels with implicit OpenCL vectorization are marked in the right columns.

	CPU 1		Xeon Phi		OpenCL vec	
Kernel	Time	BW	Time	BW	CPU	Phi
save_soln	4.15 (2.18)	44 (42)	2.6 (1.57)	71 (59)	–	✓
adt_calc	18.27 (13.23)	17.7 (12.2)	12.1 (7.2)	27 (22)	✓	✓
res_calc	31.43 (29.91)	22 (11.6)	46 (29.76)	15 (12)	–	✓
update	14.65 (7.34)	53.5 (53.4)	12 (6.5)	65 (60)	–	✓
RK_1	1.37	42	0.89	64	–	✓
RK_2	1.18	49	0.76	75	–	✓
compute_flux	6.4	51	4.91	67	✓	✓
numerical_flux	7.48	18	3.28	42	✓	✓
space_disc	9.24	40	7.95	45	–	✓

Table VII. Timing and bandwidth breakdowns for the Airfoil benchmarks in double (single) precision on the 2.8M cell mesh and Volna using the vectorized pure MPI back-end on CPU 1 and CPU 2.

	CPU 1		CPU 2	
Kernel	Time	BW	Time	BW
save_soln	4.08 (2.04)	45 (45)	2.9 (1.5)	62 (61)
adt_calc	12.7 (5.2)	25 (31)	5.6 (3.5)	57 (46)
res_calc	19.5 (13.5)	35 (26)	9.9 (7.1)	69 (48)
update	14.6 (7.1)	53 (56)	9.8 (4.7)	79 (83)
RK_1	3.27	52	2.19	78
RK_2	2.88	59	1.86	92
compute_flux	8.82	37	6	54
numerical_flux	4.59	30	3.18	43
space_disc	7.47	48	4.56	79

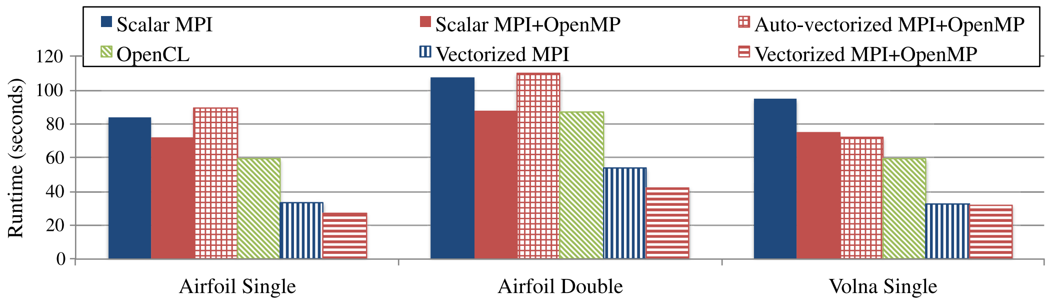


Figure 7. Performance of the Xeon Phi on Airfoil (2.8M cell mesh) and on Volna with non-vectorized, auto-vectorized, and vector intrinsic versions.

including the gather and scatter instructions, allow the compiler to vectorize more complex code. The AVX instruction set is more restrictive, and although the compiler could produce vector code, it refuses to do so if the heuristics predict worse performance. Even though the Intel OpenCL compiler can handle some branching in the control flow, optimization decisions may override these.

The kernel level breakdown of OpenCL in Table VI shows that the largest difference between the OpenMP and implicitly vectorized OpenCL comes from the `adt_calc` and `res_calc` kernels in Airfoil and from the `compute_flux` and `numerical_flux` in Volna. Even though `adt_calc` is vectorized by OpenCL, and is indeed faster than the non-vectorized version shown in Table V, the performance benefit is much smaller compared with the explicitly vectorized version shown in Table VII. The same is true for `compute_flux`. In the case of `numerical_flux`, the kernel was vectorized but the performance degraded. On the other hand, `space_disc` was not vectorized by OpenCL, but the performance still increased, but the manually vectorized kernel is still faster.

The OpenCL performance on the Xeon processors is satisfying compared with the non-vectorized OpenMP performance and even better in the case of the Xeon Phi co-processor. The results achieved with the co-processor are shown on Figure 7. Thus, OpenCL, as an environment implementing SIMT programming model that is amenable to vectorized execution, does deliver some performance improvement over previous non-vectorized implementations, but profiling shows that significant performance is lost because of scheduling. Furthermore, in comparison with results in the next section, we see that while some vectorization is achieved, especially on the Xeon Phi, it is currently not capable of creating efficient vector code.

6.4. *Vector intrinsics on CPUs*

With the parallel programming abstractions and methods described earlier (MPI, OpenMP, and OpenCL), the programmer has limited control over what becomes vectorized, but when using vector intrinsics, vectorization is explicit. Figure 6 shows performance results comparing the vectorized and the non-vectorized codes on CPU 1 and CPU 2. With one exception, the pure MPI versions are faster than the MPI+OpenMP versions, because at this scale, the overhead of MPI communications is almost completely hidden, while OpenMP has some overhead due to threading [27] and the colored execution.

Observe that the improvement in single precision is much larger than in double precision; this is because of the fact that twice as much data has to be transferred in double precision, but the length of the vector registers is the same; therefore, only half the number of floating point operations can be carried out at the same time. Another important observation is that comparing the run-times in single precision and double precision, we only see a 30–40% difference in the baseline in Figure 5 (recall that without vectorization, the computational throughput is the same in single precision and double precision), while the vectorized versions show an almost perfect 80–110% speedup when going from double precision to single precision. It is also important to note that vectorization has a much larger effect on performance on CPU 1, which is due to CPU 2 having over twice the computational capacity (2$\times$ the cores and higher frequency) but only 47% more bandwidth; at the same time, CPU 2 also has twice the cache size, resulting in improved data reuse, which is more of a bottleneck on Airfoil than on Volna.

Table VII shows per-loop breakdowns of the vectorized Airfoil benchmark on CPU 1 and CPU2, on the larger 2.8M mesh in order to minimize caching artifacts. Comparing double precision bandwidth values with that of Table V, it is clear that most direct kernels (`save_soln, update, RK_1` and `RK_2`) were already bandwidth-limited; therefore, their performance remains the same. Compute-heavy kernels `adt_calc` and `compute_flux` saw the highest increase in performance due to vectorization, although gains on CPU 2 are again lower, because of its much higher computational capacity already having saturated most of the available bandwidth. Kernels that indirectly read and write data, such as `res_calc` and `space_disc` also gain performance in most cases although significantly less because they are limited by the serialization of data accesses and the loss of parallelism, providing further evidence that these kernels were, to some extent, bound by latency. Switching to single precision should effectively half the run-time of different loops, because half the amount of data is moved, which matches the timings of direct loops, the kernel `adt_calc` gains further speedups (2.4 times) on CPU 1 by moving to single precision, because of the higher computational throughput, which hints at this kernel still being compute-limited in double precision on CPU 1 and on CPU 2, where theoretical computational throughput is 2.2 times higher, the speedup is much lower, indicating it is bound by bandwidth. The kernel `res_calc` is affected by a high number of gather and serialized scatter operations as well as insufficient data reuse resulting in superfluous data movement; moving to single precision only improves run-time by 30%; thus, this kernel is being limited by latency – from serialization as well as caching behavior. `space_disc` behaves similarly on CPU 1; however, because of the doubled size of the cache on CPU 2, much higher data reuse is achieved; therefore, its execution is bound by off-chip bandwidth.

6.5. *Vector intrinsics on the Xeon Phi*

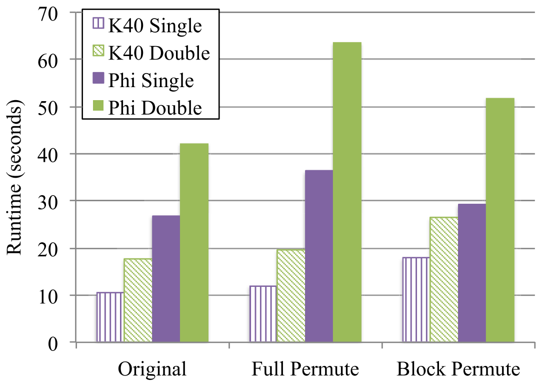
One of the main arguments in favor of the Xeon Phi is that applications running on the CPU are easily ported to the Phi. While this is true to some extent as far as compilation and execution goes, performance portability is a much more important issue. On the higher level, the balance of MPI and/or OpenMP is usually slightly different, and on a lower level, vectorization is needed more than ever to achieve high performance. Thanks to the gather and scatter instructions and given the use of alternate coloring schemes, most loops in our test applications do auto-vectorize. However, as we show auto-vectorized performance is poor, therefore we evaluate the use of intrinsics as well, and because the instruction set is not backwards-compatible, they have to be changed. Our approach of wrapping vectors in C++ classes, using constructors and operator overloading to hide vector intrinsic instructions permits us to generate the same code for both CPU and Phi vectorization, and then through compiler pre-processor macros, select classes that are appropriate for the hardware being used. Through this, we can exploit new features in the Phi, such as gather instructions and vector reduction and integrate it seamlessly into the OP2 toolchain.

To compile for the Phi, we used the `-O3 -mmic -fno-alias -mcmodel=medium -inline-forceinline` flags, and executed all applications natively on the device; we did not implement offload mode support because, as we show, pure OpenMP execution does not give the best performance and moving data between the host and the device for the purpose of hybrid MPI+OpenMP halo exchanges would result in unnecessary traffic. Tests included pure MPI execution and different combinations of MPI processes and OpenMP threads, setting `I_MPI_PIN_DOMAIN=auto`. Because of the nature of the OP2 OpenMP parallelization approach, there is no data reuse between threads, which is likely why we observed very little ($<3\%$) performance differences between the settings of `KMP_AFFINITY`; therefore, we only report results with the `compact` setting. In all MPI+OpenMP hybrid tests, the total number of OpenMP threads is 240 (60 cores, 4 threads each), as the number of MPI processes vary so does the number of OpenMP threads per process to give a total of 240; this number of total threads was found to consistently deliver the best performance.

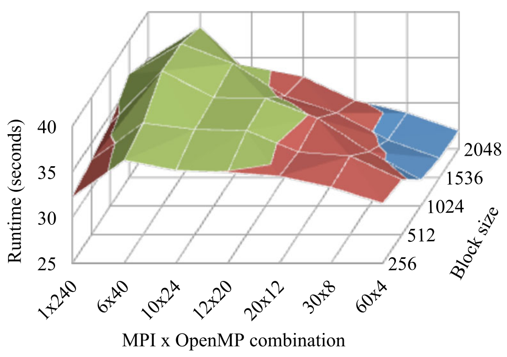
Figure 7 shows performance figures on the Xeon Phi, with Airfoil executing on the large (2.8M) mesh. It is clear that auto-vectorization approaches fail to deliver good performance, despite the fact that all kernels vectorized; although `res_calc` and `space_disc` use a different coloring scheme that permits vectorization of the gather-scatter loop. In a similar way to the CPU, vectorization using intrinsics gives a significant performance boost, but the difference is even higher, 2–2.2 times in single and 1.7–1.82 times in double precision. Unlike in the case of the CPU, the hybrid MPI+OpenMP gives better performance than the pure MPI version – which is due to the MPI messaging overhead becoming significant when the number of processes goes beyond 120. While on the CPU and the GPU, there was very little performance difference between the small and the large Airfoil meshes, comparing the run-times on the small and the large mesh, however, reveals that the Xeon Phi is actually quite sensitive to being fully utilized; while the problem size is four times bigger, it only takes 2.97 times more time in single and 3.25 times more time in double precision to finish the execution for the vectorized MPI+OpenMP version.

One of the main limiters of performance is the serialization due to coloring when indirectly incrementing data; in a vector processing setting, this can be especially harmful: for example, the Xeon Phi can store 16 single precision floating point values in a vector, so sequentially scattering these values can easily become a major bottleneck. Therefore, we have implemented two additional coloring approaches, as discussed in Section 3, in order to guarantee the independence of vector lanes; this makes it possible to use the scatter instruction on the Phi. However, it also implies that data previously accessed directly now has to be gathered, because the list of elements to be executed (that have the same color) are given by a permutation and also that there is not going to be any data reuse between elements of the same color. With the ‘full permute’ approach temporal locality is very poor, because all set elements of the same color are executed before proceeding to the next color, but parallelism is trivial. With the ‘block permute’ approach, only set elements in the same block are executed by color; therefore, temporal locality is significantly improved, but this execution scheme is more complicated and in case of small blocks, may suffer from the underutilization of vector lanes. Figure 8(a) shows the performance with different coloring approaches on the K40 and the Xeon Phi; out of the three the original approach – which causes serialization – still performs the best, but it is clear that the Phi has enough cache to support the temporal locality of the ‘block permute’ approach, although it is hit by the increased irregularity of memory accesses, whereas the K40’s cache is much too small; thus, the ‘full permute’ approach performs better because of simplified execution.

An important factor affecting the performance is the hybrid MPI+OpenMP setup – how many MPI processes and how many OpenMP threads each. This has non-trivial effects on the cost of communications, shared cache space, NUMA effects, and others. In addition to this, the size of mini-partitions or blocks formed and assigned by OP2 to OpenMP threads can be modified, trading off the number of blocks (load balancing) with block size (cache locality). Figure 8(b) shows the effects of varying these parameters on the performance on Airfoil in double precision. Observe that as the number of MPI processes increases, a larger block size is preferred, up to a point where the load imbalance is significant.



(a) Choice of coloring approach



(b) Tuning MPI+OpenMP combination and block sizes. Colors show height map.

Figure 8. Coloring approaches and parameter tuning on Airfoil (2.8M mesh). (a) Choice of coloring approach; (b) Tuning MPI+OpenMP combination and block sizes. Colors show height map.

Table VIII. Timing and bandwidth breakdowns for Airfoil (2.8M mesh) in double (single) precision and Volna using different MPI+OpenMP back-ends on the Xeon Phi.

	Scalar		Auto-vectorized		Intrinsics	
Kernel	Time	BW	Time	BW	Time	BW
save_soln	1.95 (1.01)	94 (92)	1.94 (1.02)	95 (90)	2.17 (1.11)	84 (83)
adt_calc	27.7 (25.5)	12 (6.3)	14.35 (13.1)	23 (12)	6.86 (3.54)	47 (45)
res_calc	48.8 (36.8)	14 (9.4)	84.03 (66.5)	8 (5)	27.22 (18.4)	25 (18)
update	11.8 (9.59)	66 (41)	8.33 (8.45)	94 (46)	8.77 (4.6)	89 (85)
RK_1	2.16	79	2.19	78	1.35	128
RK_2	2.37	70	3.24	53	1.32	130
compute_flux	32.1	10	29.3	11	10.95	30
numerical_flux	12.9	11	11.3	12	7.29	19
space_disc	23.6	15	24.5	15	9.93	36

Per-loop breakdowns for the Xeon Phi are shown in Table VIII for the best combination of MPI+OpenMP and block size in each case. The importance of vectorization is quite obvious; even direct loops with very little compute, such as `update`, `RK_1`, and `RK_2` gain performance, but the most profound improvement can be observed on indirect loops, such as `adt_calc`, `res_calc`, `compute_flux`, `numerical_flux`, and `space_disc`, which become 2–3 times faster. `adt_calc` is clearly bound by the expensive `sqrt` instructions in the scalar case, the intrinsic version removes this bottleneck, giving 4–7× speedup, to the point where it is bandwidth-bound. The auto-vectorized code performs poorly, despite the fact that with the exception of `save_soln` and `update` (these have for loops inside) each parallel loop is reported as vectorized; `res_calc` and `space_disc` use the 'block permute' approach to guarantee independence of `for` loop iterations. With the exception of `adt_calc`, all other loops either lose performance compared with the scalar version, or the gains are marginal.

An important factor influencing run-time is the time spent in MPI communications, either sending/receiving halo data in `res_calc`, `compute_flux`, `numerical_flux`, and `space_disc` or performing a global reduction in `update` and `numerical_flux` to compute the residual or the minimum timestep. Both of these operations result in implicit synchronization; the time spent actually transferring data is negligible compared with the time spent waiting for data to arrive because of different processes becoming slightly out of sync and then waiting for each other to catch up at a synchronization point. On the smaller Airfoil problem, this takes up to 30% of total run-time but is reduced to 13% on the larger problem, whereas on the CPU, this is only 7% and 4%, respectively. This points to load balancing issues, explaining some of the performance differences between the small mesh and the large mesh.

6.6. *Performance overview*

Having analyzed the performance of different platforms, we have exposed a number of bottlenecks that affect performance of different loops. We have seen that direct loops are generally bandwidth-bound on all hardware and achieve a high percentage of the measured peak bandwidth; 70–90% on the CPU, 60–75% on the Phi and 80–95% on the GPU. Notably, vectorization on the CPU does not increase the performance of these direct kernels, whereas on the Xeon Phi it does; this is due to the cache contention when accessing data one-by-one or as a vector.

When vectorization is not utilized, `adt_calc` and `compute_flux` loops are shown to be compute limited (in part due to the low throughput of `sqrt` operations). When vectorization in enabled `adt_calc` remains compute bound on the slower CPU, but it becomes bandwidth bound on CPU 2, the Xeon Phi and the K40. Most indirect loops (`res_calc, numerical_flux, space_disc`) gain performance by using vectorization, but they quickly become limited by latency, caching behavior and the serialized gather–scatter operations.

While the CPU and the GPU are largely unaffected by load balancing issues over MPI, as shown by minimal performance differences between the small and the large Airfoil problem, the Xeon Phi requires the larger problem to run efficiently; on the smaller test case, up to 30% of total run-time is spent in MPI.

Figure 9 shows the best performance results achieved on all platforms on Airfoil and the 2.8M mesh in both single and double precision as well as on Volna in single precision. The overall performance of the Xeon Phi is consistently similar to the slower CPU – while simple direct kernels do run faster than on either CPUs, indirect kernels are significantly slower, pulling the total run-time up. For the CPUs and the GPU, the performance differences are related to the differences in bandwidth; 40–80% between the two CPUs (with 50% bandwidth difference and 2× cache size) and the K40 is 2.5–3 times faster than CPU 1 but has over 3.5 times the bandwidth.

Relative performance of different loops compared with CPU 1 are displayed in Table IX and show how well different architectures handle the different degrees of irregularity in memory accesses that are present in these loops. Direct loops (`update`, `save_soln`, `RK_1`, `RK_2`) have regular mem-

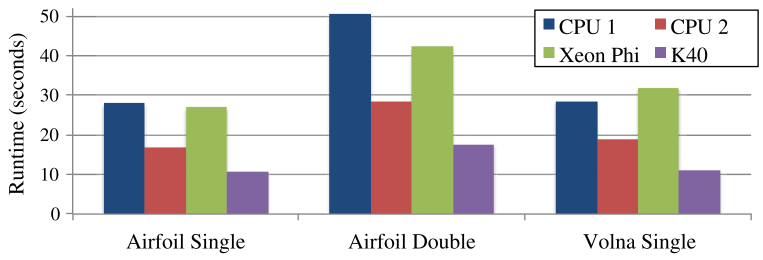


Figure 9. Comparison of the best execution times of the Airfoil benchmark on the 2.8M cell mesh and of Volna on different hardware.

Table IX. Relative performance improvements of different loops in Airfoil (double, 2.8M cell) and in Volna on different hardware.

Kernel	CPU 1	CPU 2	Xeon Phi	K40
save_soln	1.0	1.37	1.88	5.11
adt_calc	1.0	2.25	1.87	4.84
res_calc	1.0	1.95	0.81	1.79
update	1.0	1.48	1.67	4.54
RK_1	1.0	1.5	2.42	3.75
RK_2	1.0	1.54	2.18	4.05
compute_flux	1.0	1.46	0.81	2.75
numerical_flux	1.0	1.43	0.63	4.02
space_disc	1.0	1.63	0.75	1.52

ory accesses; therefore, performance follows the available bandwidth as measured in Table I; note how the GPU achieves a much higher fraction of the measured peak bandwidth; this is because of the fact that benchmarks on other platforms are tuned for that specific operation, similar techniques degrade overall performance in our applications.

Indirect loops deliver varying performance, depending on the number of operations, the degree of irregularity, the amount of data moved, and caching behavior; on the Phi and the K40, the performance increase is consistently less, especially on loops that indirectly increment data, because these are hit by serialization. While in absolute terms, this still results in the K40 GPU outperforming both CPUs; an application with even more irregular, scatter–gather kernels would see less overall benefit from the use of accelerators. Comparing the performance improvement on direct and indirect kernels, it is clear, that the longer the vectors are (for doubles, 4 on the CPU, 8 on the Phi and 32 on the GPU), the larger the performance penalty. On the other hand, the larger cache size and higher clock speed of CPU 2 enable it to be much more efficient on these indirect kernels than one would expect from the difference in available bandwidth.

7. CONCLUSIONS

Through execution schemes that lend themselves to auto-vectorization as well as implementing support for OpenCL and vector intrinsics, we evaluated approaches to achieving vectorized execution on modern Intel CPUs and the Xeon Phi. A key question was whether existing compiler-supported auto-vectorization approaches are sufficient for achieving vectorized execution for irregular computations, either through transforming execution schemes to provide loop iteration independence, or by using SIMT model for expressing parallelism. Furthermore, we have evaluated the use of vector intrinsics in order to guarantee vectorized execution.

Our results show that on CPUs, auto-vectorization does not happen for most loops, while the same code does vectorize on the Intel Xeon Phi. However, even then, performance is poor, often worse than scalar code. We have shown that OpenCL is adequately portable, but at the time of writing, the driver and the run-time are not very efficient. When compared with the simple (non-vectorized) OpenMP execution, run-time is only slightly better; even though some degree of vectorization is carried out, there is a large overhead coming from the run-time system. This is expected to improve with time and with the introduction of new instruction sets.

In the third approach, a code generation and a back-end system was set up in OP2 to support the use of vector registers and intrinsics. While this is the most involved amongst all parallel programming approaches supported in OP2, it does improve CPU performance significantly; speedups of 1.6–2.0 in single precision and 1.1–1.4 in double precision are observed. We show that by applying vectorization, the performance reaches the practical limits of the hardware, usually bound by bandwidth to DRAM or caching efficiency, and, in some cases, control or low-throughput arithmetic.

We have introduced intrinsics support for the Intel Xeon Phi as well and confirmed that vectorization is even more important than in the case of the CPUs, with speedups of 2.0–2.2 in single and 1.7–1.8 in double precision as a result of applying vectorization. Results show that while bandwidth-bound direct kernels and kernels that are compute-limited on the CPU run slightly faster on the Phi when compared with the CPU, kernels bound by scatter/gather type operations and serialization (due to race conditions) are significantly slower. To avoid the serialization issue, we introduce two additional coloring approaches that guarantee independence between vector lanes. However, these are affected by the increased irregularity and reduced data reuse and fail to improve performance. We also show that unlike other platforms, the Phi is very sensitive to load balancing issues that arise when solving smaller problems. Although its performance is on par with a mid-range dual-socket server CPU system, it is more difficult to use, and compared with the GPU, it is about 2.5 times slower. The source code used for testing is available in the OP2 repository [28].

This paper has shown that vectorization is absolutely essential to achieve maximum performance on modern CPUs, but higher level approaches, such as auto-vectorization by the compiler or

OpenCL, fail to produce efficient code in the case of unstructured grid applications; therefore, low-level vector intrinsics have to be used. We have also demonstrated that given the right high-level abstraction, it is possible to automatically generate vectorized code and achieve performance close to the practical limits of the hardware, which is increasingly the available bandwidth to off-chip memory.

ACKNOWLEDGEMENTS

This research has been funded by the UK Technology Strategy Board and Rolls-Royce Plc. through the Siloet project, the UK Engineering and Physical Sciences Research Council projects EP/I006079/1, EP/I00677X/1 on 'Multi-layered Abstractions for PDEs' and TÁMOP-4.2.1./B-11/2/KMR-2011-002, TÁMOP - 4.2.2./B-10/1-2010-0014. The authors would like to acknowledge the use of the University of Oxford Advanced Research Computing (ARC) facility in carrying out this work. We are thankful to Paul Kelly, Carlo Bertolli, Graham Markall, Fabio Luporini, David Ham and Florian Rathgeber at Imperial College London, Lawrence Mitchell at the University of Edinburgh for their contributions to the OP2 project, and András Oláh and Péter Szolgay at PPKE ITK Hungary.

REFERENCES

1. Top500 systems, 2013. (Available from: http://www.top500.org) [Accessed on 3 August 2015].
2. Dally B. Power, programmability, and granularity: the challenges of exascale computing. *2011 IEEE International Parallel Distributed Processing Symposium (IPDPS)*, Anchorage, Alaska, USA, 2011; 878–878.
3. What is GPU Computing, 2013. (Available from: http://www.nvidia.com/object/what-is-gpu-computing.html) [Accessed on 3 August 2015].
4. Skaugen K. Petascale to exascale: extending Intel's HPC commitment, 2011. ISC 2010 keynote. (Available from: http://download.intel.com/pressroom/archive/reference/ISC_2010_Skaugen_keynote.pdf).
5. Texas instruments multi-core TMS320C66x processor. (Available from: http://www.ti.com/c66multicore) [Accessed on 3 August 2015].
6. Lindtjorn O, Clapp R, Pell O, Fu H, Flynn M, Fu H. Beyond traditional microprocessors for geoscience high-performance computing applications. *Micro, IEEE* 2011; **31**(2):41–49.
7. Intel Math Kernel Library, 2013. (Available from: http://software.intel.com/en-us/intel-mkl) [Accessed on 3 August 2015].
8. Heinecke A, Vaidyanathan K, Smelyanskiy M, Kobotov A, Dubtsov R, Henry G, Shet AG, Chrysos G, Dubey P. Design and implementation of the LINPACK benchmark for single and multi-node systems based on Intel Xeon Phi coprocessor. *2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, Boston, Massachusetts USA, 2013; 126–137.
9. Rosinski J. Porting, validating, and optimizing NOAA weather models NIM and FIM to Intel Xeon Phi. *Technical Report*, NOAA, Silver Spring, Maryland, USA, 2013.
10. Brook RG, Hadri B, Betro VC, Hulguin RC, Braby R. Early application experiences with the Intel MIC architecture in a Cray CX1. *Cray User Group (CUG '12)*, Stuttgart, Germany, 2012; No. 194.
11. Vladimirov A, Karpusenko V. Test-driving Intel Xeon Phi coprocessors with a basic N-body simulation. *Technical Report*, Colfax International: Sunnyvale, California, USA, 2013. (Available from: http://research.colfaxinternational.com/post/2013/01/07/Nbody-Xeon-Phi.aspx) [Accessed on 3 August 2015].
12. Pennycook SJ, Hughes CJ, Smelyanskiy M, Jarvis SA. Exploring SIMD for molecular dynamics, using Intel Xeon processors and Intel Xeon Phi coprocessors. *2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, Boston, Massachusetts USA, 2013; 1085–1097.
13. Smelyanskiy M, Sewall J, Kalamkar D, Satish N, Dubey P, Astafiev N, Burylov I, Nikolaev A, Maidanov S, Li S, *et al.* Analysis and optimization of financial analytics benchmark on modern multi- and many-core IA-based architectures. *2012 SC Companion High Performance Computing, Networking, Storage and Analysis (SCC)*, Salt Lake City, Utah, USA, 2012; 1154–1162.
14. Kim S, Han H. Efficient SIMD code generation for irregular kernels. *SIGPLAN Notice* 2012; **47**(8):55–64.
15. Giles MB, Mudalige GR, Sharif Z, Markall GR, Kelly PHJ. Performance analysis and optimization of the OP2 framework on many-core architectures. *The Computer Journal* 2012; **55**(2):168–180.
16. Giles MB, Ghate D, Duta. MC. Using automatic differentiation for adjoint CFD code development. *Computational Fluid Dynamics Journal* 2008; **16**(4):434–443.
17. Dutykh D, Poncet R, Dias F. The {VOLNA} code for the numerical modeling of tsunami waves: generation, propagation and inundation. *European Journal of Mechanics - B/Fluids* 2011; **30**(6):598–615. Special Issue: Nearshore Hydrodynamics.
18. Mudalige GR, Giles MB, Thiyagalingam J, Reguly IZ, Bertolli C, Kelly PHJ, Trefethen AE. Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems. *Parallel Computing* 2013; **39**(11):669–692.

19. Rabenseifner R, Hager G, Jost G. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Weimar, Germany, 2009; 427–436.
20. Dagum L, Menon R. OpenMp: an industry standard API for shared-memory programming. *Computational Science Engineering, IEEE* 1998; **5**(1):46–55.
21. Scotch and PT-Scotch, 2013. (Available from: http://www.labri.fr/perso/pelegrin/scotch/) [Accessed on 3 August 2015].
22. Poole EL, Ortega JM. Multicolor ICCG methods for Vector computers. *SIAM Journal on Numerical Analysis* 1987; **24**(6):1394–1418.
23. Intel SDK for OpenCL applications: Intel, 2013. (Available from: http://software.intel.com/en-us/vcsource/tools/opencl-sdk) [Accessed on 3 August 2015].
24. Intel C++ Compiler XE 13.1 user and reference guide: Intel, 2013. (Available from: https://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/index.htm) [Accessed on 3 August 2015].
25. Giles MB, Mudalige GR, Spencer B, Bertolli C, Reguly I. Designing OP2 for GPU architectures. *Journal of Parallel and Distributed Computing* 2013; **73**:1451–1460.
26. NVIDIA Tesla Kepler GPU accelerators, 2012. (Available from: http://www.nvidia.com/object/tesla-servers.html) [Accessed on 3 August 2015].
27. Lindberg P. Basic OpenMP threading overhead. *Technical Report*, Intel, Santa Clara, California, USA, 2009. (Available from: http://software.intel.com/en-us/articles/basic-openmp-threading-overhead) [Accessed on 3 August 2015].
28. OP2 github repository, 2013. (Available from: https://github.com/OP2/OP2-Common) [Accessed on 3 August 2015].