Vectorizing Unstructured Mesh Computations for Many-core Architectures

I. Z. Reguly[†‡], E. László[†‡], G. R. Mudalige[†], M. B. Giles[†]
[†]Oxford e-Research Centre, University of Oxford
[‡] Faculty of Information Technology and Bionics, Pázmány Péter Catholic University
{istvan.reguly,endre.laszlo,gihan.mudalige}@oerc.ox.ac.uk
mike.giles@maths.ox.ac.uk

ABSTRACT

Achieving optimal performance on the latest multi-core and many-core architectures depends more and more on making efficient use of the hardware's vector processing capabilities. While auto-vectorizing compilers do not require the use of vector processing constructs, they are only effective on a few classes of applications with regular memory access and computational patterns. Irregular application classes require the explicit use of parallel programming models; CUDA and OpenCL are well established for programming GPUs, but it is not obvious what model to use to exploit vector units on architectures such as CPUs or the Xeon Phi. Therefore it is of growing interest what programming models are available, such as Single Instruction Multiple Threads (SIMT) or Single Instruction Multiple Data (SIMD), and how they map to vector units.

This paper presents results on achieving high performance through vectorization on CPUs and the Xeon Phi on a key class of applications: unstructured mesh computations. By exploring the SIMT and SIMD execution and parallel programming models, we show how abstract unstructured grid computations map to OpenCL or vector intrinsics through the use of code generation techniques, and how these in turn utilize the hardware.

We benchmark a number of systems, including Intel Xeon CPUs and the Intel Xeon Phi, using an industrially representative CFD application and compare the results against previous work on CPUs and NVIDIA GPUs to provide a contrasting comparison of what could be achieved on current many-core systems. By carrying out a performance analysis study, we identify key performance bottlenecks due to computational, control and bandwidth limitations.

We show that the OpenCL SIMT model does not map efficiently to CPU vector units due to auto-vectorization issues and threading overheads. We demonstrate that while the use of SIMD vector intrinsics imposes some restrictions, and requires more involved programming techniques, it does result in efficient code and near-optimal performance, that is up to 2 times faster than the non-vectorized code. We observe that the Xeon Phi does not provide good performance for this class of applications, but is still on par with a pair of high-end Xeon chips. CPUs and GPUs do saturate the available resources, giving performance very near to the optimum.

PMAM'14 February 15-19 2014, Orlando, FL, USA

http://dx.doi.org/10.1145/2560683.2560686 .

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: C.1.2 Multiple Data Stream Architectures

Keywords

Vectorization, Xeon Phi, AVX, CUDA, Unstructured Grid, OP2, Domain Specific Library

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, every 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. The high-end, systems such as large parallel high performance computing (HPC) clusters are developed with combinations of these processors and co-processors [22], 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.

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 accelerators such as GPUs (e.g. NVIDIA GPUs [14]), the Intel Xeon Phi [20] and similar co-processors (DSPs [1], FPGAs [11]) 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. AVX), and require special programming techniques in order get full utilization.

Many-core co-processors such as GPUs have brought with them new extensions (e.g. CUDA, 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 programming extension, 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 to that of CUDA - in part because it struggles with the performance portability of OpenCL codes and 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 a SIMD programming model, operations are carried out on a packed vector (64-512 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.

The need to specifically program with a SIMD model to utilize vectorization had been largely circumvented in previous generations of CPUs by auto-vectorizing compilers such as Intel's ICC/IFort. Even then, 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. On the other hand, utilizing vector units were not as important to gain good performance on earlier generation CPUs. However, with increasing vector lengths on current micro-processors, vector units are becoming a key hardware feature that is essential for gaining higher performance on any class of applications. The programming techniques that make it possible to utilize vectorization are thus becoming more and more important not only for CPU based systems, but also for 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 and may require 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 [12] or Linpack [7]. Several studies have been published that 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 [19, 3] molecular dynamics [23, 16] and finance [21]. While most of the computations in these applications were engineered to lend themselves easily to auto-vectorization due to the structure of the underlying problems, the use of low level vector programming was still required in many cases.

The focus of this paper is to present research into gaining higher performance through vectorization on CPUs and the Xeon Phi for a key class of applications, unstructured mesh based applications that have very irregular access patterns. Our aim is to present the challenges in achieving good vectorizing code for this class of applications, which goes beyond the simple structured and regular applications that are well documented in many-core computing literature. In support of this work we make use of the OP2 [4] 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. In this paper, OP2's code generation techniques are utilized to develop a range of platform specific vectorizing implementations for multi-core CPUs and the Intel Xeon Phi. We then benchmark these implementations and compare their performance with other (non-vectorized) parallelizations previously developed with OP2 for the same application. We show that it is extremely difficult to achieve vectorization on both CPUs and the Xeon Phi, often involving coding in very low-level intrinsics even when a high-level code generation framework such as OP2 is used to 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, OpenCL) and techniques (e.g. vector intrinsics) available for programming modern CPUs and accelerators/co-processors map to these execution models. A benchmark CFD application, Airfoil [5], previously developed with OP2 is used to illustrate the contrasting strategies required to achieve good vectorization. As a result of this work, we develop for Airfoil (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.

2. The resulting vectorized code is benchmarked on a number of CPU and many-core processor systems including two CPUs based on the latest generation of Intel's Sandy Bridge micro-architecture, and an Intel Xeon Phi. These results are compared against Airfoil's performance on NVIDIA GPUs (K40 and C2070) developed in previous work [13] 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.

3. 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 get vectorization, and Section 5 discusses the improvements

made to existing backends 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 top-most 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, on a multi-core processor, it is often difficult to maintain such a coarse level of parallelism due to 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.

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 an MPI process that maintains its own private memory block. This model, called Simultaneous Multi-Threading (SMT), executes a collection of processing streams operating largely independently, each with its own execution path, working on different chunks of shared data. Currently popular SMT implementations include OpenMP and Pthreads.

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 Single Instruction Multiple Thread (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 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. NVIDIA's CUDA language and the OpenCL standard are based on the SIMT model, and the latter maps to both CPU vector units and GPUs.

Finally, the Single Instruction Multiple Data (SIMD) execution and programming mode is used by the vector units on Intel CPUs and the Xeon Phi. While some programming models (such as OpenCL) do support a 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 Advanced Vector Extensions (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 or16 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 has 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.

When developing large-scale applications, these models have to be layered on top of each other to achieve high performance. However, managing the parallelization in each of these layers and across layers is extremely difficult. Additionally, the use of various programming models reduces the productivity of application developers and also requires intimate knowledge of different hardware.

3. THE OP2 LIBRARY FOR UNSTRUCTURED GRIDS

Recently, domain specific languages (DSLs) have been utilized to reduce the complexity of programing 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 [4], 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 grids 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 an other, with a given arity, e.g. each edge connects to 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.

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 [18], 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, that are colored based on potential data races [17]. 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.

When using a 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, however special care must be taken with scatter operations - we use a set element coloring approach within each block with explicit synchronization between colors to make sure no two threads update the same value in memory. Finally, when using a SIMD model, we assign adjacent set elements to vector lanes, however, unlike in the case of SIMT, we have to load data explicitly into vector registers, branching is restricted to *select* instructions, and data is also written back explicitly out of the vector registers, avoiding race conditions. Further details are provided in Section 4.2.

4. VECTORIZATION

Intel provides several programming approaches to achieve vectorization, the most trivial being the auto-vectorization features implemented in the Intel Compilers; this takes loops that iterate over arrays executing a "kernel function" and vectorizes them. However, it has some limitations, for example a kernel function with a non-trivial execution path, a loop-carried dependency or indirect memory accesses usually does not get auto-vectorized, because the compiler first and foremost has to produce correct code. Since access patterns in unstructured grid computations are irregular, this class of applications does not lend itself well to auto-vectorization over computations carried out on different set elements; data races cannot be determined statically. In this section we investigate OpenCL and vector intrinsics as avenues to getting vectorization, with a special attention to programmability considerations.

4.1 OpenCL

OpenCL (Open Computing Language) was introduced as an open standard language that is aimed to support the wide variety of heterogeneous computing platforms, including CPUs, GPUs, DPSs, FPGAs etc. Although in many technical details it is similar to the CUDA language, the framework is more restrictive, since a common subset of all the capabilities of the different hardware architectures have to be supported. Since OpenCL is a standard, the implementation of the driver and the runtime environment relies on the support of the hardware manufacturers. The underlying philosophy of the framework is based on the SIMT abstraction, the workload is broken up into large blocks - work-groups in OpenCL vernacular - each consisting of a few hundred or thousand work-items. The way work-items and work-groups are mapped to the hardware is influenced by the parallel capabilities of the device and the possible optimizations that can be carried out by the compiler or synthesizer. Creating a single portable parallel abstraction comes with a performance trade-off on different architectures, as it has to be translated to the parallel execution model supported by the hardware.

The way OpenCL is implemented on CPUs is different from the GPU implementation; a single CPU thread sequentially iterates over the work-items in a work-group. A number of work-items may be bundled into vector instructions if the compiler is capable of implicit (automatic) inter-kernel vectorization. Depending on the instruction set (SSE4.x, AVX, IMCI etc.) the number of work-items bundled is defined by the vector length of the ISA, eg. doing single precision floating point arithmetics with AVX instructions will allow packing the workload of 8 work-items into one instruction. Even though the OpenCL abstraction fully describes any concurrency issues that may arise, therefore vectorization of work-items would always be valid, just like in 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 etc. Since Intel's IMCI instruction set is more extensive than AVX it gives more flexibility for the compiler to produce vector code.

Spreading the work-load on CPUs is done by mapping work-groups onto CPU hardware threads. Each work-group

```
// Divide for loop of work−groups across CPU hardware threads
// similar to #pragma omp parallel for
for(wg=0; wg < get_num_groups(2)*
                get_num_groups(1)*
                get_num_groups(0); wg++)
  for(k=0; k<get_local_size(2); k++)
    for(j=0; j<get_local_size(1); j++)
      // Run vectorized for loop (similar to #pragma simd)
      for(i=0; i<get_local_size(0); i+=SIMD_LENGTH) {
        // Execute work−item
        kernel(...);
```

Figure 1: Pseudo-code for OpenCL work distribution and vectorization.

then is executed sequentially by mapping the 3D work-group to three nested loops over the length in the different dimensions. If possible, the innermost loop is then vectorized. A pseudo-code on Figure 1 explains the implementation in more details.

The task level parallelism at the level of the outermost loop with Intel OpenCL is implemented using Intel's TBB (Thread Building Blocks) library [2]. Although TBB offers an efficient solution for multi-threading, the overall overhead of scheduling work-groups is larger in OpenCL than in OpenMP; however this keeps improving.

OpenCL 1.2 offers a feature called Device Fission. This capability was introduced in OpenCL to improve memory locality, load balancing and partial use issues. A device can be divided into sub-devices that use common L1, L2, L3 or LLC cache levels or are located in the same NUMA node. Alternatively one can select the number of CPU threads to use. This way it is theoretically possible to handle the NUMA problem inside the OpenCL framework.

OpenCL uses run-time compilation, which allows for more advanced code optimization and run-time code generation. Parameters that are only known at runtime, can be passed to the JIT compiler as macro constants. In some cases this might help implicit vectorization.

Since CPUs - as opposed to GPUs - don't 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. Special attention needs to be payed to pointers: a global pointer (eg. *__global float* p*) inside a user kernel is placed in private memory by default and points to global space - it is equivalent to *__global float* __private p*). Thus, explicit use of the global modifier needs to be added. In OP2 we need to parse the function declaration of the user kernel and add these keywords to the arguments.

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 a 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; they would force splitting the 3 innermost loops in Figure 1. 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: since every work-group processes one block (mini partition) of the whole domain, some information about the block data structure can to be shared amongst the work-items of the work-group. Either this data is 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, since work-item 0 is executed before any other work-item in the group, due to the sequential execution model.

- Reductions: doing a reduction is straightforward due to 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, since 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.

4.2 Vector intrinsics

Our experiments have shown that the primary obstacle to getting auto-vectorization 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. Figure 2 shows a code snippet with a typical gather-type memory access that occurs when iterating over edges, accessing the x and y coordinates of the two nodes at the two ends of the edge. While it is possible to do packing without the use of vector types or intrinsics, with simple C arrays of explicit compile-time known size, the result is an array-of-arrays which is too complex for the compiler to auto-vectorize, furthermore it would require manipulation of the user kernel (inserting $[j]$ indices), which is something our code generator is not equipped to carry out reliably.

By using vector types however, it is possible to explicitly vectorize the user kernel, and this approach only requires changing scalar types to vector types in the user kernel, which in most cases is a matter of a simple find-and-replace during code generation. While operators are not defined for these types, the Intel Compiler includes a header file that defines container classes for these types, overloading some operators. By extending these classes it is possible to maintain the original simple arithmetic expressions in the user kernels, but instead of scalars they will now operate on vectors. Furthermore, with preprocessor macros, it possible to have a single source code generated for different vector lengths, and select one at compile-time. 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 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 to this problem.

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

```
//serial code
for (int i =0; i < set_size; i++) {
  double *n1 = coords[2*edge2node[i*2+0]];
  double *n2 = coords[2*edge2node[i*2+1]];
  //inlined user kernel
  double dx = n1[0] - n2[0];
  double dy = n1[1] - n2[1];
}
//code with packed data
for (int i =0; i < set_size; i+=2) {
  double n1[2][2]={{coords[2*edge2node[i*2+0]+0],
                    coords[2*edge2node[(i+1)*2+0]+0]},
                   {coords[2*edge2node[i*2+0]+1],
                    coords[2*edge2node[(i+1)*2+0]+1]}};
  for (int j = 0; j < 2; j++) {
    double dx = n1[0][j] - n2[0][j];
    double dy = n1[1][j] - n2[1][j];
  }
}
//code with intrinsics wrapped in C++ classes
#ifdef MIC
#define VEC_HALF 8
typedef Vec8d doublev;
class Vec8d : public F64vec8 {
...
  Vec8d(const double *p, const Vec8im &idx) {
    vec = _mm512_i32logather_pd(idx,p, 8);
  }
}
#endif
...
for (int i =0; i < set_size/VEC_HALF; i++) {
  //edge2node is transposed
  intv_half map1(&edge2node[VEC_HALF*i]);
  intv_half map2(&edge2node[VEC_HALF*i+set_size]);
  doublev n1[2] = {doublev(coords, map1),
                   doublev(coords+1, map1)};
  doublev n1[2] = {doublev(coords, map2),
                   doublev(coords+1, map2)};
  //inlined user kernel
  doublev dx = n1[0] - n2[0];
  doublev dy = n1[1] - n2[1];
}
```

Figure 2: Serial code, code prepared for auto-vectorization, and code with vector intrinsics

to utilize different instructions on different hardware - for example the IMCI has a gather intrinsic. In the case of mapping-based scatter operations it is necessary to avoid data races; while in case of the GPU we use colored updates, this proved to be inefficient on the Xeon Phi (which supports masked scatter operations), therefore we simply sequentially scatter data from the vector container.

Code resulting from the use of vector containers is rather verbose, and results in a much higher number of source lines due to explicit packing and unpacking of data. 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 aligned to the vector length, the main vectorized loop, and a scalar post-sweep to compute set elements left over.

5. OPTIMIZATIONS TO EXISTING BACKENDS

Results presented in previous papers [4, 6] report pure MPI and hybrid MPI+OpenMP performance on clusters of CPUs as well as MPI+CUDA performance results running on Fermi-generation NVIDIA GPUs. However, as newer hardware generations become available, it is necessary to revise optimization techniques due to changing performance characteristics and best practices; for example the Kepler generation of GPUs features a much higher number of cores per Scalar Multiprocessor (SMX) than the Fermi generation, but the amount of shared memory available remains unchanged. Therefore the 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 is 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".

Futhermore, we carried out an in-depth investigation of our MPI and OpenMP backends 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. The generated code for OpenMP execution was similarly optimized and simplified, and explicit staging of data is no longer used.

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 and memory throughput. Details of test hardware can be found in Table 1; the list includes a mid-range Intel Xeon machine, a high-end Xeon machine, an Intel Xeon Phi [20] and an NVIDIA K40 [8] GPU. In addition to theoretical performance numbers, as advertised by the vendors, we display results of standard benchmarks (stream for memory bandwidth (ECC off) and generic matrix-matrix multiply (GEMM) for arithmetic throughput) which serves 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.

As a benchmark application we chose Airfoil, which is a non-linear 2D inviscid airfoil code that uses an unstructured grid [5], implemented in both single and double precision. The simulation uses the finite volume discretization, which is generally considered bandwidth-bound. Table 2 details the communication and computational requirements of individual parallel loops in Airfoil, in terms of useful data transferred (ignoring e.g. mapping tables) as number of floating-point values and useful computations (ignoring e.g. indexing arithmetics) for each grid point during execution. Transcendental operations (sin, cos, exp, sqrt) are counted

Table 1: Benchmark systems specifications

System	CPU 1	CPU 2	Xeon Phi	K40
Architecture	2×Xeon E5-2640	2×Xeon E5-2680	Xeon Phi 5110P	Tesla K40
Clock frequency	2.4 GHz	2.7 GHz	1.053 GHz	0.87 GHz
Core count	2×6	2×8	61 (60 used)	2880
Last level cache	2×15MB	2×20MB	30MB	1.5MB
Peak bandwidth	2×42.6 GB/s	2×51.2 GB/s	320 GB/s	288 GB/s
Peak compute DP(SP)	2×120(240) GFLOPS	2×172(345) GFLOPS	1.01 (2.02) TFLOPS	1.43 (4.29) TFLOPS
Stream bandwidth	65 GB/s	78 GB/s	171 GB/s	229 GB/s
D/SGEMM throughput	188(414) GFLOPS	304(614) GFLOPS	833(1729) GFLOPS	1420(3730) GFLOPS
FLOP/byte achieved DP(SP)	2.9(6.37)	3.9(7.81)	4.87(10.1)	6.35(16.3)
FLOP/byte w/o vectorization	0.73(0.79)	0.98(0.97)	0.6(0.63)	N/A

Table 2: 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	8	4	0	0	4	0.04(0.08)	Direct copy
adt_calc	5	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)	Gather, colored scatter, boundary
update	13	8	0	0	17	0.1(0.2)	Direct update and reduction

Table 3: Airfoil mesh sizes and memory footprint in double(single) precision

Mesh	cells	nodes	edges	memory
small	720000	721801	1438600	94(47) MB
large	2880000	2883601	5757200	373(186) MB

as one. 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 to those in Table 1 it is clear that these kernels are theoretically bandwidth-bound, however, we have to account for the fact that indirect kernels in Airfoil are not vectorizing, therefore, for a fair comparison, the FLOP/byte ratios of CPU architectures have to be divided by the vector length (4 in double and 8 in single precision with 256 bit vectors). This pushes `res_calc` and `adt_calc` 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.

Performance on two problem sizes is evaluated, a 720k cell mesh and its quadrupled version, a 2.8M cell mesh. Mesh sizes and memory footprints are detailed in Table 3. These meshes are halved for OpenCL benchmarks on CPU 1 and CPU 2, because execution is restricted to a single socket due to limitations in the runtime discussed in Section 4.1.

6.2 Baseline performance

First, we evaluate the performance of the old and the updated backends, described in Section 5, which will serve as a baseline for analysing the performance of vectorized code. Our main focus is on the double precision version of the Airfoil benchmark, which is most representative of larger-scale CFD codes. The data and the computational requirements shown in Table 2 give an idea of the cost of kernels, several factors that influence performance are not immediately obvious. The kernels `save_soln` and `update` are loops that only access data that is defined on the set being operated over, therefore no indirection or coloring is necessary. However, they are computationally very simple; the first only copies data from one array to an other, the second also updates flow variables based on fluxes as well as carrying out a reduction to compute the r.m.s. value. These kernels are clearly bandwidth-bound, with trivial access patterns, albeit no data reuse, making good use of prefetch 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 runtime. This kernel poses challenges in terms of compilation due to branching, but its contribution to the total runtime is negligible, therefore we drop it from further analysis in the rest of the paper. The kernel `adt_calc` reads and writes data directly on the iteration set (cells) as well as reading coordinate values from the four nodes connected to a cell. While this kernel is indirect, no indirect write takes place, therefore coloring is not required. `adt_calc` includes transcendental operations which 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 and on the order of 32 on the GPU, making it a very costly instruction compared to the 1 instruction per 1 clock cycle of a multiplication; this further shifts the kernel towards being compute limited. Finally, `res_calc` accesses all data indirectly through mappings and has an indirect update which 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. Furthermore, with vector programming models (CUDA and AVX), this requires safe update out of a vector register or a CUDA thread block which introduces serialization and/or branching. The operations carried out in `res_calc` are simple (multiplications and additions only) which can easily push this kernel to being limited by control and caching behavior.

Figure 3 shows the performance of the old and the updated pure MPI backend, the MPI+OpenMP backend (run in hybrid mode to avoid NUMA [9] effects) and the CUDA backend with Fermi optimizations on a Fermi card (C2070) and the Kepler card (K40) as well as with the Kepler optimiza-

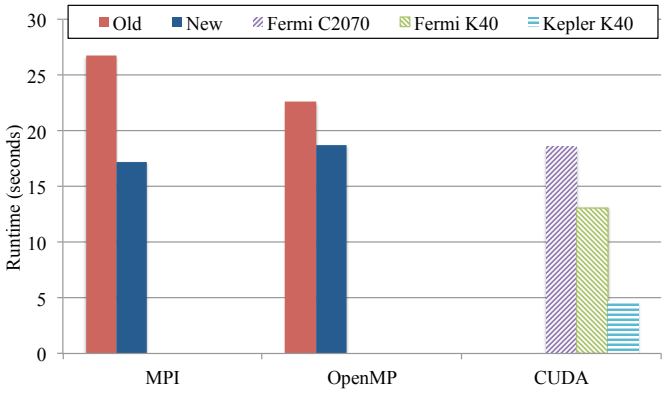

Figure 3: Performance of old and updated backends in double precision on the 720k cell mesh, including execution of Fermi-optimized code on Fermi hardware (C2070), Kepler hardware (K40), and Kepler-optimized code on Kepler hardware (K40)

Table 4: Bandwidth (BW - GB/s) and computational (Comp - GFLOP/s) throughput of updated but not vectorized backends on the Airfoil benchmark in double precision on the 720k cell mesh on CPU 1 and the K40 GPU

Kernel	pure MPI			CUDA		
	Time	BW	Comp	Time	BW	Comp
save_soln	0.99	46.55	2.9	0.20	230	14.4
adt_calc	6.3	18.24	14.6	0.71	161.2	129.2
res_calc	6.58	56.72	31.87	2.8	133.4	74.95
bres_calc	0.03	27.15	13.62	0.03	26.3	13.2
update	3.23	60.62	7.57	0.85	228	28.52

tions on the K40 card. Observe that the pure MPI backend gained a significant speedup due to the enabling of compiler optimizations, while the OpenMP backend was sped up only slightly as the code generated was simplified. The most profound difference however, can be observed when updating the GPU backend; simply compiling and running the Fermi optimized code on a Kepler-generation GPU hardly increases performance despite the much higher theoretical throughput of the hardware; this is due to different architectural changes from one generation to the next. Applying a new set of optimizations dramatically increases performance, giving an almost 3× speedup over the previous generation GPU. Estimated bandwidth and FLOP values are displayed for each kernel in Table 4 for the updated backends when running the double precision version of Airfoil on CPU 1. While the GPU achieves a high percentage of streaming bandwidth, showing a good balance in performance between the kernels, the CPU versions clearly show a very low throughput for `adt_calc`.

6.3 OpenCL performance on CPUs

Due to a limitation on the tested Intel CPUs, neither AMD's nor Intel's OpenCL 1.2 driver is currently able to select - using the device fission feature - a subset of processor cores to execute on a single NUMA socket. Since a fully operational MPI+OpenCL backend 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 - get memory allocated to the same NUMA memory region.

Table 5: Implicit vectorization of user kernels by Intel OpenCL.

User Kernel	save_soln	adt_calc	res_calc	bres_calc	update
AVX	–	✔	–	✔	–
IMCI	✔	✔	✔	✔	✔

In the presented performance measurements the one-time cost of run-time compilation is not counted. Only the time spent on effective computation and synchronisation is shown. Figures 4 and 5 show that OpenCL execution time is close to the plain OpenMP time, and somewhat better for the larger problem size. 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` and `bres_calc` kernels are vectorized, the overall performance of the OpenCL implementation is not significantly better.

The Intel Offline Compiler [2] has been used to test whether kernels with AVX or IMCI instruction set have been vectorized or not. Results are shown in Table 5. The extended capabilities of the IMCI ISA, 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 6 shows that the largest difference between the explicitly vectorized OpenMP and implicitly vectorized OpenCL comes from the `adt_calc` and `res_calc` kernels. Even though `adt_calc` is vectorized by OpenCL, and is indeed faster than the non-vectorized version shown in Table 4, the performance benefit is much smaller compared to the explicitly vectorized version.

Although OpenCL performance on the Xeon processors is satisfying, compared to the non-vectorized OpenMP performance, the Xeon Phi has major performance issues. It is important to notice that the Phi gains more performance as the problem size increases; for the 720k cell mesh the speed difference compared to the explicitly vectorized code is significant. For the 2.8M problem the difference is not as significant and it is even better then the non-vectorized code. Even though all the kernels are vectorized, the overall performance is far from the expected. The assembly code produced by the OpenCL compiler didn't have prefetch instructions inserted automatically. Although manual prefetching was forced, it didn't improve the performance. Moreover, the Intel VTune profiler shows significant time spent in the TBB Scheduler. These observations suggest that the overhead of initializing and executing a kernel with the task level parallelism available in TBB is very costly at the time of writing.

6.4 Vector Intrinsics on CPUs

With the parallel programming abstractions and methods described above (MPI, OpenMP and OpenCL), the programmer does not have direct control over what gets vector-

Table 6: Timing and bandwidth breakdowns for the Airfoil benchmarks in double(single) precision using the OpenCL backend on a single socket of CPU 1, with problem sizes halved

Kernel	720k/2 cells		2.8M/2 cells	
	Time	BW	Time	BW
save_soln	1.13(0.43)	41(53)	4.15(2.18)	44(41)
adt_calc	5.00(3.61)	23(16)	18.27(13.23)	26(18)
res_calc	8.74(7.92)	42(26)	31.43(29.91)	47(28)
update	3.76(1.68)	52(58)	14.65(7.34)	53(53)

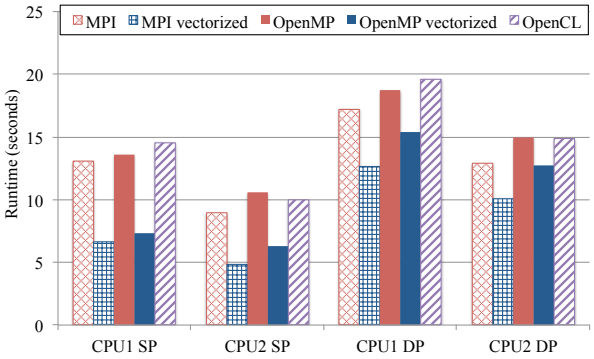

Figure 4: Vectorization with explicit vector intrinsics and OpenCL, performance results of the Airfoil benchmark in single (SP) and double (DP) precision on CPU 1 and 2 using the 720k cell mesh

ized, when using vector intrinsics, vectorization is explicit. Figures 4 and 5 show performance results comparing the vectorized and the non-vectorized code on CPU 1 and CPU 2 in both single and double precision on a mesh with either 720k or 2.8M cells. The pure MPI versions are consistently 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 [10] and the colored execution.

Observe that the improvement in single precision is 67-97%, but in double precision it is only 15-37%; this is due to 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 runtimes in single and double precision, we only see a 30-40% difference in the baseline (recall that without vectorization the computational throughput is the same in single and double precision), while the vectorized version shows an almost perfect 80-110% speedup when going from double to single precision.

Table 7 shows per-loop breakdowns of the vectorized Airfoil benchmark on CPU 1, both single and double precision, and both problem sizes. Comparing double precision bandwidth values with that of Table 4, it is clear that the two direct kernels (`save_soln` and `update`) were already bandwidth-limited, therefore their performance remains the same, however, `adt_calc` almost doubled in performance, and `res_calc` also sped up by 30%, providing further evidence that these kernels were, to some extent, bound by compute throughput. Switching to single precision should effectively half the runtime of different loops, because half

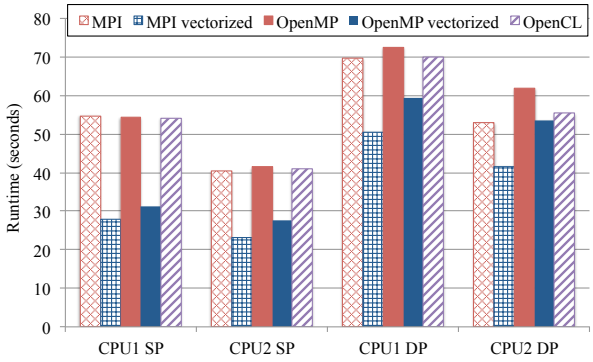

Figure 5: Vectorization with explicit vector intrinsics and OpenCL, performance results of the Airfoil benchmark in single (SP) and double (DP) precision on CPU 1 and 2 using the 2.8M cell mesh

Table 7: Timing and bandwidth breakdowns for the Airfoil benchmarks in double(single) precision using the vectorized pure MPI backend on CPU 1

Kernel	720k cells		2.8M cells	
	Time	BW	Time	BW
save_soln	1.01(0.28)	45(82)	4.1(2.0)	45(45)
adt_calc	3.3(1.33)	34(44)	12.7(5.2)	37(46)
res_calc	5.06(3.5)	73(59)	19.5(13.5)	76(62)
update	3.33(1.5)	59(65)	14.6(7.0)	54(56)

the amount of data is moved, which matches the timings of direct loops on the large mesh, however, superlinear scaling can be observed on the small mesh on `save_soln`; this is due to the fact that this kernel follows `update`, and modifies a subset of the same data, some of which remains in the cache. On the large mesh this artefact disappears. The kernel `adt_calc` gains further speedups (2.4 times) by moving to single precision, due to the higher computational throughput, which hints at this kernel still being compute-limited in double precision on CPU 1 - on CPU 2, where theoretical computational throughput is 40% higher, the speedup is near the expected, 1.9 times. The kernel `res_calc` is affected by a high number of gather and serialized scatter operations, moving to single precision only improves runtime by 30%, which hints at this kernel being limited by control and caching behavior.

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 trivially 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. As long as one depends on compiler auto-vectorization this is automatic to some extent, but as we have discussed, unstructured gird algorithms do not auto-vectorize well. Therefore, the use of intrinsics is necessary, and since 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 preprocessor macros select classes that are

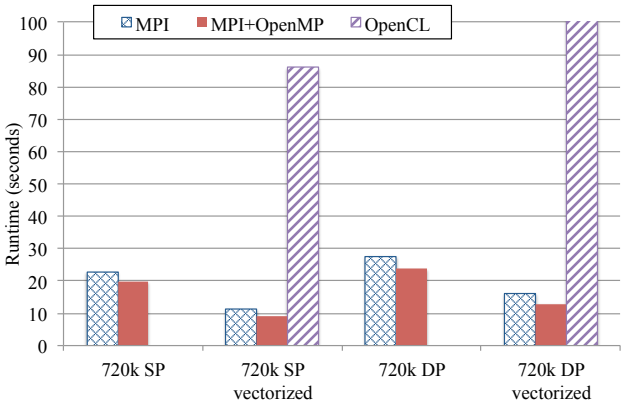

Figure 6: Performance of the Xeon Phi on the 720k cell mesh

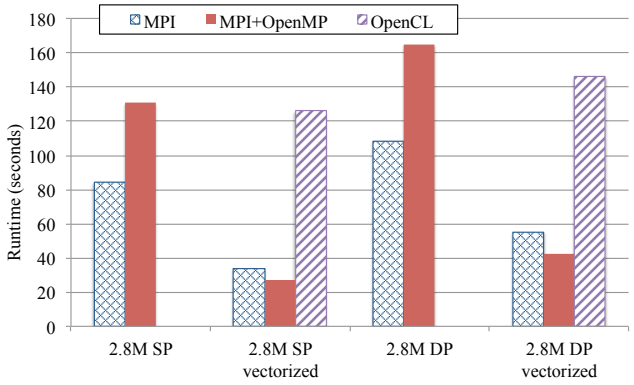

Figure 7: Performance of the Xeon Phi on the 2.8M cell mesh

appropriate for the hardware being used. Through this, we can exploit new features in the Phi, such as gather instructions, vector reduction, etc., and integrate it seamlessly into the OP2 toolchain. We had to override the `malloc, free` and `realloc` functions to ensure allocations are aligned to a 512 bit boundary (which does not happen automatically), so the aligned load instructions could be used when appropriate.

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, offload mode is currently not supported, because it requires extensive changes to the backend in order to manage halo exchanges. Tests included pure MPI execution and different combinations of MPI processes and OpenMP threads, setting $I\_MPI\_PIN\_DOMAIN = auto$. Due to 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.

Figure 6 shows performance figures on the Xeon Phi, solving the 720k cell mesh. Similar to the CPU, vectorization 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

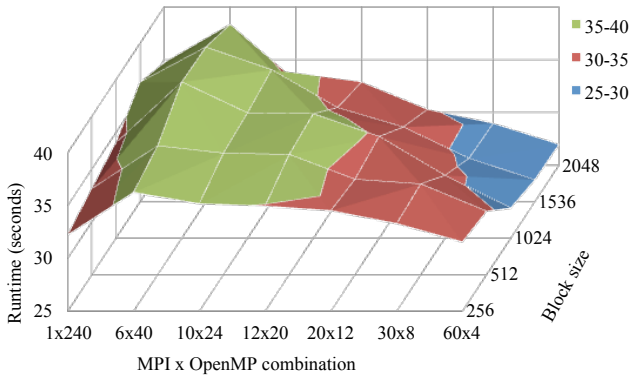

Figure 8: Performance on the Xeon Phi when varying OpenMP block size and the MPI+OpenMP combination on the 2.8M cell mesh in double precision

Table 8: Timing and bandwidth breakdowns for the Airfoil benchmarks in double(single) precision using the vectorized MPI+OpenMP backend on the Xeon Phi

Kernel	720k cells		2.8M cells	
	Time	BW	Time	BW
save_soln	0.58(0.25)	80(90)	2.18(1.18)	84(76)
adt_calc	2.0(1.16)	57(50)	6.83(3.33)	68(72)
res_calc	7.52(5.47)	45(38)	24.1(17.4)	61(48)
update	2.55(1.93)	77(50)	8.77(4.69)	89(83)

version - which is due to the MPI messaging overhead becoming significant when the number of processes goes beyond 120. Quadrupling the problem size to 2.8M cells shown an even bigger gap between the non-vectorized and the vectorized version, as shown in Figure 7, while the pure MPI version speeds up by 2.5 in single and 1.95 in double precision, for MPI+OpenMP the difference is 4.8 and 3.9 times, with the non-vectorized version performing unreasonably slow, the reasons for which are currently not well understood. Comparing the runtimes of 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. As a comparison, these figures are almost exactly 4 times (or slightly higher due to caching effects) on the CPU, and 3.85 times on the GPU.

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 to threads by OpenMP can be modified, trading off the number of blocks (load balancing) with block size (cache locality). Figure 8 shows the effects of varying these parameters on the performance, 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.

Per-loop breakdowns for the Xeon Phi are shown in Table 8 for the best combination of MPI+OpenMP in each case. Performance is generally slightly better compared to CPU 1 (Table 7). `adt_calc` does not show signs of being

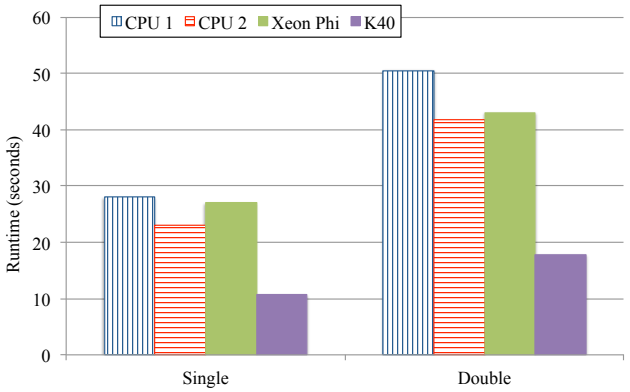

Figure 9: Comparison of execution times of the Airfoil benchmark on the 2.8M cell mesh on different hardware

compute-limited, but the performance penalty from gather operations and serialized scatter operations in `res_calc` is more severe, making this loop significantly slower than on the CPU. An important factor influencing runtime is the time spent in MPI communications, either sending/receiving halo data in `res_calc` or performing a global reduction in `update` to compute the residual. Both of these operations result in implicit synchronization; the time spent actually transferring data is negligible compared to the time spent waiting for data to arrive due to different processes getting slightly out of sync and then waiting for each other to catch up at a synchronization point. On the smaller problem this takes up to 30% of total runtime, 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 and the large mesh.

6.6 Performance overview

Having analysed 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 bandwidth-bound on all hardware regardless of using vectorization or not. The `adt_calc` loop is shown to be compute limited (due to the low throughput of *sqrt* operations) when vectorization is not utilized, and remains so on the slower CPU, but becomes bandwidth bound on CPU 2, the Xeon Phi and the GPU. Finally, `res_calc` is generally limited by gather and the serialized scatter operations on all hardware. While the CPU and the GPU are largely unaffected by load balancing issues, as shown by minimal performance differences between the small and the large problem, the Xeon Phi requires the larger problem to run efficiently.

Figure 9 shows performance results on all platforms on the 2.8M mesh in both single and double precision. The performance of the Xeon Phi is consistently somewhere between the faster and the slower CPU - while simple direct kernels do run faster than on either CPUs, `res_calc` is significantly slower, pulling the total runtime up. For the CPUs and the GPU the performance differences are related to the differences in bandwidth; 13-18% between the two CPUs (with 16% bandwidth difference) and the K40 is 2.1-2.4 times faster than CPU 2, but has almost three times the bandwidth.

Relative performance of different loops compared to CPU 1 are displayed in Table 9 and show how well different architectures handle the different degrees of irregularity in memory accesses that are present in these loops. Direct loops, such as `update` and `save_soln` have regular memory accesses, therefore performance follows the available bandwidth. `adt_calc` indirectly gathers and directly writes, it utilizes caches efficiently, performance improvements are in line with the improvements on direct kernels (except between CPU 1 and 2, due to the limited throughput of `sqrt`). The kernel with the most irregular memory accesses - `res_calc` - suffers from serialization; the longer the vector units are, the more severe the performance degradation: the improvement on the Xeon Phi is 2.3 times less than that of a direct loop, and on the GPU 2.8 times. While in absolute terms this still results in the GPU outperforming the CPU, an application with even more irregular, scatter-gather kernels would see less overall benefit from the use of accelerators.

Table 9: Relative timing breakdowns for different loops of the Airfoil benchmark in double precision on the 2.8M cell mesh on different hardware

Kernel	CPU 1	CPU 2	Xeon Phi	K40
save_soln	1.0	1.04	1.88	5.11
adt_calc	1.0	1.43	1.87	4.67
res_calc	1.0	1.33	0.81	1.79
update	1.0	1.03	1.67	4.49

7. CONCLUSIONS

In this paper we have started by presenting a study of parallel programming abstractions and how the OP2 abstraction for unstructured grids can map onto them. By utilizing all levels of parallelism, ranging from distributed memory using MPI, through shared memory multithreading using OpenMP, to either SIMT using CUDA and OpenCL or SIMD using explicit vector intrinsics, we have shown how OP2 can map execution to this multi-level parallel setting. Through supporting OpenCL and vector intrinsics we aimed to enable vectorized execution on modern Intel CPUs and the Xeon Phi. One of the main questions was whether a SIMT program can be efficiently compiled to SIMD machine code through Intel's support for the OpenCL language. We have shown that while OpenCL is adequately portable, at the time of writing the driver and the runtime are not very efficient - this is expected to improve with time and with the introduction of new instruction sets. When compared to the simple (non-vectorized) OpenMP execution, runtime is only slightly better; even though some degree of vectorization is carried out, there is a large overhead coming from the runtime system.

A code generation and a backend system was set up in OP2 to support the use of vector registers and intrinsics. While this approach is the most involved amongst all parallel programming approaches supported in OP2, it does improve CPU performance significantly; speedups of 1.7-2.0 in single and 1.15-1.4 in double precision are observed. We show that by applying vectorization, the performance is pushed very close to the practical limits of the hardware, usually bound by bandwidth to DRAM, but in some cases control or low-throughput arithmetics.

We have introduced 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 2-2.2 in single and 1.7-1.82 in double precision as a result of ap-

plying 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 to the CPU, the kernel bound by scatter/gather type operations and serialization (due to race conditions) is significantly slower. We also show that unlike other platforms, the Phi is very sensitive to load balancing issues that arise when solving smaller problems. While its performance is on par with server-grade dual-socket CPU systems, it is more difficult to use, and compared to the GPU it is about 2.5 times slower. All the source code used for testing is available in the OP2 repository [15].

This paper has shown that vectorization is absolutely essential to achieving 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 show 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.

8. ACKNOWLEDGMENTS

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.

9. REFERENCES

[1] Texas Instruments Multi-core TMS320C66x processor. `http://www.ti.com/c66multicore`.

[2] Intel SDK for OpenCL Applications, 2013. `http://software.intel.com/en-us/vcsource/tools/opencl-sdk`.

[3] R. G. Brook, B. Hadri, V. C. Betro, R. C. Hulguin, and R. Braby. In *Cray User Group (CUG), 2012*, 2012.

[4] M. Giles, G. Mudalige, Z. Sharif, G. Markall, and P. H. J. Kelly. Performance analysis and optimization of the OP2 framework on many-core architectures. *The Computer Journal*, 55(2):168–180, 2012.

[5] M. B. Giles, D. Ghate, and M. C. Duta. Using Automatic Differentiation for Adjoint CFD Code Development. *Computational Fluid Dynamics Journal*, 16(4):434–443, 2008.

[6] M. B. Giles, G. R. Mudalige, B. Spencer, C. Bertolli, and I. Reguly. Designing OP2 for GPU Architectures. *Journal of Parallel and Distributed Computing*, 73:1451–1460, November 2013.

[7] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. Shet, G. Chrysos, and P. Dubey. Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems Based on Intel Xeon Phi Coprocessor. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 126–137, 2013.

[8] NVIDIA Tesla Kepler GPU Accelerators, 2012. `http://www.nvidia.com/object/tesla-servers.html`.

[9] R. P. LaRowe and C. S. Ellis. Page placement policies for NUMA multiprocessors. *Journal of Parallel and Distributed Computing*, 11(2):112 – 129, 1991.

[10] P. Lindberg. Basic OpenMP threading overhead. Technical report, Intel, 2009. `http://software.intel.com/en-us/articles/basic-openmp-threading-overhead`.

[11] O. Lindtjorn, R. Clapp, O. Pell, H. Fu, M. Flynn, and H. Fu. Beyond traditional microprocessors for geoscience high-performance computing applications. *Micro, IEEE*, 31(2):41 –49, March - April 2011.

[12] Intel Math Kernel Library, 2013. `http://software.intel.com/en-us/intel-mkl`.

[13] G. R. Mudalige, M. B. Giles, J. Thiyagalingam, I. Z. Reguly, C. Bertolli, P. H. J. Kelly, and A. E. Trefethen. Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems. *Parallel Computing*, 39(11):669–692, 2013.

[14] What is GPU Computing, 2013. `http://www.nvidia.com/object/what-is-gpu-computing.html`.

[15] OP2 github repository, 2013. `https://github.com/OP2/OP2-Common`.

[16] S. Pennycook, C. Hughes, M. Smelyanskiy, and S. Jarvis. Exploring SIMD for Molecular Dynamics, Using Intel Xeon Processors and Intel Xeon Phi Coprocessors. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 1085–1097, 2013.

[17] E. L. Poole and J. M. Ortega. Multicolor ICCG Methods for Vector Computers. *SIAM J. Numer. Anal.*, 24(6):pp. 1394–1418, 1987.

[18] Scotch and PT-Scotch, 2013. `http://www.labri.fr/perso/pelegrin/scotch/`.

[19] J. Rosinski. Porting, validating, and optimizing NOAA weather models NIM and FIM to Intel Xeon Phi. Technical report, NOAA, 2013.

[20] K. Skaugen. Petascale to Exascale: Extending Intel's HPC Commitment, June 2011. ISC 2010 keynote. `http://download.intel.com/pressroom/archive/reference/ISC_2010_Skaugen_keynote.pdf`.

[21] M. Smelyanskiy, J. Sewall, D. Kalamkar, N. Satish, P. Dubey, N. Astafiev, I. Burylov, A. Nikolaev, S. Maidanov, S. Li, S. Kulkarni, and C. Finan. Analysis and Optimization of Financial Analytics Benchmark on Modern Multi- and Many-core IA-Based Architectures. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1154–1162, 2012.

[22] Top500 Systems, 2013. `http://www.top500.org/list/`.

[23] A. Vladimirov and V. Karpusenko. Test-driving Intel Xeon Phi coprocessors with a basic N-body simulation. Technical report, Colfax International, 2013.