Performance Portability Challenges for Fortran Applications

Abigail Hsu*[§], David Neill Asanza[†][§], Joseph A. Schoonover[‡], Zach Jibben[§], Neil N. Carlson[§], Robert Robey[§]
*Stonybrook University, Email: abigail.hsu@stonybrook.edu
[†]Grinnell College, Email: neillasa@grinnell.edu
[‡]Fluid Numerics, LLC, Email: joe@fluidnumerics.com
[§]Los Alamos National Laboratory
https://www.overleaf.com/project/5b6a028c86009b3c10c53a9e

***Abstract*—This project investigates how different approaches to parallel optimization impact the performance portability for Fortran codes. In addition, we explore the productivity challenges due to the software tool-chain limitations unique to Fortran. For this study, we build upon the Truchas software, a metal casting manufacturing simulation code based on unstructured mesh methods and our initial efforts for accelerating two key routines, the gradient and mimetic finite difference calculations. The acceleration methods include OpenMP, for CPU multi-threading and GPU offloading, and CUDA for GPU offloading. Through this study, we find that the best optimization approach is dependent on the priorities of performance versus effort and the architectures that are targeted. CUDA is the most attractive where performance is the main priority, whereas the OpenMP on CPU and GPU approaches are preferable when emphasizing productivity. Furthermore, OpenMP for the CPU is the most portable across architectures. OpenMP for CPU multi-threading yields 3%-5% of achievable performance, whereas the GPU offloading generally results in roughly 74%-90% of achievable performance. However, GPU offloading with OpenMP 4.5 results in roughly 5% peak performance for the mimetic finite difference algorithm, suggesting further serial code optimization to tune this kernel. In general, these results imply low performance portability, below 10% as estimated by the Pennycook metric. Though these specific results are particular to this application, we argue that this is typical of many current scientific HPC applications and highlights the hurdles we will need to overcome on the path to exascale.**

***Index Terms*—Fortran, OpenMP, offload, CUDA, GPU, performance portability, productivity**

I. Introduction

Fortran codes still see significant use and active development at national laboratories, universities, and other research institutions. These applications present different challenges for performance portability and productivity than other High Performance Computing (HPC) languages such as C and C++. These issues are primarily in the maturity of Fortran standards implementations in compilers and in the support of parallel programming APIs, such as OpenMP and CUDA. With Graphics Processing Units (GPUs) becoming a focus during the transition to exascale, Fortran developers are tasked with writing robust code that can make effective use of both multi-core CPU and GPU hardware. Coupling this challenge with a desire to produce manageable code, an ideal solution is to transition currently existing code so that it can target these architectures with minimal refactoring and code modification.

Directive-based APIs, like OpenACC [1] and OpenMP [2], are possible solutions to achieving this ideal source code. In these APIs hints to the compiler are provided through special comments. By enabling a compiler flag, these hints can be interpreted and the compiler attempts to parallelize an encapsulated region of code. For well optimized serial codes, this approach can provide significant performance gains, with little to no changes required in the original code. However, with GPU offloading becoming a major focus, questions arise about the proficiency of the automatic code generation for these architectures.

Kernel-based APIs, like OpenCL, CUDA, and HIP offer the ability to offload to GPUs only; with the exception of OpenCL, which can target a variety of hardware. Kernel-based APIs require that developers add new code (compute kernels) that express per-thread instructions. Additionally, when working with GPUs, additional code is needed to manage the distinct memory spaces associated with the CPU and GPU. Relative, to directive-based approaches, kernel-based approaches introduce more code to manage but come with explicit control over the utilization of hardware. Tuning of compute kernels, memory movement, and memory layout on the GPU is under the explicit control of the developer, making it easier to obtain desired performance goals.

There are an increasing number of options to performance portability for Fortran scientific codes. Shown in Fig. 1 is an overview of the possible approaches for Fortran performance portability (the degree of support is indicated by thicker lines and dashed lines indicate partial support). Currently, OpenACC and OpenMP offer both CPU and GPU offloading for Fortran applications in select compilers. A mature implementation of OpenACC exists in the PGI compilers. GCC 8.0 offers OpenACC (2.0a standard) but it is suggested that the performance falls short of the PGI implementation [3]. Most compilers support multi-core CPU implementations of OpenMP, but, as we'll show in this work, the performance varies with the choice of compiler and hardware. GPU offloading with OpenMP is still in its early stages, with IBM's `xlf` compiler having the most mature implementation. CUDA, OpenCL, and HIP are languages intended for C/C++, and are compatible with most C/C++ compilers. PGI does support CUDA-Fortran, which offers the ability to write CUDA-Kernels using Fortran's array syntax, but this requires PGI compilers for GPU acceleration. An implementation of FortranCL exists, though this project has seen little to no activity for the last six years and is in need of revitalization.

Overall, there are a number of options available to achieve performance portability that allow developers to keep the attractive features of Fortran such as multi-dimensional arrays, array syntax, and bounds checking that make it such a popular language for scientific programming. New APIs, like OpenMP with GPU offloading and OpenACC are being adopted by compiler writers. Because of this, it is relevant to investigate the effort required to accelerate a Fortran application with these APIs and estimate the obtained performance and performance portability.

In this paper, we describe the efforts required and the obtained performance improvements on two kernels of the Truchas code using OpenMP for multi-core CPU and GPU offloading and CUDA (Compute Unified Device Architecture). This exploration highlights key differences in directive-based and kernel-based acceleration approaches. Section II-A gives a brief background on OpenACC and OpenMP, two directive based options for multi-core CPU and GPU offloading. Section II-B provides an overview of CUDA, OpenCL, and HIP, three APIs for accelerating applications with GPUs through expression of Same-Instruction-Multiple-Thread (SIMT) kernels. Section III-B highlights challenges in maintaining portable Fortran that also effectively leverages available compute hardware.

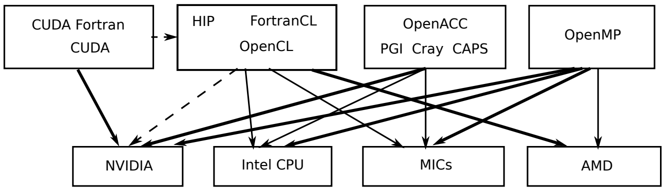

Fig. 1. There are several options for generating applications for various architectures from a single Fortran source code(The degree of support is indicated by thicker lines and dashed lines indicate partial support)

II. Background

A. Directive-Based Approaches

OpenACC [1] and OpenMP [2] are two dominating directive-based approaches for parallelizing applications for multi-core CPUs and GPUs. The vision of these APIs, expressed in the standards (currently 2.6 for OpenACC and 4.5 for OpenMP), is that developers can insert comments in their source code to instruct the compiler to generate instructions for multi-core or accelerator execution. In this view, developer's don't necessarily need to modify their underlying serial code and can issue special compiler flags to target the different architectures. Although this is the aim of the OpenMP and OpenACC standards, competition between vendors and the rapid pace of changing hardware has resulted in portability issues for scientific software developers.

Currently, the OpenACC 2.6 standard is fully supported within the PGI compiler [4]. Cray compilers support OpenACC 2.0 [5] as of Fortran compiler version 8.7. GCC 8.0 supports the OpenACC 2.0a standard and a feature branch of GCC partially supports OpenACC 2.5 [6]. Recent assessments of the performance of the OpenACC implementations suggest that the PGI implementation is more mature and obtains better performance for the same directives [3], although some additional effort can be made with the GCC implementation to push the performance closer to PGI's. All implementations of the OpenACC standard support offloading to Nvidia GPUs only, excluding the use of AMD accelerator hardware.

OpenMP traditionally has been used for multi-core CPU programming using a loop-level work sharing concept. Kapinos and Mey [7] looked at the task-based concepts introduced in OpenMP 3.0 in 2010 with a Fortran 90/95 code and find that compiler implementations could be buggy or had performance issues and thread safety tools were still not available. While the compiler support has probably improved for tasking since then, their experience with cutting-edge features is all too familiar. Now with the introduction of GPU offloading using OpenMP 4.0+, successes have been noted with the Cray compilers and there are efforts to support this feature in future releases of the GCC compiler suite and IBM's xlf. All signs are that there will be some healthy compiler implementations available.

The Heterogeneous Compute Compilers (HCC) for C++ [8], are currently in development under AMD's ROCm platform and will offer GPU offloading support for AMD and Nvidia GPUs. However, HCC is a C/C++ compiler, which leaves Fortran developers excluded from reaching AMD hardware through currently offered directive-based APIs.

B. Kernel-Based Approaches

Kernel-based APIs include CUDA and CUDA Fortran, OpenCL and FortranCL, and HIP. When accelerating an application with a kernel based approach, the developer is required to manage the distinct memory spaces of the CPU and the GPU. Additionally, kernels must be written that express the operations that each thread must execute. Because of this, kernel based approaches result in more code than directive-based but come with the advantage of having more control over memory transactions, access patterns and multiprocessor utilization.

CUDA is written in C/C++ syntax and is compiled with the `nvcc` compiler, developed by Nvidia. Code written with CUDA offloads compute kernels onto Nvidia GPUs only. There are two options for making use of CUDA in Fortran applications : (1) Develop CUDA kernels in C/C++ and provide a wrapper and `ISO_C_BINDING` layer callable in Fortran, or (2) Develop CUDA-Fortran Kernels and use the PGI Fortran compilers. The first option brings in new challenges of expressing the GPU memory management in C/C++ while managing CPU memory in Fortran. This mixed language approach increases code complexity and requires developer teams to maintain a broad set of programming skills.

OpenCL provides a more flexible Kernel-based API that permits offloading to multicore CPUs, AMD and Nvidia GPUs, and Field Programmable Gate Arrays (FPGAs). As with CUDA, OpenCL code is written in a C/C++ syntax. It can be compiled with any C/C++ compiler with additional linker options that point to the headers and libraries for OpenCL. FortranCL, currently found on a repository maintained by Google [9], provides the `ISO_C_BINDING` to most of the OpenCL API so that it is accessible to Fortran code. Building code with FortranCL is similar to building its C/C++ counterpart except that you can use any Fortran compiler. The FortranCL project has not seen activity for about 6 years, and it's difficult to determine if there are any current production use cases.

HIP is a relatively new language from AMD. It is similar in syntax to CUDA and is advertised as an alternative to CUDA that permits offloading to both AMD and Nvidia GPUs. For current CUDA developers, AMD's software tools come with a HIPify script that is capable of converting most CUDA code to HIP. As with CUDA, HIP is written in C/C++ syntax and does not provide an out-of-the-box solution for Fortran developers.

III. Methodology

This project focuses on analyzing the performance portability of typical algorithms in computational fluid dynamics applied on an unstructured mesh. We investigate the performance portability of these routines accelerated with multi-threaded OpenMP and GPU offloading with OpenMP and CUDA. In addition, we explore the productivity of each of these parallelization approaches by measuring the number of line changes as an indicator of programmer effort.

A. Case Study

As a case study, we optimized two computational kernels from the open source Truchas code [10]. Truchas is a 3D multi-physics simulation tool for metal casting and other applications developed by the Los Alamos National Laboratory[1]. Truchas includes physics models for heat transfer, phase change, incompressible free-surface fluid flow, and several others. It uses unstructured meshes to model complex geometries and uses finite volume, finite element, and mimetic finite difference spatial discretizations. It currently uses MPI domain decomposition to implement data parallelism across nodes. Truchas is written in modern Fortran, making heavy use of object-oriented language features introduced in the 2003 standard. The three parallelism approaches (OpenMP CPU, OpenMP GPU, CUDA) are implemented in following two computational kernels of the Truchas code.

The first computational kernel calculates the cell-centered gradient of a cell-centered scalar field. The continuous domain is approximated using an unstructured grid. The cell-centered gradient is defined as the volume average of the gradient of the scalar ϕ over the domain Ω_i of cell i. The volume integral is converted to a surface integral of ϕ using the Gauss-Green Theorem. This is then approximated by summing the face-centered $\overline{\phi}_f$, face area A_f, and face normal $\widehat{m}_{fi}$ over $F(i)$, the set of faces associated with cell i, as shown in (1). This quantity is finally divided by the cell volume v_i.

$$\begin{aligned}(\quad\phi)_i &\equiv \frac{1}{v_i}\int_{\Omega_i} \quad\phi dV = \frac{1}{v_i}\oint_{\partial\Omega_i}\phi\widehat{m}dS \\ &\approx \frac{1}{v_i}\sum_{f\quad F(i)}\overline{\phi_f}A_f\widehat{m_f}_i\end{aligned} \tag{1}$$

The face-centered $\overline{\phi}_f$ are calculated from an average of node-centered $\widetilde{\phi}_n$ associated with face f, as shown in (2). Here, $N(f)$ is the set of nodes associated with face f, and $|N(f)|$ is the number of those nodes.

$$\overline{\phi}_f = \frac{1}{|N(f)|}\sum_{n\quad N(f)}\widetilde{\phi}_n \tag{2}$$

The node-centered values $\widetilde{\phi}_n$ are calculated by an average of the input cell-centered values ϕ_i associated with the node n's cell neighbors $C(n)$, weighted by the inverse of their volumes v_i, as shown in (3).

$$\widetilde{\phi}_n = \left(\sum_{i\quad C(n)}\phi_i/v_i\right)\Bigg/\left(\sum_{i\quad C(n)}1/v_i\right) \tag{3}$$

The second kernel computes the terms that come from a mimetic finite difference (MFD) discretization of the diffusion operator $\quad\cdot\vec{f}$, $\vec{f} = -\kappa\quad u$. In this hybrid formulation of MFD [11], the cell-centered unknowns $\{u_c\}$ are augmented with auxiliary unknowns $\{\lambda_f\}$ at mesh faces.

[1] The Truchas software is available at https://gitlab.com/truchas

Let f_j^k denote the flux from cell j through its kth side. Then the kernel computes the discrete divergence

$$\sum_k f_j^k \quad \text{for each cell } j, \tag{4}$$

and the flux discrepancy

$$f_j^k + f_{j'}^{k'} \quad \text{for each face } i, \tag{5}$$

where side k of cell j and side k of cell j , $j = j$, are the two cell sides that correspond to mesh face i. The flux f_j^k is given by

$$f_j^k = \kappa_j \sum_l a_{kl}^j (u_j - \lambda_j^l), \tag{6}$$

where λ_j^l is the λ value on the face corresponding to side l of cell j. The symmetric matrix $M_j{}^{-1} = ((a_{kl}^j))$ depends only on the geometry of cell j and its upper triangle is precomputed and stored in upper packed column format. Equation (6) represents a weak form of $\vec{f} = -\kappa \quad u$ integrated over cell j. The end result is the concise Fortran code shown in Listing 1.

```
rface = 0.0_r8
do j = 1, ncell
   flux = coef(j) * upm_matvec(minv(:,j),
                   ucell(j) - uface(cface(:,j)))
   rface(cface(:,j)) = rface(cface(:,j)) - flux
   rcell(j) = sum(flux)
end do
```

Listing 1. Original Mimetic Finite Difference Kernel

B. Challenges

Optimizing a Fortran scientific application like Truchas in a portable and productive way is challenging. The task is difficult due to the available software tools, limited support for Fortran features, and the unstructured mesh memory layout.

Truchas makes heavy use of array syntax and Fortran 2003 features such as object-oriented programming. This causes difficulties when using certain compilers, such as PGI. Features that are particularly troublesome with the PGI compilers include unlimited polymorphic variables, structure constructors, and deferred-length character variables. With some effort, the compiler issues can be fixed, but at present they limit the possible pathways to parallel portability.

The performance portability of Fortran applications is also limited by the available software tools. Kokkos [12] is a performance portable library that enables running the same code on multiple HPC platforms, such as GPUs and multi-core CPUs. However, Kokkos is a C/C++ package and does not natively support Fortran. Similarly, Raja [13] and SYCL [14] are only available for C++. The performance portability packages specifically targeted at Fortran include FortranCL and CUDA Fortran. However, FortranCL is no longer under active development, and only the PGI compilers support CUDA Fortran. In order to run on Nvidia GPUs, we therefore had to rewrite the computational kernels in CUDA C and implement a Fortran-C interface. Another option, OpenMP 4.0+ GPU offloading is a relatively new standard, and compiler support is still in development. Due to the xlf compiler's out-of-the-box support for the standard and relatively few hurdles for compiling Truchas, it was our compiler of choice when implementing OpenMP GPU offloading.

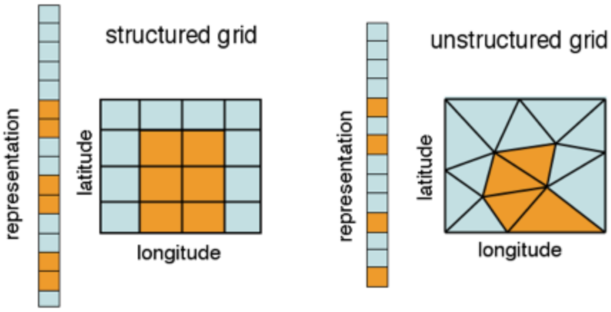

Fig. 2. Non-Contiguous Memory Access for Unstructured Mesh (From Stuebe [15]).

The use of unstructured meshes poses another challenge for optimizing the performance of Truchas. The indirect addressing characteristic to unstructured meshes make effective cache utilization more difficult in contrast to the regular access patterns of structured meshes. The multi-dimensional arrays common to large scientific applications also result in inefficient strides through memory.

Fig. 2 illustrates the optimization challenges posed by unstructured meshes. The orange highlighted cells of the structured grid on the left tend to be close together in memory and can be accessed in a regular pattern. By contrast, the highlighted triangular cells of the unstructured grid on the right can be anywhere in memory, resulting in inefficient caching.

C. Approaches

We chose to implement three different parallel frameworks for achieving performance portability for Truchas. These include OpenMP for the CPU, OpenMP for the GPU and CUDA C. What follows is an overview of each approach.

1) OpenMP CPU: Using OpenMP CPU multi-threading for on-node parallelism is simple: "`!$omp parallel do`" automatically distributes a loop's iteration space among all available threads. This pragma is used in Listing 2 to decompose the Truchas mesh to multiple threads. "`!$omp do private`" ensures that each thread has its own private copy of the specified variables within the enclosed region. The private variables are undefined upon entry to and exit from the OpenMP region. The code in Listing 2 computes nodal values by looping over cells and scattering contributions to the adjacent nodes. This leads to a race condition where multiple threads may read from and write to the same node value. One solution is to add “!$omp atomic” to serialize updating the particular memory location, ensuring correct results at the cost of decreased performance due to waiting threads.

```
!$omp parallel do private(j,k,n)
do j = 1, num_cell
  do k = cnode_size
     nodeid = cnode(k,j)
     !race condition for each cell
     !$omp atomic
     unode(nodeid) = unode(nodeid) + ucell(j)
                / volume(j)
    end do
end do
!$omp end do parallel
```

Listing 2. OpenMP CPU for Gradient Calculation Kernel Needs Atomic

In order to avoid using “atomic”, we can rewrite the kernel to loop over nodes and directly accumulate adjacent cell values as shown in Listing 3. This causes OpenMP threads to be to be distributed across nodes rather than cells, thereby avoiding the race condition. While this does involve repeated reads and calculations of the same cell values, the cost of the redundant accesses is more than offset by eliminating threads waiting on atomic operations.

```
!$omp parallel do private(j,c,cellid)
do j = 1, num_node
   !Iterate over cells adjacent to node
   unode(j) = 0.0_r8
   do c = 1, size(vcell(:,j))
      cellid = vcell(c,j)
      unode(j) = unode(j) + ucell(cellid)
                 / volume(cellid)
   end do
end do
!$omp end do parallel
```

Listing 3. OpenMP CPU for Gradient Calculation Kernel

2) OpenMP GPU: OpenMP GPU enables offloading subroutines, and the data they need, to target devices. The “!$omp target data” region specifies how data should be moved between the host and device. We specify which data to copy from host to device with “!$omp& map(to:)”. At the end of the “!$omp target data” region, all the data in the “!$omp& map(from:)” clause is copied back from the device to the host.

The “!$omp& map(tofrom:)” clause is a combination of the previous two, copying data to the device when entering the region, and moving the result back to the host when exiting it. For example, in Listing 4 we use the clause to initialize rface on the host, compute its value on the device, and make the result available on the host. Finally, we use “!$omp& map(alloc:)” to allocate variables that are only used on the device.

Once the allocations and data transfers are complete, the pragma on line 7 distributes the work among GPU threads. Like it’s CPU counterpart the “private” clause declares thread private variables.

It is worth noting that we had to include the “thread_limit” clause to explicitly limit the number of threads per team (i.e. CUDA block). The OpenMP runtime automatically created thread teams large enough to consume more than the maximum available memory per block on the tested Nvidia GPUs.

Fortran array syntax allows operating on array elements that are indexed with other arrays, as shown in Listing 1. This feature is implemented by allocating a temporary array to store the intermediate values. However, when run on the GPU the implicit allocations and de-allocations significantly impacted performance. We therefore created a tmp array to avoid the additional memory allocations. This change alone resulted in approximately a 2x speed-up over the previous iteration. Similar to OpenMP on CPU, this port also requires an tmp to avoid race conditions when different threads with the same faceid write to rface simultaneously.

```
1  rface = 0.0_r8
2  !$omp target data &
3  !$omp& map(alloc: flux,tmp)&
4  !$omp& map(to: ucell,uface,ncell,coef,minv,
       cface)&
5  !$omp& map(from: rcell)&
6  !$omp& map(tofrom: rface)
7  !$omp target teams distribute parallel do
       thread_limit(128) private(j,f,faceid,flux,
       tmp)
8  do j = 1, ncell
9    tmp = uface(cface(:,j)
10   tmp = ucell(j) − tmp
11   call upm_matvec(minv(:,j), tmp, flux)
12   flux = flux*coef(j)
13   do f = 1, size(cface,dim=1)
14     faceid = cface(f,j)
15     !$omp atomic
16     rface(faceid) = rface(faceid) − flux(f)
17   end do
18   rcell(j) = sum(flux)
19 end do
20 !$omp end target teams distribute parallel do
21 !$omp end target data
```

Listing 4. OpenMP GPU for the Mimetic Finite Difference Kernel

3) CUDA: CUDA is a kernel-based parallelization API developed by Nvidia for executing code on their GPUs.

As discussed in section III-B, we were unable to use CUDA Fortran with Truchas. Therefore, we had to rewrite the computational kernels in CUDA C, and wrote an interface that makes C functions callable from Fortran. We used the “ISO_C_BINDING” feature introduced in Fortran 2003 as shown in Listing 5.

```
subroutine function_C() bind(C,NAME ="
    function_C")
use, intrinsic :: ISO_C_Binding, only :
    C_DOUBLE, C_INT
```

Listing 5. Fortran-C Interface

CUDA code can be conceptually divided into host and device routines. Host routines manage the GPU from the CPU by launching kernels, initiating data copies, and synchronizing devices, among others. Device routines include CUDA kernels and any subroutines called by them.

The host code for the MFD kernel is shown in Listing 6. The code handles allocation and initialization of device data (lines 3-8), launching the C kernels that operate on the data (line 14), and copying the results back to the host (lines 16,17).

As discussed, the MFD kernel algorithm has a possible race condition when multiple threads with the same `faceid` write to `rface` simultaneously. Listing 7 shows device code where we used the CUDA intrinsic operator “`atomicAdd()`” to prevent data races. The 64-bit floating-point version of “`atomicAdd()`” is only supported by devices of compute capability 6.x and higher [16]. This requirement limits the portability of our CUDA code, although with some effort we could implement atomic operations compatible with devices of lower compute capability.

CUDA devices have a heap size of 8 MB by default [16]. Our original implementation of the MFD kernel allocated `d_flux` and `d_tmp` on the heap, resulting in runtime errors once the heap space was exhausted. Increasing the heap size resulted in correct execution.

We optimized the MFD kernel by using shared memory to reduce heap memory usage and minimize global memory accesses. Taking advantage of shared memory's low-latency dramatically sped up the kernel. It is worth noting that CUDA only allows one dynamic shared memory allocation. In order to allocate two arrays in shared memory, we declare a single large array and subdivide it accordingly.

```
1  void apply_diff_C(double *ucell,double *rcell,
       double *rface,...) {
2    // Copy field data (Init already done)
3    cudaMemcpyAsync(d_ucell, ucell, ... );
4    // Allocate local arrays
5    double *d_rcell, *d_rface;
6    cudaMalloc((void**) &d_rcell, ... );
7    // Initialize local arrays
8    cudaMemset(d_rface, 0, ... );
9    // TPB is Threads Per Block
10   int numBlocks = (ncell+(TPB-1))/TPB;
11   // NFC is the number of faces per cell
12   const size_t shared_mem = 2*TPB*NFC*sizeof(
         double);
13   //call CUDA kernel
14   apply_diff<<<numBlocks, TPB, shared_mem>>>(
         d_rcell, d_rface, ... );
15   // Copy data back to host
16   cudaMemcpyAsync(rcell, d_rcell, ... );
17   cudaDeviceSynchronize();
18   /* Wrap up free varables */
19   cudaFree(d_rcell);
20   }
21 }
```

Listing 6. Partial CUDA Host Code for the Mimetic Finite Difference Kernel

```
// Multi-dimensional array indexing
#define IDX2(i,j,i_stride)((i)+(i_stride)*(j))

__global__ void apply_diff_Cuda(double *
    d_rcell, double *d_rface, ... ) {
  const int j = blockIdx.x*blockDim.x +
      threadIdx.x;
  const int tid = threadIdx.x;
  // Split up shared memory
  extern __shared__ double shared[];
  double *flux = shared;
  // NFC is the number of faces per cell
  double *tmp = shared + blockDim.x * NFC;
  int faceid;
  if ( j < ncell) {
    for (int k=0; k < NFC;k++){
      faceid=d_cface[IDX2(k,j,NFC)]-1;
      tmp[IDX2(k,tid,NFC)] = d_ucell[j]
                           - d_uface[faceid];
    }
    upm_matvec_C(flux+tid*NFC, d_minv+j*
        UPM_SIZE, tmp+tid*NFC, NFC);
    for (int i=0; i < NFC;i++) {
      flux[IDX2(i,tid,NFC)] *= d_coef[j];
      faceid=d_cface[IDX2(i,j,NFC)]-1;
      atomicAdd(&d_rface[faceid],
               - flux[IDX2(i,tid,NFC)]);
      d_rcell[j] += flux[IDX2(i,tid,NFC)];
    }
  }
}
__device__ __host__ void upm_matvec_C(double *
    c, double *a, double *b, int size_b) {
 //Computes a matrix-vector product
              ...
}
```

Listing 7. Partial CUDA kernel code for the Mimetic Finite Difference Kernel

IV. Performance Portability and Productivity Analysis

We assess the performance of our initial parallel ports of the two Truchas kernels across different hardware, and evaluate the effort invested to achieve that performance.

A. Performance Results

In Fig. 3, the Gradient Kernel performance scales well initially as the number of cores increases, but scales poorly with more than one thread per core. Only KNL and Power9 gfortran continue to speed-up with more than one thread per core. However, in Fig. 4 the Mimetic Finite Difference Kernel shows only Power9 gfortran continues to speed-up with more than one thread per core.

In Fig. 5, Haswell outperforms KNL for small vector sizes due to its higher clock speed. KNL's vector units support a gather instruction that can reduce memory latency [17]. It is possible that this accounts for KNL's lower runtime for 256-bit vectors. The performance for 512-bit vectors likely decreased due to indirect addressing. However, in Fig. 6 there is no significant difference in performance across the different vector instruction sets. The consistent runtime is likely due to the large proportion of non-contiguous reads and writes of the MFD kernel.

In Fig. 7 and Fig. 8 the speed-up factor is relative to the serial kernels running on Haswell. Each approach is shown in Fig. 7 to speed-up the Gradient Kernel by at least a factor of 4. OpenMP on the CPU performs best due to powerful CPUs, namely Power9 and Skylake. OpenMP on the GPU and CUDA perform similarly on one GPU. Power9 xlf runs 4 times faster than Power9 gfortran

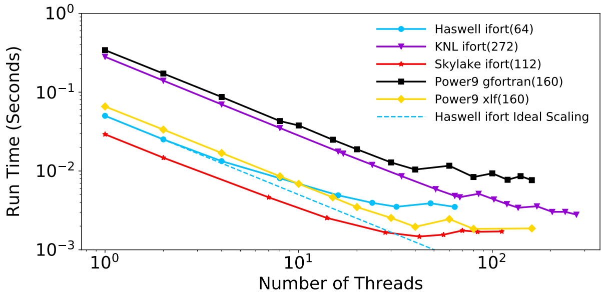

Fig. 3. Strong Scaling of OpenMP CPU using all Hyperthreads for the Gradient Kernel (lower is better)

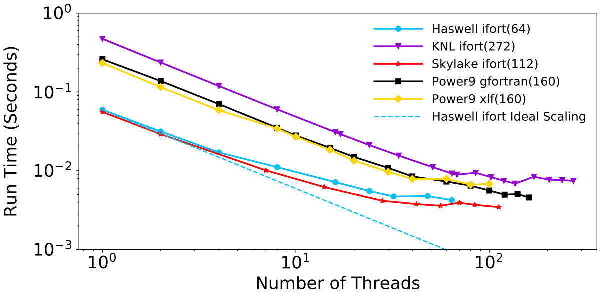

Fig. 4. Strong Scaling of OpenMP CPU using all Hyperthreads for the Mimetic Finite Difference Kernel (lower is better)

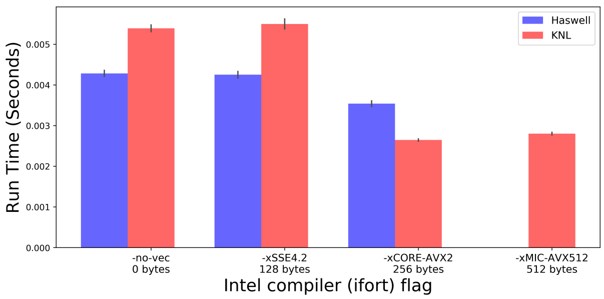

Fig. 5. Performance of Different Vector Instructions Sets for Gradient Kernel (lower is better)

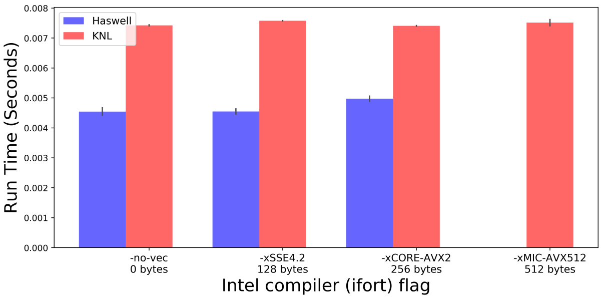

Fig. 6. Performance of Different Vector Instructions Sets for Mimetic Finite Difference Kernel (lower is better)

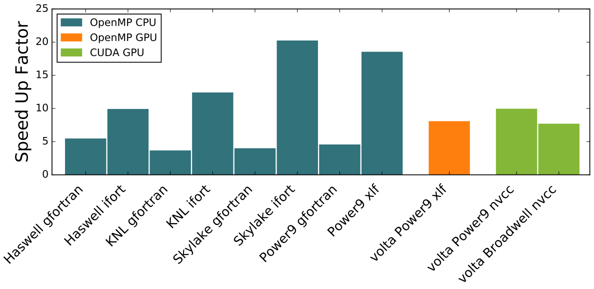

Fig. 7. Performance of Optimization Approaches for Gradient Kernel (higher is better)

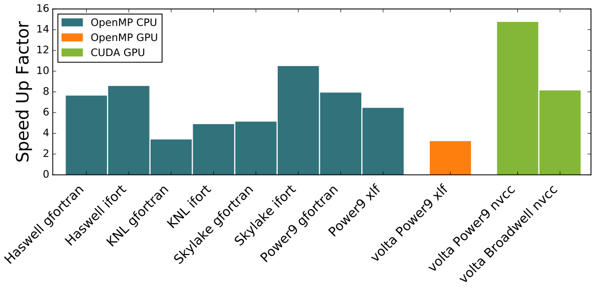

Fig. 8. Performance of Optimization Approaches for Mimetic Finite Difference Kernel (higher is better)

on same hardware but with different compiler. However, due to the complexity of the Mimetic Finite Difference Kernel in Fig. 8, Power9 xlf yields similar performance with Power9 gfortran. By using shared memory in the CUDA platform, it gains a huge speed-up.

Looking at CUDA performance with a different number of GPUs, Fig. 9 shows the speed-up is nonlinear for the Gradient Kernel as the number of GPUs increases. The speed-up factor is relative to the original serial code running on Haswell. The *Kernel* line corresponds to just the CUDA kernel runtime, while the *Total* line includes the time to copy the data between the host and device. The CUDA kernel scales to multiple GPUs, but the memory copies between mean that the total runtime doesn't scale past 2 or 3 GPUs. The performance is best for 3 GPUs.

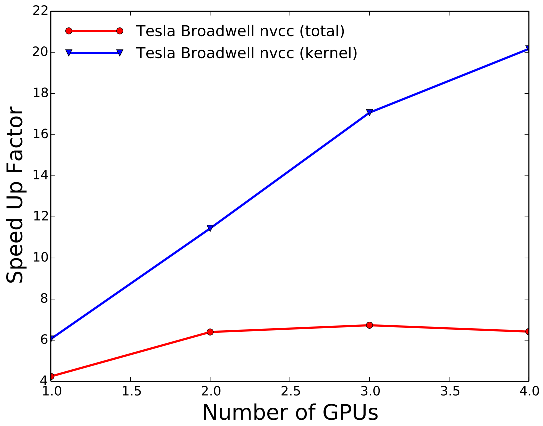

Fig. 9. Multi GPU performance for Gradient Kernel in CUDA (higher is better)

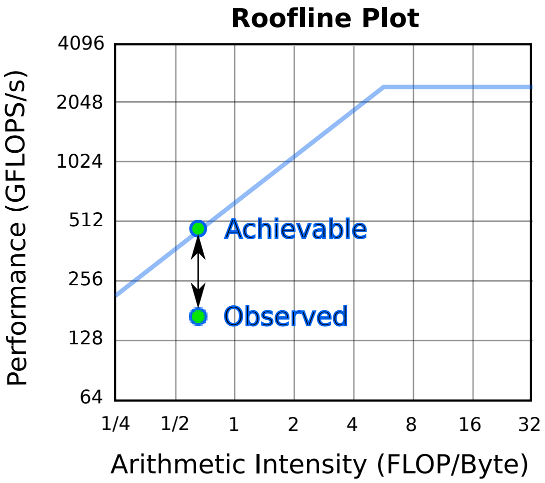

Fig. 10. The Architectural Efficiency is based on the main memory bandwidth from a roofline plot (Architectural Efficiency = observed/achievable).

TABLE I
A Performance Portability metric based on Architectural Efficiency for the Gradient Kernel

Architecture[1]	Measured GB/s	Stream Triad GB/s	Percent
. Haswell ifort (OpenMP CPU)	5.700	122.09	4.67%
KNL ifort (OpenMP CPU)	5.833	88.54	6.59%
Skylake ifort (OpenMP CPU)	14.389	226.56	6.35%
Power9 xlf (OpenMP CPU)	8.728	264.29	3.30%
Volta + Power9 xlf (OpenMP GPU)	585.777	788.75	74.27%
Volta + Power9 nvcc (CUDA)	590.108	787.97	74.89%
Volta + Broadwell nvcc (CUDA)	470.23	576.88	81.51%
Performance Portability			8.1%

[1] Hardware and software configurations from benchmarking are available in Appendix A

TABLE II
A Performance Portability metric based on Architectural Efficiency for the Mimetic Finite Difference Kernel

Architecture[1]	Measured GB/s	Stream Triad GB/s	Percent
. Haswell ifort (OpenMP CPU)	5.112	122.09	4.19%
KNL ifort (OpenMP CPU)	6.872	88.54	7.76%
Skylake ifort (OpenMP CPU)	12.436	226.56	5.49%
Power9 xlf (OpenMP CPU)	9.090	264.29	3.44%
Volta + Power9 xlf (OpenMP GPU)	42.659	788.75	5.41%
Volta + Power9 nvcc (CUDA)	612.368	787.97	77.71%
Volta + Broadwell nvcc (CUDA)	519.446	576.88	90.04%
Performance Portability			6.7%

[1] Hardware and software configurations from benchmarking are available in Appendix A

B. Performance Portability

To obtain a performance portability (Φ) value, we use the metric proposed by Pennycook, et al. [18]. This metric uses the harmonic mean of the performance relative to the best achievable performance on each platform as shown in (7).

$$\Phi = \begin{cases} \dfrac{|H|}{\sum_{i \quad H} \frac{1}{e_i(a,p)}}, & \text{if } i \text{ is supported } \quad i \quad H \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

where e is performance efficiency of application a for problem p on platform i in the set of platforms H. We apply this metric across a set of representative platforms using the architectural efficiency as shown in Fig. 10. To calculate the architectural efficiency, the roofline analysis was done in Intel Advisor for Intel CPUs as shown in Fig. 11, with nvprof on Nvidia GPUs or by running the stream benchmark. The platform performance is shown for both kernels in Table I and Table II and a performance portability metric is calculated for each of them.

C. Productivity

As shown in Fig. 12 of the Gradient Kernel, OpenMP on the CPU provides the best performance for the least effort. Adding CUDA enables the use of the GPU, but requires rewriting the kernel in CUDA C. OpenMP on the GPU performs similarly to CUDA, enabling GPU computation for significantly less effort. Similarly, Fig. 13 of the Mimetic Finite Difference Kernel shows the same pattern as in Fig. 12, but by implementing the shared memory in CUDA, it speeds up a lot. Using the effort logging method from [19], developers collect and compare the net line changes and implementation hours across the three approaches, Table III and Table IV both show that CUDA requires the largest amount code line changes and longest hours for planning, implementing, debugging while OpenMP CPU requires less line changes and little effort in hours.

In seeking a single metric to describe the code specialization for each platform rather than a single performance portable source, we look to some recent work by Harrell, Kitson, et al. [19]. They define a measure of *code divergence*, D, for a set of parallel code ports as given in (8) for a set of ports, A.

$$D(A) = \frac{\sum_{\{a_i,a_j\} \quad A:i<j} d(a_i,a_j)}{\binom{|A|}{2}} \tag{8}$$

The term $d(a_i,a_j)$ is a measure of distance between code ports where we define it more specifically in terms of the transformation, t, of the application, a, from a serial

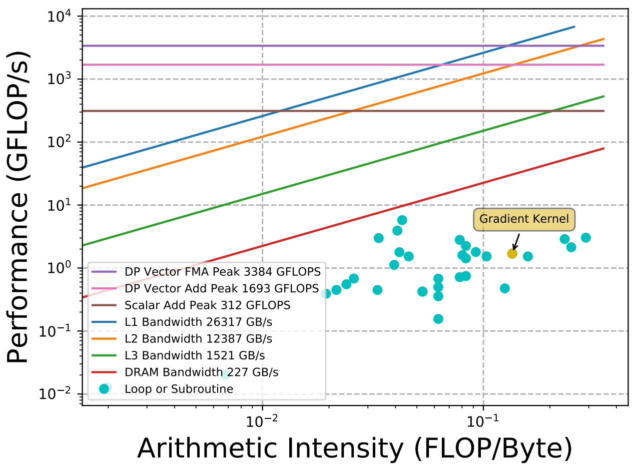

Fig. 11. Roofline of Skylake ifort on Gradient Kernel. The highlighted point represents the performance and arithmetic intensity of the Gradient Kernel function and all its subroutines.

TABLE III
EFFORT SUMMARY OF GRADIENT KERNEL

Approaches	Time to Adopt (in Hours)	Net Line Changes
OpenMP CPU	8	81
OpenMP GPU	11	139
CUDA GPU	17.5	257

TABLE IV
EFFORT SUMMARY OF MIMETIC FINITE DIFFERENCE KERNEL

Approaches	Time to Adopt (in Hours)	Net Line Changes
OpenMP CPU	18	28
OpenMP GPU	21	151
CUDA GPU	41	284

code to a parallel port, p. We define the distance between two parallel ports as the difference in source lines of code (SLOC) divided by the minimum size of all ports, as show in (9). The denominator of (8), $\binom{|A|}{2}$, is the number of unique pairs of ports. If the parallel version of the code is a true single-source, there would be no difference and we would get a *divergence* of 0.

$$d(a_{t_1,p_1}, a_{t_2,p_2}) = \frac{|\mathrm{SLOC}(a_{t_1,p_1}) - \mathrm{SLOC}(a_{t_2,p_2})|}{\min\{\mathrm{SLOC}(a_{t_i,p_i}) : a_{t_i,p_i} \quad A\}} \quad (9)$$

For measuring the distance between ports for Truchas, we use a *git diff* between the branches as given in Table V. Adding 28 and 81 to the 7223 original lines of code gives a minimum size of 7332 for the OpenMP CPU port. This yields a divergence metric of 4.58% of the code specialized on average in the parallel ports.

D. Discussion

This performance portability analysis of the initial Truchas kernel ports has some clear patterns. There are multiple indicators that the CPU implementations may have some opportunity for optimization improvements. These indicators include the good OpenMP thread scalability, the performance impact of different compilers, and the architectural efficiency measures for CPUs. We'll briefly discuss each in turn.

TABLE V
CODE DIVERGENCE METRIC FOR ALL PORTS

Compared Approaches	Distance
OpenMP CPU vs. OpenMP GPU	148
OpenMP CPU vs. CUDA	503
OpenMP GPU vs. CUDA	357
Divergence Metric	4.58%

The OpenMP thread scalability shown in Fig. 3 and Fig. 4 is excellent and would appear to be only good news. But when there is good parallel scalability, one of the reasons may be poor serial optimization. The reason is that as the data gets divided up, it is more likely to be in cache just because the data is smaller. This can in some cases even lead to super-linear speed-up where the parallel runs are faster than even the ideal scaling curve. But alone, the excellent scalability is not enough information to make any firm conclusion; it may just be a good parallelization implementation.

The second indicator is that compilers make a difference as shown in Fig. 7 and Fig. 8. If the algorithm inherently loads data in contiguous patterns, it would perform close to the stream bandwidth, and there would be no room for the compiler to make a difference.

The third indicator is that the architectural efficiency is less than 10% of the stream benchmark. This is equivalent to only using 1 value out of the 8 in each cache line. So we immediately suspect that the striding through the data is not contiguous. In real application kernels, especially in unstructured mesh applications, this may be unavoidable. Even in regular grid loops, when operating on the y-data (or x-data in some layouts), the stride will be larger than the cache length leading to poor cache utilization. Both x and y-data cannot be laid out ideally at the same time. Despite poor layouts, there may be ways to restructure the loops to minimize the cache misses.

It is also interesting that the architectural efficiency numbers for the GPUs are much better for the most part. This is because when a workgroup stalls on a memory load, the GPU automatically switches to another workgroup, effectively hiding some memory latency. Also the breakdown of data into small tiles naturally improves cache utilization. So for kernels with less than ideal memory loads, the CPU architectural efficiency will usually be much lower than the GPU's. This may not necessarily mean that the CPU implementation is poor; it may be the best that can be achieved, but it certainly bears some investigation.

The disparity between the CPU and GPU architectural efficiency results in a low performance portability metric, as shown in Table I and Table II. The metric was designed

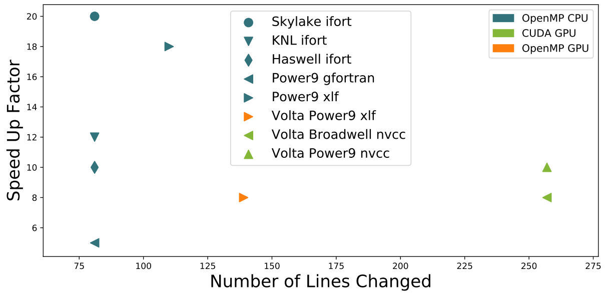

Fig. 12. Productivity of different approaches for Gradient Kernel (upper left is better)

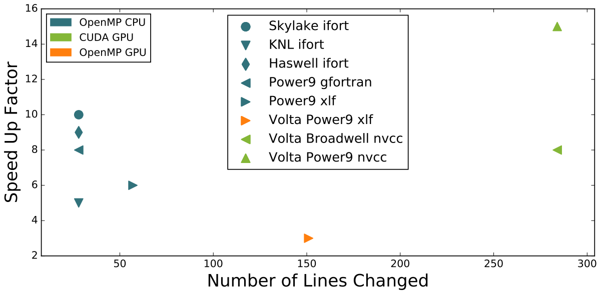

Fig. 13. Productivity of different approaches for Mimetic Finite Difference Kernel (upper left is better)

to mitigate the impact of large outliers [18], so the low CPU numbers bias the metric toward them. However, the low CPU architectural efficiency is common for unstructured grid applications where non-ideal memory accesses are often unavoidable. This investigation suggests that a low performance portability metric may be characteristic of applications with irregular geometries. The low metric also informs the direction of future work, suggesting that performance tuning on the CPU may well be worth the effort.

Another important observation is the significantly higher CUDA architectural efficiency for the mimetic finite difference kernel. This is due to the explicit use of shared memory in the implementation. The same is not seen for the OpenMP offload implementation for the GPU where the performance is much lower at about 5%. This is much different than the results for the gradient kernel. It points out that the directive-based languages may not be able to perform the same optimization in some cases as can be done in kernel-based implementations and performance may suffer. However, it takes significantly more development effort to write code for a kernel-based language as can be seen from the productivity data in Fig. 12 and Fig. 13 and Table III and Table IV.

OpenMP on the CPU is the most productive approach, requiring the least programmer effort for a good speed-up. It is also the most portable due to widespread compiler and hardware support. OpenMP on the GPU is a viable optimization approach, requiring little programmer effort, though performance compared to CUDA is mixed. This approach is currently limited to running on IBM hardware (Power9) using the xlf compiler, but it may become more portable as compilers adopt the standard. CUDA requires the most programming effort because it requires adding a Fortran-C interface and rewriting the computational kernel in CUDA C. But CUDA is the most attractive where performance is the highest priority because more fine-grained optimizations, such as using shared memory, are available. Portability is limited because the CUDA API can only run on Nvidia hardware. For this investigation we did not use CUDA-Fortran, which is supported only by the PGI compilers. Partly this is because it would take some work to get Truchas working with PGI, but there are also more limits on portability than CUDA-C.

Fully utilizing all available on-node GPUs could provide a significant performance improvement, especially given the increasing adoption of GPUs in high-performance computing.

V. Conclusion

We have shown the benefit of a cross-architectural performance analysis and the insights that it yields. The impact of irregular memory accesses of a complex physics kernel become clear for the different hardware. In particular, the disparity between the CPU and GPU architecture efficiency are probably characteristic of these types of physics kernels. The dramatic effect of the use of shared memory also becomes apparent, highlighting the need for a comparable optimization for the CPU implementation. The most important benefit of this study is a high-level view of where to allocate scarce optimization resources.

Through this study, we hope to encourage users and vendors to focus on the productive pathways to developing Fortran applications for exascale architectures. There are a few options and each has its strengths and weaknesses. The best choice for each code is dependent on the priorities of performance versus effort and the architectures that are targeted. Despite the recent progress, a satisfactory single-source solution for Fortran across all architectures is elusive.

Many Fortran codes also face additional limitations on portability because of issues with the Fortran toolchain. Implementation of Fortran 2003 and 2008 features [20] as claimed by vendors, is still lagging and even those that are implemented may not be fully working. Similar issues were encountered with the rest of the parallel Fortran toolchain, including the OpenMP 4.0+ and OpenACC implementations, and Fortran versions of CUDA and OpenCL.

Acknowledgment

Many thanks to our mentors Hai Ah Nam, Kris Garrett, John Pennycook, and Doug Jacobsen for thier guidance and help making this study possible.

Thank you to Joy Kitson and Stephen Harrell for their insights and help collecting productivity data.

Support provided by ASC Integrated Codes Telluride Project. Overall support provided by U.S. Department of Energy at Los Alamos National Laboratory supported by Contract No. DE-AC52-06NA25396.

Data collected on the Cori cluster at the National Energy Research Scientific Computing Center, and the Darwin and Kodiak clusters at the Los Alamos National Laboratory.

References

[1] OpenACC-Standard.org, "The OpenACC application programming interface version 2.6," November 2017. [Online]. Available: https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf

[2] OpenMP Architecture Review Board, "OpenMP application program interface version 4.5," November 2015. [Online]. Available: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[3] R. Allen, "Evaluating the performance of OpenACC in GCC," June 2018. [Online]. Available: https://blogs.mentor.com/embedded/blog/2018/06/06/evaluating-the-performance-of-openacc-in-gcc/

[4] PGI, "PGI accelerator compilers with OpenACC directives." [Online]. Available: https://www.pgroup.com/resources/accel.htm#spec

[5] Cray, "Cray Fortran reference manual (8.7) s-3901: OpenACC use." [Online]. Available: https://pubs.cray.com/content/S-3901/8.7/cray-fortran-reference-manual/openacc-use

[6] "GCC Wiki: OpenACC." [Online]. Available: http:/gcc.gnu.org/wiki/OpenACC

[7] P. Kapinos and D. an Mey, "Productivity and performance portability of the OpenMP 3.0 tasking concept when applied to an engineering code written in Fortran 95," *International Journal of Parallel Programming*, vol. 38, no. 5, pp. 379–395, Oct 2010. [Online]. Available: https://doi.org/10.1007/s10766-010-0138-1

[8] "HCC: Heterogeneous Compute Compiler." [Online]. Available: https://gpuopen.com/compute-product/hcc-heterogeneous-compute-compiler/

[9] "FortranCL: a Fortran 90 interface for OpenCL." [Online]. Available: https://github.com/xavierandrade/fortrancl

[10] D. A. Korzekwa, "Truchas – a multi-physics tool for casting simulation," *International Journal of Cast Metals Research*, vol. 22, no. 1-4, pp. 187–191, 2009. [Online]. Available: https://doi.org/10.1179/136404609X367641

[11] L. Konstantin, G. Manzinia, and M. Shashkovb, "Mimetic finite difference method," *Journal of Computational Physics*, vol. 257, pp. 1163 – 1227, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999113005135

[12] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[13] R. D. Hornung and J. A. Keasler, "The RAJA portability layer: overview and status," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2014.

[14] M. Wong, A. Richards, M. Rovatsou, and R. Reyes, "Khronos's OpenCL SYCL to support heterogeneous devices for C++," 2016.

[15] D. Stuebe, "Unstructured grid services." [Online]. Available: https://confluence.oceanobservatories.org/display/CIDev/Unstructured+Grid+Services

[16] *CUDA C Programming Guide Version 9.2*, Nvidia Corporation, August 2018. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[17] L. Q. Nguyen, "Compiling for the Intel® Xeon Phi™ Processor and the Intel® Advanced Vector Extensions 512 ISA," July 2016. [Online]. Available: https://software.intel.com/en-us/articles/compiling-for-the-intel-xeon-phi-processor-and-the-intel-avx-512-isa

[18] S. J. Pennycook, J. D. Sewall, and V. Lee, "A metric for performance portability," *arXiv preprint arXiv:1611.07409*, 2016.

[19] S. Harrell, J. Kitson, R. Bird, J. Sewall, D. Jacobsen, and R. Robey, "Effective performance portability," unpublished.

[20] "Fortran wiki: Fortran 2008 status." [Online]. Available: http://fortranwiki.org/fortran/show/Fortran+2008+status

[21] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Evaluating attainable memory bandwidth of parallel programming models via BabelStream," *International Journal of Computational Science and Engineering (special issue)*, 2017.

Appendix A
Artifact Description Appendix: Truchas Performance Experiments

A. Abstract

The ports to OpenMP for the CPU, OpenMP for the GPU and to CUDA with C kernels were the focus of this effort and for the performance studies.

B. Description

1) Check-list (artifact meta information):

- **Program:** Truchas
- **Compilation:** CUDA, Intel® Fortran Compiler, GFORTRAN and XLF
- **Run-time environment:** OpenMP environment variables set to best performing for each platform
- **Hardware:** Intel® Xeon® E5-2698 processor, Intel® Xeon Phi™ 7250 processor, Intel® Xeon® Platinum 8176 processor, Power9, Volta, Tesla
- **Output:** Kernel averaged timings reported by kernel driver programs

2) How software can be obtained : The Truchas application is available at the official Truchas GitLab repository (https://gitlab.com/truchas/truchas-release).

The computational kernels were extracted from Truchas into a separate Truchas Kernels repository, which is publicly available at the web address https://gitlab.com/truchas/pcsri2018/truchas-kernels. Each of the three optimization approaches was implemented in its own git branch, as described in Table VI.

TABLE VI
Truchas Kernels Optimization Approaches

Optimization Approach	Git Branch Name
OpenMP CPU	`openmp_orig`
OpenMP GPU	`openmp_orig_offload_one_memcpy`
CUDA	`cuda_orig`

3) Hardware: For this study we run on a variety of hardware detailed in Table VII.

TABLE VII
Truchas Performance: Hardware Specifications

Hardware Platform	Processor SKU	Cores per socket	Sockets per node	Threads per node	Memory per node (*GB*)
Intel® Haswell	Xeon® E5-2698	16	2	64	125
Intel® KNL	Xeon Phi™ 7250	68	1	272	94
Intel® Skylake	Xeon® Platnium 8176	28	2	112	376
IBM Power 9	8335-GTG	20	2	160	285
NVIDIA Volta	V100 SXM2	5120	1	N/A	16
NVIDIA TITAN V	V100 PCIE	5120	1	N/A	12
NVIDIA Tesla	P100 PCIE	3584	4	N/A	64

4) Software: The compiler version used on each platform:

- **Intel® Xeon® E5-2698 processor:** Intel® Fortran Compiler 18.0.2 or GNU Fortran 7.3.0
- **Intel® Xeon Phi™ 7250 processor:** Intel® Fortran Compiler 18.0.2 or GNU Fortran 7.3.0
- **Intel® Xeon® Platinum 8176 processor:** Intel® Fortran Compiler 18.0.2 or GNU Fortran 7.3.0
- **IBM Power 9:** IBM XLF 16.1.0
- **NVIDIA Volta:** CUDA 9.2 or IBM XLF 16.1.0
- **NVIDIA Titan V:** CUDA 9.2

5) Datasets: The "puck-cast" mesh file located in the meshes folder of the Truchas Kernels repository was used for all experiments.

C. Installation

See the current build instructions on the Truchas Kernels GitLab repository. The repository's config folder contains a cmake configuration file for each of the compilers used in the experiment. For each installation, the -DCMAKE_BUILD_TYPE=RELEASE cmake flag was used to enable compiler optimizations and disable debugging code.

D. Experiment workflow

The existing Gradient and Mimetic Finite Difference Kernel were extracted from the Truchas repository and placed into a timing harness to experiment with the different ports. The driver programs that execute and time the extracted kernels are available in the Truchas Kernels repository.

Each experiment was run on a single node. OpenMP settings OMP_PROC_BIND=spread and OMP_PLACES=cores were used for all OpenMP CPU experiments. The number of OpenMP threads was set to the maximum number of threads per node, as shown in Table VII.

Architectural Efficiency was computed as the ratio of main memory bandwidth of the computational kernel to the main memory bandwidth of a STREAM benchmark. The STREAM benchmarks for Intel® CPUs were obtained from the roof line plots generated by Intel® Advisor 2018. The STREAM benchmarks for IBM CPUs and Nvidia GPUs were calculated using the different STREAM implementations provided by the BabelStream [21] suite. BabelStream's OpenMP CPU, OpenMP GPU offloading, and CUDA implementations were compared against the corresponding optimization approaches.

E. Evaluation and expected result

Similar changes to the code can be made as is shown in the source code excerpts in the paper. Similar performance results should be obtained.

F. Experiment customization

Ports to other parallel frameworks and hardware can be made to see what performance might be obtained as well as the amount of code that needs to be changed. The hours required to make the port can be compared to those presented in the paper. An evaluation of a single-source pathway can be made along with a code divergence assessment.