An Evaluation of Emerging Many-Core Parallel Programming Models

Matt Martineau
HPC Group, University of Bristol
m.martineau@bristol.ac.uk

Simon McIntosh-Smith
HPC Group, University of Bristol
cssnmis@bristol.ac.uk

Mike Boulton
HPC Group, University of Bristol
michael.boulton@bristol.ac.uk

Wayne Gaudin
Atomic Weapons Establishment
wayne.gaudin@awe.co.uk

Abstract

In this work we directly evaluate several emerging parallel programming models: Kokkos, RAJA, OpenACC, and OpenMP 4.0, against the mature CUDA and OpenCL APIs. Each model has been used to port TeaLeaf, a miniature proxy application, or mini-app, that solves the heat conduction equation, and belongs to the Mantevo suite of applications. We find that the best performance is achieved with device-tuned implementations but that, in many cases, the performance portable models are able to solve the same problems to within a 5-20% performance penalty. The models expose varying levels of complexity to the developer, and they all present reasonable performance. We believe that complexity will become *the* major influencer in the long-term adoption of such models.

Categories and Subject Descriptors D.1.3 [*Software*]: Parallel Programming

1. Introduction

HPC is undergoing significant growth as science and engineering are increasingly reliant upon large-scale simulations to support their cutting edge progress. In order to take advantage of supercomputing platforms, scientific codes need to be carefully re-engineered to exploit concurrency. Even after an application has been parallelised, portability can be a major issue, with many parallel programming models tying you to a particular device or platform. Scientific applications are often long-lived and monolithic, meaning that they cannot be easily rewritten to take advantage of modern supercomputing resources, greatly inhibiting their potential [6].

Energy efficiency has become a limiting factor in designing new HPC technologies, leading to a shift towards many-core devices and heterogeneous computing [9]. Many of the world's fastest supercomputers include a mix of CPUs, GPUs and accelerators, and this massive increase in node-level parallelism means that applications are becoming harder to develop for current architecture, and even harder to future-proof. These factors have created a demand for programming models that will enable scientific applications to take advantage of heterogeneous HPC resources, without having to maintain versions for each device. Importantly, while there are libraries and standards that enable some level of functional portability, they do not necessarily guarantee performance portability [7].

Given the prohibitive cost of rewriting scientific applications and the current rapid rate of change, it is imperative that application developers are well informed when they consider a modern parallel programming model, in order to safeguard their HPC investments.

1.1 TeaLeaf – A Heat Conduction Mini-App

TeaLeaf is an open source project that belongs to both the UK Mini App Consortium (UKMAC) [25] and Mantevo project [6]. The UKMAC represents a consolidated national effort to understand modern technologies and algorithms, that is fed into by Warwick University, Oxford University and the University of Bristol, supported and funded by the Atomic Weapons Establishment (AWE). The Mantevo project, run by Sandia National Laboratories, is an award winning collection of open source applications geared towards analysing high performance computing applications.

Mini-apps support the investigation of optimisation, scalability and performance portability, free from the limitations imposed by attempting such analyses with fully functional scientific applications. Importantly, this supports research into techniques for optimising such codes that can eventually be transferred into real scientific applications. TeaLeaf, in particular, has just enough functionality to be representative of the performance profile and computational complexity exposed by a production code, whilst maintaining a small codebase that is amenable to experimentation. The program is characterised by two of the seven dwarfs of High Performance Computing [1], Structured Grid and Sparse Linear Algebra.

The two-dimensional TeaLeaf implementation contains three iterative sparse matrix solvers: Conjugate Gradient (CG), Chebyshev, and Chebyshev Polynomially Preconditioned CG (PPCG) [2], each of which uses a 5 point stencil to solve the heat diffusion equation using face centred diffusion coefficients based on cell average densities. To ensure numerical stability of a parabolic partial differential equation with a tractable timestep, an implicit method is employed. The explicit solution, though simple to implement is constrained by a timestep that scales as $1/dx^2$ [25].

PPoPP '16, March 12-16 2016, Barcelona, Spain.

ACM 978-1-4503-4196-7/16/03.
http://dx.doi.org/10.1145/2883404.2883420

```
// Setup device data environment
#pragma omp target data \
    map(to: r[:r_len]) map(tofrom: p[p_len])
{
    // Offload calculation using resident data
    #pragma omp target if(OFFLOAD) device(DEVICE_ID)
    #pragma omp parallel for
    for(int jj = 0; jj < y; ++jj)
    {
        for(int kk = 0; kk < x; ++kk)
        {
            const int index = jj * x + kk;
            p[index] = beta * p[index] + r[index];
        }
    }
} // Only p is read back from device
```

Figure 1. Example of TeaLeaf function written using OpenMP 4.0.

2. Programming Models Background

Each of the programming models presented in this work require a different development approach, and expose varying levels of complexity and functional portability. Some details of the background, abstract approach and syntax relevant to TeaLeaf are presented.

2.1 OpenMP 4.0

OpenMP is a directive-based programming model that is widely adopted for parallel programming targetting CPUs in shared memory environments. The new standard, OpenMP 4.0 [18], introduces a number of directives that are designed to allow portability to accelerators through offloading [14]. The execution model takes directions from the host to offload computationally expensive operations to an accelerator device [26].

It must be noted that there is currently only limited compiler support for the offloading, which means that, at the time of writing, the TeaLeaf OpenMP 4.0 version can only be tested on Intel Xeon Phi Knights Corner (KNC) devices.

The syntax of relevant and new OpenMP 4.0 statements is presented in Figure 1 and discussed below:

- **omp target map(direction: array):** This region surrounds a section of code that will be offloaded, while potentially mapping some data onto the device.
- **omp target data:** Maps data onto a device for the duration of the scope, allowing multiple target regions to utilise data maintained on the device, avoiding unnecessary data transfers.
- **omp update to(variable or array):** Makes a variable or array in both memory spaces consistent by copying *to* or *from* the device.
- **omp simd:** Although not related to offloading, the **simd** directive is an important new feature that attempts to force loops to vectorise that wouldn't usually auto-vectorise, limiting the changes required to enable vectorisation.

OpenMP 4.0 is the principal open standard using a directive based approach, and offers a highly usable interface for parallel performance on heterogeneous devices. At the time of writing compiler support is limited, but several of the main compiler vendors are planning to introduce GPU-targetting functionality in the near future.

```
// Setup data scope on the device
#pragma acc data copyin(r[:r_len]) copy(p[:p_len])
{
    // Loops to be offloaded to the device
    #pragma acc kernels loop independent \
        collapse(2) present(p[:p_len], r[:r_len])
    for(int jj = 0; jj < y; ++jj)
    {
        for(int kk = 0; kk < x; ++kk)
        {
            const int index = jj * x + kk;
            p[index] = beta * p[index] + r[index];
        }
    }
} // Only p is read back from device
```

Figure 2. Example of TeaLeaf function written with OpenACC.

2.2 OpenACC

OpenACC is another directive-based programming model that supports offloading to NVIDIA GPUs and, more recently, x86 CPUs when using the PGI 15.10 compiler suite. Developers can use a selection of directives to inform the compiler as to how optimal code can be generated with minimal changes to a parallelisable codebase.

The directives are very similar to those in OpenMP 4.0, and expose similar functionality, the syntax of which is shown in Figure 2 and discussed below:

- **acc data copy(a):** Copies **a** to and from the device at the beginning and end of the scope.
- **acc kernels present(a):** Denotes a region of code that is to be offloaded to the target device, where **a** has already been copied onto the device by an enclosing data scope.
- **acc loop independent:** Signifies that a particular loop has data-independent iterations that can be offloaded to a device for parallel execution without internal synchronisation.

In order to support finer control over parallelism, it is possible to suggest the way that the iteration space should be decomposed using the **gang**, **worker**, and **vector** directives.

2.3 The RAJA Portability Layer

The RAJA programming model is a brand new abstraction layer designed by Lawrence Livermore National Laboratories (LLNL) to improve performance portability of advanced simulation and computing (ASC) codes. The key technical paper by Hornung et al. [7] outlined two core goals: (1) to abstract away "non-portable compiler and platform-specific directives" and other implementation details, insulating application developers, and (2) to make it easier for application developers to tune data layout and memory access for optimal operation on diverse memory hierarchies.

They suggest that organising and controlling memory locality is an essential step in porting serial scientific applications to run on parallel architectures, a position consistent with others who have tackled the same problem [22, 27]. Decomposing the problem domains into smaller units allows threads to have improved utilisation of shared data caches, but can lead to non shared data and "domain management operations" saturating the available resources. In principal, RAJA makes it easier to perform chunking, allowing optimisation of instruction and data cache utilisation.

```
// One time initialisation of indirection
RAJA::RangeISet full_domain(0, x * y);

// Dispatch computation by policy
forall<policy>(full_domain, [&](int index) {
    p[index] = beta * p[index] + r[index];
});
```

Figure 3. Example of TeaLeaf function written using RAJA.

An example of the syntax is shown in Figure 3, and there are several abstractions that are foundational to the design of the model, which are discussed below:

- **Separate loop body from traversal:** This decoupling makes it possible to choose device-optimal access patterns for a function without altering the loop body.
- **Partition iteration space into work units (Segments):** Abstracting access patterns into **Segments**, which fetch data using different access strategies. When different access patterns are required for a single operation, dividing memory access patterns into similar types allows the strategies to be handled separately, potentially in parallel.
- **Segment dispatch and execution (Indexsets):** RAJA supports combining the previous features, allowing **Segment**s to be aggregated based on type and dispatched for execution using a loop template. **Indexset**s represent a policy, e.g. "Dispatch segments in parallel and launch each segment on either a CPU or GPU as appropriate". The developer can recouple the core logic to a particular **Indexset** by choosing one of the built-in dispatch functions and passing a lambda statement (C++11 anonymous function declaration) containing the loop body.

Internally, the built-in dispatch functions wrap up platform-specific implementations, for instance a CPU-targetting implementation can contain OpenMP code, and a GPU-targetting implementation can use CUDA. Unfortunately, CUDA 7.0 does not currently support offloading lambda statements from host to device, which has slowed RAJA's GPU development. More recently CUDA 7.5 has added experimental support for lambda-based kernels that can be defined in host code, and the RAJA developers are in the process of writing an NVIDIA GPU targetting implementation based on this functionality.

2.4 Kokkos

The Kokkos framework is part of the Trilinos project, developed by Sandia National laboratories to provide a modern abstract approach to developing applications that require performance portability. The project emphasises the development of "robust algorithms for scientific and engineering applications on parallel computers". Edwards et al. [4], acknowledge two principal techniques that embody the philosophy of the model: (1) utilising abstraction to perform computation on many-core devices, and (2) leveraging the power of C++ templates to provide portable high performance data layout tuning functionality.

The programming model provides a range of generic abstractions that allow the user to create new codes or port existing applications. An example of the syntax is shown in Figure 4 and some of the abstract model and implementation details are discussed below:

- **Execution and Memory Spaces**: The library makes a distinction between execution space and memory space to support GPUs and accelerators, which need to have distinct memory from the host CPU. The library handles much of the interaction between these spaces but the developer can move data to and from the spaces using built-in copy methods.

```
// Kokkos functor-style kernel
template <class Device>
struct cg_calc_p
{
    typedef Device device_type;
    typedef Kokkos::View<double*, Device> k_view;

    cg_calc_p(double beta, KView p, KView r)
        : beta(beta), p(p), r(r) {}

    KOKKOS_INLINE_FUNCTION
    void operator()(const int index) const
    {
        p[index] = beta * p[index] + r[index];
    }

    double beta;
    k_view p;
    k_view r;
};

// Invoke the kernel from the host
cg_calc_p<DEVICE> cg_calc_p_kernel(beta, p, r);
Kokkos::parallel_for(n, cg_calc_p_kernel);
```

Figure 4. Example of TeaLeaf function written using the Kokkos framework.

- **Data Structures:** Kokkos uses **View**s, which are abstract data types that support mixing dynamic and compile-time dimensions for optimisation, as well as copy semantics analogous with the C++ std::shared_ptr, avoiding complex ownership constraints.
- **Functors:** The library utilises C++ class constructs called functors, where the function operator is overloaded and encapsulates the core functional logic. This pattern requires that **View**s are declared as local variables inside the class, and supports customisable reductions of complex types.
- **Lambda Support:** Lambda constructs can be used instead of functors to greatly reduce the amount of code required to write each kernel, but CUDA 7.0 does not currently support the feature, limiting its functional portability.
- **Parallel Execution:** Two key data parallel execution operations are provided: **parallel_for** which encapsulates iterative execution and **parallel_reduce** which further allows reduction of data using some function (defaults to zero-initialised sum).

Underpinning the abstract semantics is an implementation that uses C++ template meta-programming to rewrite the functors into device-specific code. Kokkos is currently able to output pthreads, OpenMP, and CUDA, which supports a good level of functional portability.

2.5 OpenCL

OpenCL is an open standard for writing applications that can execute on heterogeneous many-core devices, that was released in 2008 [17]. The standard was created to offer a low level but portable abstract model that can support both data and task parallel programming approaches. Given the number of vendor implementations of

```
// Create the kernel and setup the kernel args
cg_calc_p = clCreateKernel(program, "cg_calc_p", NULL);
clSetKernelArg(cg_calc_p, 0, sizeof(cl_int), &x);
...

// Enqueue the kernel from host code
clEnqueueNDRangeKernel(queue, cg_calc_p, work_dim,
        NULL, global_size, local_size, 0, NULL, NULL);

// OpenCL kernel to calculate new value of P
__kernel void cg_calc_p(int x, int y, double beta,
        __global double* p, __global double* r)
{
    int index = get_global_id(0);

    if(index < x * y)
    {
        p[index] = beta * p[index] + r[index];
    }
}
```

Figure 5. Example of TeaLeaf function written using OpenCL.

OpenCL, it is by far the most functionally portable model evaluated by our research.

Some key syntax is presented in Figure 5, and OpenCL's three primary abstract models are discussed below:

- **Platform model:** The model represents a host that interacts with some number of devices, with each device containing compute units, which in turn contain processing elements [8].
- **Execution model:** Code that is executed on the host and devices have isolated execution spaces, where kernels represent work that can be completed by a device. Kernels operate within a particular **context** established by the host code, which connects devices, kernels, program source and variables. Device-specific *command queues* can be created that allow work to be queued into the device by the host. The queues accept groups of *work items* called *work groups*, which are executed on a compute unit, with each *work item* being handled by an individual processing element [15].
- **Memory model:** The memory model separates the distinct memory regions and objects available to the host and devices that share a context. Similarly to CUDA this includes a distinction between host and device memory, as well as some hierarchy of memory on the devices that relates to the way data is mapped in an implementation-specific manner [23].

Because of this hierarchy of abstraction, OpenCL requires boilerplate code that is not necessarily required by other models. This includes setting up platforms, contexts, command queues, all kernel arguments and managing data transit between host and device. As a consequence, OpenCL is able to support a generic and open standard that is flexible to modern architecture, and can benefit from wide adoption on a range of devices, whilst offering extensive scope for performance tuning.

2.6 Compute Unified Device Architecture (CUDA)

CUDA is a mature parallel computing platform developed by NVIDIA to allow application developers to offload computation to their own GPUs, and should enable the greatest possible performance on NVIDIA devices. The platform supports C-based kernels and avoids abstracting the GPU architecture, instead exposing it to

```
// CUDA kernel to calculate new value of P
__global__ void cg_calc_p(int x, int y, double beta,
        double* r, double* p)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;

    if(index < x * y)
    {
        p[index] = beta * p[index] + r[index];
    }
}

// Invoke kernel from host code
cg_calc_p<<<num_blocks, BLOCK_SIZE>>>(beta, r, p);
```

Figure 6. Example of TeaLeaf function written using CUDA.

enforce decomposition of problems for task, data and thread parallelism. Underpinning the CUDA framework is the Parallel Thread Execution (PTX) intermediate representation (IR), and CUDA can be compiled using the *nvcc* compiler for immediate execution or for just-in-time compilation by the CUDA runtime.

The CUDA syntax is presented in Figure 6 and the abstract model is discussed below:

- **Threading:** Local resources are shared amongst threads, with each processing element of a streaming multiprocessor performing the same operations on individual data elements.
- **Kernels and decomposition:** To inform the GPU of which instructions to run, the application developer can create kernels. Kernels can then be structured onto a grid, and the problem domain decomposed into sets of thread blocks that are isolated from each other, each running on an individual multiprocessor and only sharing *global* memory.
- **Memory spaces:** There are a number of memory spaces other than *global*, for instance *registers* are the lowest latency memory, accessible only by individual threads , and *local* memory is shared between threads in a particular group. The library provides memory copy operations for moving data to and from the device, and there is additional functionality to map data for direct memory access between device and host for improved data transfer rates.

CUDA represents a pioneering and mature platform for writing optimised computation to offload to NVIDIA GPUs, but limits the user to only ever run their code on those devices. It is important to recognise that any parallel programming model targetting NVIDIA GPUs will use the CUDA platform and PTX IR to actually offload computation to the device, meaning that CUDA applications can provide a lower bound for performance on supported devices.

3. Design, Development, and Findings

Although it is not possible to guarantee that each of the models is perfectly implemented, the ports were individually considered and optimised. Each model was categorised as either cross-platform (OpenCL, Kokkos, RAJA, OpenACC, and OpenMP 4.0) or platform-specific (CUDA, and OpenMP 3.0), and non-portable optimisations were strictly avoided for the cross-platform implementations. In particular, optimisations were chosen that represented the best performance portability across all supported devices, with some consideration for future portability. Importantly, TeaLeaf's core solver logic and parameters were kept consistent

Model	CPUs	NVIDIA GPUs	KNC
OpenMP 3.0	Yes		Native
OpenCL	Yes	Yes	Offload
CUDA		Yes	
OpenMP 4.0	Yes	Experimental	Offload
Kokkos	Yes	Yes	Native
RAJA	Yes		Native
OpenACC	Yes	Yes	

Table 1. Supported implementations for each model.

between ports to ensure that each of the programming models were objectively compared.

While OpenMP 3.0 can be compiled natively on the KNC architecture, it is not considered a performance portable programming model and so was used as a best case for performance on the CPU *and* KNC. Although the implementation of RAJA available to us for this research is unreleased and excludes GPU support, we hope to extend this research to include RAJA GPU results in the future. Further to this, it must be reiterated that OpenCL's support extends beyond that shown in Table 1, to AMD GPUs, FPGAs and a number of other platforms, making it the most functionally portable of the models.

Each of the models have an associated development cost, and expose varying levels of complexity. As TeaLeaf is a fairly regular and structured data-parallel application, it required the use of the core feature set of each model, but didn't necessarily represent some of the more complicated use cases that will be encountered with large scientific applications. An important trend observed is that *all* of the programming models focus on node-level parallelism and exclude support for inter-node communications, which is handled with MPI in TeaLeaf.

There are restrictions on the languages supported by some of the models, for instance Kokkos and RAJA require that the resulting application code is compiled as C++11. Although the original TeaLeaf application was an OpenMP Fortran 90 codebase, we developed a functionally identical OpenMP C implementation to serve as a starting point for all of the ports.

3.1 OpenMP 4.0

Our initial implementation added a **target** region to each of the performance critical functions. At the highest possible scope, above the main timestep loop, a **target data** region was introduced that kept all data resident on the device until convergence was achieved for the particular step.

As part of other investigations we have ported several mini-apps to use OpenMP 4.0, and each time have encountered a performance overhead dependent upon the number of **target** invocations performed during execution. In TeaLeaf's case it is possible to wrap the entire solve step in a single **target** region, achieving a nearly identical runtime to the optimal OpenMP Fortran 90 native implementation. Unfortunately, this pattern has two flaws that make it unsuitable in real codes: (1) MPI communication cannot be handled from within a **target** region, and (2) it is potentially non-portable to other devices, in particular for targetting GPUs that are not intended to perform complex control flow and data allocations.

While we recognise that there is an overhead inherent with the **target** offload regions, we have not been able to prove exactly what causes it. Our understanding of the **target** offloading model is that each region is handled synchronously, theoretically leading to stalling around each computation. The recently released OpenMP 4.5 specification [19] includes the **nowait** directive for **target** regions, ensuring that a stream of target invocations can be queued on the device for immediate back-to-back execution. We hypothesise that this functionality will have a significant influence on the **target** overheads.

Overall, the complexity involved in developing the OpenMP 4.0 port was low, but required more specialist knowledge than OpenMP 3.0, especially in managing the flow of data to and from the device. One small difficulty we encountered was that the current OpenMP 4.0 standard does not perform deep copies of members of Fortran datatypes inside the **map** directives. In order to overcome this problem, it was necessary to pass all variables individually to functions that map data onto the device. We believe that this small restriction could present a surprisingly significant overhead for a large application with pervasive use of custom datatypes.

Also, the **target data** regions are currently constrained to lexically structured scopes, which was not an issue for porting TeaLeaf but may not scale well to more complicated applications. This issue is addressed in the OpenMP 4.5 specification with the introduction of the unstructured **target enter data** and **target exit data** directives.

3.2 OpenACC

Our OpenACC implementation of TeaLeaf was completed *after* the OpenMP 4.0 port, and we found that the two approaches had many similarities. In fact, it was possible to use the OpenMP 4.0 codebase as a starting point, changing the directives but maintaining the same data transitions.

Our final design embellished each of the data-affecting loops with the **kernels** directive, affording the compiler as much flexibility as possible when generating the offloading code. To successfully compile all of the loops as accelerator kernels, it was further necessary to append **loop independent** to each of the directives, telling the compiler that the iterations can be executed in parallel. Finally, to achieve the best possible performance all of the loops were collapsed, ensuring enough work was available to the target device. The **collapse** statement certainly improves performance on the GPU, but might make performance worse on the CPU.

As with OpenMP 4.0, a **data** region was created at the highest possible scope, ensuring that data was kept on the device for an entire step of the solver, reducing the amount of data transfer between host and device. Once we had determined the best approach for parallelising an individual loop, the port took little time to implement, presenting similar complexity to the OpenMP 4.0 port. In future we want to take the PGI 15.10 compilers and test how the OpenACC models translates onto CPUs, and discover what level of performance portability is achievable.

3.3 Kokkos

To port TeaLeaf to use Kokkos, every data-affecting function was wrapped into a functor, which included a template declaration, constructor, overloaded function call operator, and set of local variables. Generally, the simple reductions in TeaLeaf could be handled by the default Kokkos implementation, which initialises the reductee to zero and aggregates the value from all sources. In the one TeaLeaf kernel where a multi-variable reduction was required, it was necessary to write custom initialisation and join functions which further extended the code size of the functor. Also, all communication between the host and device had to be handled with the Kokkos abstract copy functions, necessarily exposing some memory management complexity.

Because each functor in Kokkos flattens the iteration space and provides a single index parameter, it was necessary to reform each cell's spatial location to exclude the halo regions from the computation. Our original implementation ignored the halo cells using a conditional statement within the functor body, but it transpired that Intel MIC native compilation did not optimally handle the local

```
using namespace Kokkos;

// Call from host
parallel_for(TeamPolicy<DEVICE>(dims.x - 2*halo, kernel);

// Inside Functor
KOKKOS_INLINE_FUNCTION
void operator()(const int index) const
{
    int team_offset = (team.league_rank() + halo) * dims.y;

    parallel_for(TeamThreadRange(team, halo, dims.y-halo),
        [&] (int& j)
        {
            int index = team_offset + j;
            p[index] = beta * p[index] + r[index];
        });
}
```

Figure 7. Kokkos kernel implementing hierarchical parallelism for re-encoding loop-level halo exclusions.

conditions. Collaborators from Sandia National Laboratories proposed an alternative approach using hierarchical parallelism.

This solution was incorporated into each performance critical functor, introducing layers of parallelism throughout the code in the form of nested lambda functions. Figure 7 demonstrates two-dimensional hierarchical parallelism added to the functor in Figure 4, and if three-dimensional exclusions were needed, an additional nested lambda statement would be required. When performing a reduction, additional code is needed to critically add the results from each team. This additional control over the parallelism allows the halo exclusion to be encoded back into the iteration space, which is more abstract and well expressed than a loop-body conditional, but does significantly increase the complexity of each call.

We believe that a key requirement of models like Kokkos is that they reduce the barrier to entry for scientific application developers wanting to target heterogeneous platforms. They must limit the distance between the development effort required of a basic OpenMP F90/C application and a portable solution written with their APIs. Kokkos can now be written using lambda expressions instead of the templated functor syntax, making the code significantly more succinct, however, the lack of support in CUDA 7.0 meant this improvement could not be evaluated in our research. When using the lambda style, Kokkos presents a convenient and expressive style that abstracts platform-specific complexities, making it a powerful model for new applications using C++11.

3.4 RAJA

As RAJA is currently still in pre-release development, our implementation may not be representative of the style that will be required to target GPUs and accelerators. However, for targetting the CPU, the port required little knowledge beyond C++ lambda functions, and involved a similar development effort to OpenMP 3.0.

Porting to RAJA required changing all of the main loops to be lambda calls, and creating **IndexSet**s to handle the data traversal. Because RAJA wraps each function's iteration space into an indirection array, it was possible to exclude the halo boundaries without any explicit conditions or index calculations in the loop body. While this did make each of the lambda calls succinct, the precomputation of those indirection lists still had to be occur earlier in the application. Given lots of repetitive access patterns throughout an application this would likely lead to a reduction of code when compared to OpenMP, but for cases where data traversal is fairly diverse between functions, this may lead to a bloat of decoupled code that generates the indirection lists.

The TeaLeaf application does not require particularly complicated data access patterns, and so our evaluation cannot give much insight into the full power of this feature. We do however believe that, when introducing RAJA into large codebases, careful design will be required to outline exactly where the indirection array initialisation belongs.

We did find that it was necessary to create our own implementations of the dispatch functions, to handle situations where we had multiple reduction variables, and for multiple indexing. This flexibility was very useful, but could potentially inhibit long-term portability, as the custom implementations diverge away from the core RAJA implementation over time.

Our impression of RAJA is that, given it is not yet released, the philosophy is sound, and its use of C++11 features made porting the application very straightforward. Should RAJA maintain this usability once functional portability is improved, we expect it to represent a desirable approach to developing new parallel applications using C++11.

3.5 CUDA

CUDA is well discussed in other research [3, 11, 20], but overall we found that it exposed greater complexity than all of the ports except for OpenCL. In order to port TeaLeaf to CUDA we essentially converted all of the loops into CUDA kernels, and wrote data copying and reduction logic. While this was close in development effort to Kokkos, CUDA was more complex, primarily because it was necessary to create a custom GPU-specific reduction, including reduction code inside all of the individual reduction-based kernels.

Assuming a 1D grid of 1D blocks of threads, you also need to calculate a block size and corresponding number of blocks, as well as checking for iteration overspill from within the kernels. Importantly, CUDA offers no portability beyond NVIDIA GPUs, and offers several features that can potentially increase this complexity for potentially improved performance.

3.6 OpenCL

Our immediate impression of OpenCL is that it exposed more complexity than the other models, and also required more boilerplate code to handle the abstract model. However, this model allows the framework to support many different architectures, and offer the greatest functional portability of any of the models presented in this paper. It also means that there is a lot of scope for tuning for a particular device, should it be necessary. Once the boilerplate code is complete, the porting experience is not much more complicated than CUDA, requiring some additional abstract code when kernels and buffers are created.

One important complication with OpenCL is the reductions as, similar to CUDA, they have to be manually written but, contrary to CUDA, they potentially have to target multiple different devices. In our case, targetting the CPU, GPU and KNC, we would ideally create device-specific reductions for each of them that take advantage of the device characteristics, but this puts the responsibility on the developer and inhibits long term portability.

OpenCL 2.0 includes built-in workgroup reductions that can be implemented by particular vendors, and may offer an important improvement for performance portability. To reduce the complexity of OpenCL, there are C++ and Python wrappers, which allow some improvement in the host-code syntax.

4. Mesh Convergence Performance Analysis

Each of the ports were tested individually using the same problem parameters for the three solvers: CG, PPCG and Chebyshev. The

Device	Peak BW	STREAM BW
Xeon E5-2670 CPU x 2	102.4 GB/s	76.2 GB/s
NVIDIA K20X GPU	250.0 GB/s	180.1 GB/s
Xeon Phi 5110P KNC	320.0 GB/s	159.9 GB/s

Table 2. Devices and corresponding memory bandwidth (BW).

testing was performed on the Blue Crystal supercomputer at the University of Bristol, and the Swan XC40 supercomputer provided by Cray Inc., using the modern HPC devices listed in Table 2.

All of the results presented are for a mesh size of 4096x4096, which represents the point of mesh convergence for the problem, where a larger mesh size would provide no additional scientific information.

4.1 CPU

The CPU results were collected on dual socket Intel Xeon E5-2670 8-core Sandybridge processors, with 16 threads and thread affinity set to *compact*. All of the models except for CUDA support parallel execution on CPU architectures.

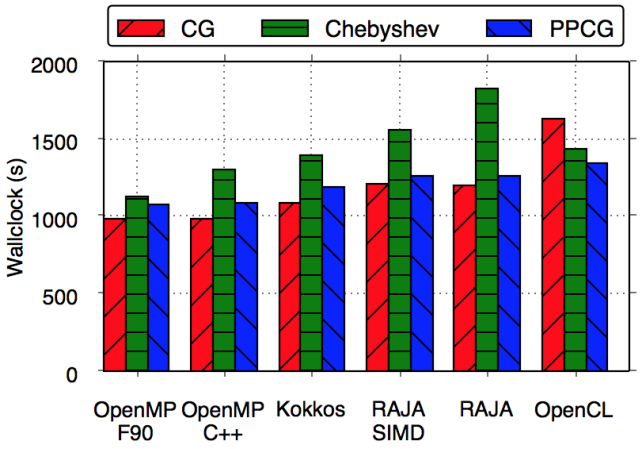

Figure 8. Results for dual socket Intel Xeon E5-2670 CPUs solving across a 4096x4096 mesh *(lower is better)*.

The pure OpenMP implementations are the fastest options, with the C++ implementation performing worst on the Chebyshev solver, experiencing 15% increased runtime compared with the Fortran 90 version. Our research found that this performance difference occurs for identical TeaLeaf code, depending on whether it was compiled as C or C++, with Intel compilers (15.0.3).

Kokkos demonstrates excellent performance across all of the solvers, with at most a 10% penalty compared to the C++ implementation. This is a strong indication of the model's potential, and ability to output well configured and optimised code.

The RAJA port exhibits a roughly 20% penalty for the CG and PPCG solvers, but the Chebyshev solver consistently requires an additional 40% solve time. We hypothesised that, as the use of indirection lists in RAJA precludes vectorisation, the performance could be indicative of the role that vectorisation plays in good performance for the Chebyshev solver. By creating proof of concept RAJA loop implementations that utilised the OpenMP 4.0 **simd** statement (*RAJA SIMD*), we were able to improve this performance by around 20% for the Chebyshev solver bringing it in line with the other solvers.

The OpenCL CPU implementation suffered from very high variance, with minimum runtime of 1631s and maximum of 2813s across 15 tests. While we do not know exactly why this variation was occuring, we have observed that the Intel OpenCL implementation uniquely doesn't use OpenMP to handle the CPU parallelism, instead using Intel Thread Building Blocks (TBB). Intel TBB operates a non-deterministic work-stealing scheduler [10], and we have considered that the variability may have been affected by this functionality. If developer directed thread affinity control were possible, we expect this variance could be limited. Lee et al. [12] take this point even further and suggest that affinity control would likely allow enhanced performance tuning of OpenCL in general.

Overall, the performance was quite consistent across the models on the CPU and, excepting some minor performance issues, at most a 20% performance penalty is likely to be observed by choosing any of the performance portable options.

4.2 GPU

The results were collected on an NVIDIA Tesla K20X using CUDA 7.0, hosted on Cray Inc.'s XC40 Swan supercomputer. It is important to re-iterate here that OpenCL is the only option that can also target AMD GPUs, which gives it an advantage over the other models in terms of functional portability. Also, Kokkos uses template meta-programming to re-write the application code into CUDA, and OpenCL and OpenACC directly output PTX code.

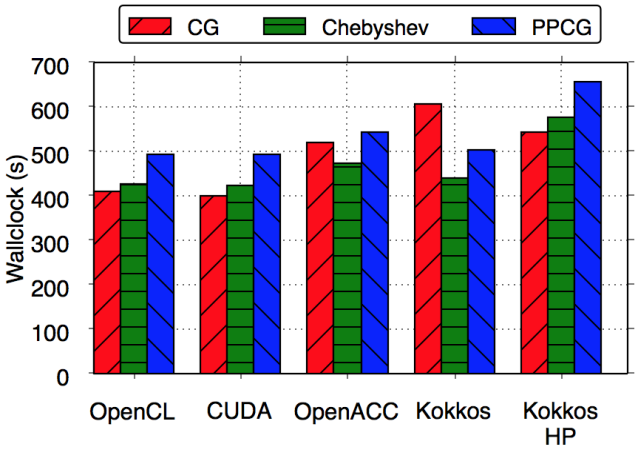

Figure 9. Results for GPU implementations on an NVIDIA K20X solving across a 4096x4096 mesh *(lower is better)*.

The performance results show that both CUDA and OpenCL perform almost identically, and achieve better results than the other models. This is a very good result that shows that OpenCL is able to perform exceptionally well on the GPU, matching the non-portable and device-optimised CUDA implementation.

OpenACC achieved acceptable results for all of the solvers, with a roughly 30% penalty for CG and 10% for the other two solvers, but it must be recognised that the port was the easiest to develop for the GPU.

The Kokkos implementation also exhibits very good performance for the Chebyshev and PPCG solver, suffering less than a 5% performance penalty compared to the CUDA implementation. Unfortunately, the CG solver demonstrates an unexplained performance problem, requiring roughly 50% additional solve time compared with OpenCL and CUDA. In order to test this issue we also attempted to test on an NVIDIA K20c with CUDA 6.5, but saw identical problems with the CG solver. Given the results for the other solvers, we expect that this is a performance issue that could be fixed or improved given further investigation.

Some collaboration with Sandia did find that a solution using hierarchical parallelism (*Kokkos HP*) was able to improve the performance by around 10% for the CG solver. Unfortunately, this was to the detriment of the PPCG and Chebyshev solver, which experienced a more than 20% overhead following the change. In spite of the problem with the CG solver, the results for Kokkos are impressive and demonstrate a good level of performance portability between Intel CPU and NVIDIA GPU architectures.

4.3 Intel Xeon Phi Knights Corner

The results were collected using 60 cores with 4 hardware threads (240 threads total) and thread affinity set to *compact*. Our overall impression is that the KNC architecture is challenging to achieve reasonable and consistent performance on, and we found that significant differences in performance profile can be seen between different versions of the device.

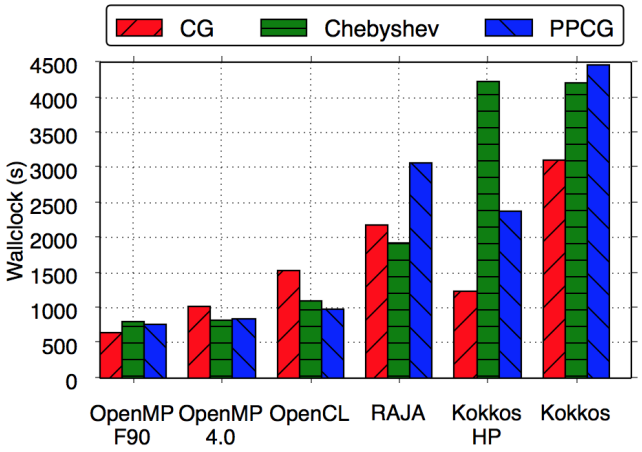

Figure 10. The results on a 61 core Intel Xeon Phi Knights Corner SE10P on a 4096x4096 mesh *(lower is better)*.

The performance results clearly show that the performance on the KNC device was far more varied between the programming models, which is indicative of the challenge it posed. The natively compiled OpenMP Fortran 90 implementation of TeaLeaf represents the best possible performance achievable for all solvers, maintaining fairly consistent runtimes between the three solvers.

The OpenMP 4.0 port required 45% additional runtime for the CG solver compared to the Fortran 90 implementation, but achieved performance to within 10% for both the Chebyshev and PPCG solvers. As previously discussed, we were able to improve upon our portable OpenMP 4.0 implementation by reducing the number of target regions, achieving identical performance to the Fortan OpenMP native port, but making the application non-portable.

Our OpenCL implementation suffered from unusual behaviour, achieving acceptable performance for the Chebyshev and PPCG solvers, but poor performance for the CG solver at nearly 3x worse performance than the best port. We did observe that running this identical code on a different version of KNC resulted in the CG solver runtime reducing to roughly 1100s, with the other solvers maintaining the same performance. This would appear to suggest that there is a performance problem that is being caused by an issue with the architecture or software, as opposed to improper implementation.

Although RAJA does not come with any automatic support for native compilation, it was straightforward to use the **mmic** switch to natively compile the RAJA port. The results above show that this did not lead to good performance compared to the Fortran 90 OpenMP implementation, with substantially higher runtimes required for all solvers. We know that vectorisation is very important for performance on the KNC and plan to test this with our proof-of-concept SIMD implementation in the future.

As previously discussed, the Kokkos hierarchical parallelism variant (*Kokkos HP*), was developed by Sandia National Laboratories to overcome a performance issue with halo exclusion conditions being present in the loop body. This solution originally came about because the conditions in the body of each loop are handled particularly inefficiently when being natively compiled. The hierarchical parallelism solution re-encodes this information so that no check is required, roughly halving the solve time for the CG and PPCG solvers on the KNC.

Overall, the results show that each model is able to achieve acceptable results for at least one solver with some tuning. We believe that this is enough to suggest that performance portability could be possible given more maturity and focus on the KNC architecture.

5. Even-Step Mesh Increment Analysis

The previous results focus upon the mesh convergence limit because it represents the point at which the programming models are subjected to the most intensive yet realistic data load. It is also interesting to observe the behaviour that occurs at lower problem sizes, as it reveals interesting details not seen at the convergence limit.

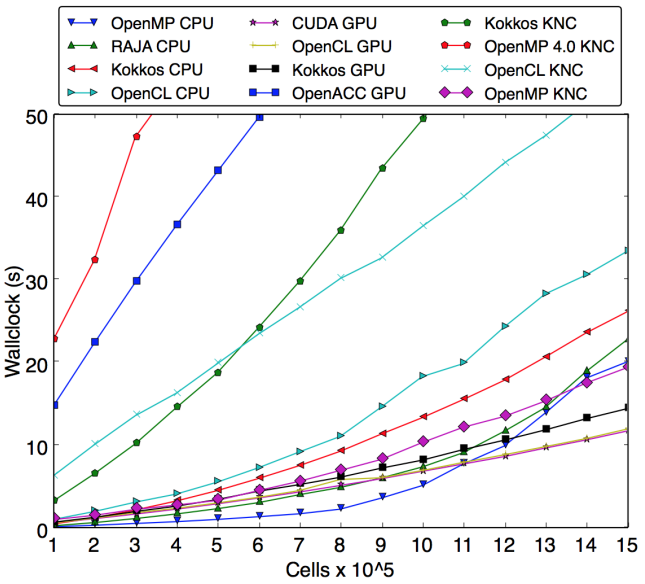

Figure 11. A plot of the runtime as problem size is increased in even steps for all models *(lower is better)*.

There are many distinct features of the plot in Figure 11, in particular all of the models present different runtime growth patterns. The problem sizes are shown up to 1225x1225, or 15×10^5 cells, an order of magnitude smaller than the number of cells at a mesh size of 4096x4096. Several of the programming models appear to have a very fast runtime growth rate, in particular OpenMP 4.0, OpenACC, Kokkos KNC and OpenCL KNC. This growth in runtime eventually slows and the models become more consistent with the others as the mesh convergence limit is reached. Each of those models has a high intercept on the plot and we expect that this runtime growth is indicative of large overheads in each of the models

that are hidden as the amount of computation and data processing is increased.

Another notable feature is that the OpenMP Fortran 90 implementation achieves the best performance up to 9×10^5 cells, but then the CPU models experience a gradual decrease in performance. This change point appears to indicate when the CPU caches have become saturated and data needs to be stored in DRAM, over time creating a memory latency and bandwidth bottleneck.

It can also be seen that the GPU-targetting implementations continue to benefit from linear runtime growth, which demonstrates that they are effectively utilising the device's data processing capabilities. For the KNC, most of the models suffer from the large overheads at these mesh sizes, but the OpenMP Fortran 90 implementation demonstrates fairly linear runtime growth similar to the GPU-targetting models.

6. Bandwidth Analysis

As TeaLeaf is a memory bandwidth bound application, observing the peak bandwidth achieved on each device presents an important measure of the success of the models at taking advantage of the targetted resources. We present the average bandwidth achieved across all solvers relative to the bandwidth achieved by the STREAM benchmark on each of the target devices.

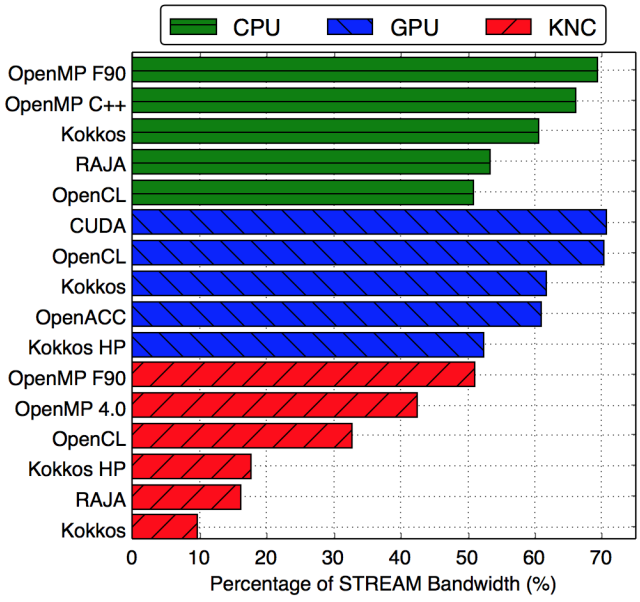

Figure 12. The percentage of STREAM bandwidth achieved by each model averaged over all solvers *(higher is better)*.

The results unequivocally show that the device-optimised implementations, OpenMP 3.0 and CUDA, achieve the best overall memory bandwidth utilisation. Aside from this, we see that most of the performance portable options fall within a 20% bandwidth reduction from this point, with several of the CPU and GPU alternatives experiencing at most a 10% memory bandwidth penalty.

The Kokkos implementation performs to within 10% of the best achieved memory bandwidth for both the CPU and GPU, which is a very impressive result and clearly advocates the potential of the model. The results on the KNC are poor but the improvement seen with the hierarchical parallelism update show that better performance may be possible given some device-specific tuning or implementation maturity.

The hierarchical parallelism implementation of Kokkos improved performance on KNC and maintained CPU performance, but the performance reduction of Chebyshev and PPCG solvers on the GPU mean there are some trade-offs. It would be possible to achieve better average performance by combining both solutions with some conditionality regarding the target device and solver. However, this starts to put the performance portability responsibility back in the hands of the application developer, and makes the resulting code much more complicated than maintaining a single solution throughout.

7. Related Work

Herdman et al. [5] performed a similar analysis to the one presented in this research, evaluating OpenACC, OpenCL and CUDA using a sister mini-app of TeaLeaf's, CloverLeaf.

Rupp et al. [21] perform an extensive inter- and intra-vendor performance portability investigation of OpenCL using miniature linear algebra kernels. Concentrating on structured grid codes, McIntosh-Smith et al. [16] used three benchmarks, including the mini-app CloverLeaf, to investigate the performance portability of OpenCL across a number of devices. Similarly, Mallinson et al [15] compared the performance of the OpenCL implementation of CloverLeaf relative to the performance attainable using device-optimised ports. Also concentrating on low-level frameworks: CUDA, OpenCL, and AMD Intermediate Language, Du et al. [20] investigated inter- and intra-vendor performance portability on GPU devices.

On the topic of directive-based programming models targetting accelerators, Lee et al [13] evaluated a range of models with 13 distinct benchmarks. In many cases the performance matched their hand-tuned CUDA implementations, but certain optimisations were difficult or even impossible to express with the high-level directive models.

Using a pattern-based comparison, Wienke et al. [26] compared both OpenMP 4.0 and OpenACC, finding that OpenACC exposes more features overall, but that OpenMP 4.0 will likely achieve better long-term adoption because of its success on the CPU. Teodoro et al [24] evaluate the performance profiles of a KNC, GPU, and CPU with respect to Microscopy Image Analysis. They concluded that the devices had a significant variance between particular operations, exposing some preference for particular operations.

8. Future Work

TeaLeaf has a specific performance profile, and it would be very useful to consider the success of each model relative to applications that have different requirements such as CloverLeaf and the SN Application Proxy (SNAP). As described throughout, many of the programming models evaluated are awaiting compiler support for particular platforms. Performance portability could be assessed on additional target hardware not investigated in this paper, or where there are novel architectural differences such as the Intel Xeon Phi Knights Landing with its high bandwidth memory. Finally, we believe that it would be useful to investigate each model's ability to handle more complicated requirements such as heterogeneous compute or adjusting data layouts per device.

9. Conclusions

We have used the mini-app TeaLeaf to test a range of parallel programming models, exposing their functional portability and performance portability on three distinct modern HPC devices. Our research has shown that among the increasing selection of parallel programming models, there is a varied range of techniques being used to exploit increasing node-level parallelism, each presenting individual levels of complexity and support for scientific application development.

Given that the performance has been reasonable for all of the programming models, we expect that the future of such models will depend on their ability to improve the ease with which applications are developed. Beyond the functional portability offered, the level of complexity that a model exposes is likely to become the deciding factor as to whether a model is useful to a particular application developer.

Kokkos and RAJA have been shown to be promising options for performance portability that are growing in usefulness as they mature. However, those models will still require up-front investment to migrate existing C and especially Fortran codes, and may need to expose additional complexity to achieve good performance across multiple devices. Although a port written with Kokkos today would likely have to use functors, which are quite verbose, the more mature lambda implementations possible with both Kokkos and RAJA are definitely competitive with those directive based models for ease of development.

In spite of the variability issues on the CPU, OpenCL is one of the most performance portable options. Its extensive support on a range of devices not targetted by the other models, also sets OpenCL apart. The directive based programming models represent the best case in terms of development ease, and the fact that they are language-agnostic between C and Fortran is also a significant benefit. If implementations of OpenMP 4.0 become available that can match OpenACC's performance, we predict it will be well adopted by scientific application developers as a usable interface for future-proof performance.

Acknowledgments

This work was funded by an EPSRC CASE studentship supported by the UK Atomic Weapons Establishment. The authors would like to extend our gratitude to David Beckingsale at Lawrence Livermore National Laboratory for his support with the RAJA port, and Christian Trott from Sandia National Laboratory for his support with the Kokkos port. We would also like to thank the University of Bristol Intel Parallel Computing Center (IPCC), High Performance Computing Group, and Advanced Computing Research Centre. Intel for the provision of an Intel Xeon Phi and Cray Inc., for allowing us to test on their Swan XC40 supercomputer.

References

[1] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] M. Boulton and S. McIntosh-Smith. Optimising sparse iterative solvers for many-core computer architectures, 2014. URL `http://www.many-core.group.cam.ac.uk/ukmac2014`. UK Many-Core Developer Conference (UKMAC).

[3] S. Che, M. Boyer, et al. A Performance Study of General-Purpose Applications on Graphics Processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370 – 1380, 2008.

[4] H. C. Edwards, C. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12): 3202–3216, 2014.

[5] J. Herdman, W. Gaudin, S. McIntosh-Smith, M. Boulton, D. Beckingsale, A. Mallinson, and S. Jarvis. Accelerating Hydrocodes with OpenACC, OpenCL and CUDA. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 465–471. IEEE, 2012.

[6] M. Heroux, D. Doerfler, et al. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[7] R. Hornung, J. Keasler, et al. The RAJA Portability Layer: Overview and Status. Technical Report LLNL-TR-661403, Lawrence Livermore National Laboratory, 2014.

[8] Khronos OpenCL Working Group. The OpenCL Specification Version 1.2, 2015.

[9] P. Kogge and J. Shalf. Exascale Computing Trends: Adjusting to the "New Normal" for Computer Architecture. *Computing in Science & Engineering*, 15(6):16–26, 2013.

[10] A. Kukanov, V. Polin, and M. Voss. Flow Graphs, Speculative Locks, and Task Arenas in Intel® Threading Building Blocks.

[11] D. Lee, I. Dinov, B. Dong, et al. CUDA optimization strategies for compute-and memory-bound neuroimaging algorithms. *Computer methods and programs in biomedicine*, 106(3):175–187, 2012.

[12] J. H. Lee, K. Patel, N. Nigania, H. Kim, and H. Kim. Opencl performance evaluation on modern multi core cpus. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1177–1185. IEEE, 2013.

[13] S. Lee and J. Vetter. Early evaluation of directive-based gpu programming models for productive exascale computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 23. IEEE Computer Society Press, 2012.

[14] C. Liao, Y. Yan, B. de Supinski, D. Quinlan, and B. Chapman. Early Experiences with the Owill primarily be targettable aopenMP Accelerator Model. In *OopenMP in the Era of Low Power Devices and Accelerators*, pages 84–98. Springer, 2013.

[15] A. Mallinson, D. Beckingsale, W. Gaudin, J. Herdman, and S. Jarvis. Towards Portable Performance for Explicit Hydrodynamics Codes. 2013.

[16] S. McIntosh-Smith, M. Boulton, D. Curran, and J. Price. On the Performance Portability of Structured Grid Codes on Many-Core Computer Architectures. In *Supercomputing*, volume 8488 of *Lecture Notes in Computer Science*, pages 53–75. Springer International Publishing, 2014.

[17] A. Munshi, B. Gaster, T. Mattson, and D. Ginsburg. *OpenCL Programming Guide*. Pearson Education, 2011.

[18] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.0, 2013.

[19] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.5, 2015.

[20] D. Peng, W. Rick, L. Piotr, et al. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391 – 407, 2012.

[21] K. Rupp, P. Tillet, F. Rudolf, et al. Performance Portability Study of Linear Algebra Kernels in OpenCL. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, IWOCL '14, pages 8:1–8:11, New York, NY, USA, 2014. ACM.

[22] A. Sidelnik, S. Maleki, et al. Performance portability with the Chapel language. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 582–594. IEEE, 2012.

[23] J. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[24] G. Teodoro, T. Kurc, J. Kong, L. Cooper, and J. Saltz. Comparative Performance Analysis of Intel (R) Xeon Phi (TM), GPU, and CPU: A Case Study from Microscopy Image Analysis. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1063–1072. IEEE, 2014.

[25] UKMAC. UK Mini-App Consortium: TeaLeaf. http://ukmac.github.io/TeaLeaf, 2015.

[26] S. Wienke, C. Terboven, J. C. Beyer, and M. Müller. A Pattern-Based Comparison of OpenACC and OpenMP for Accelerator Computing. In *Euro-Par 2014 Parallel Processing*, pages 812–823. Springer, 2014.

[27] Y. Zhang, M. Sinclair II, and A. Chien. Improving performance portability in OpenCL programs. In *Supercomputing*, pages 136–150. Springer, 2013.