MetaMorph: A Library Framework for Interoperable Kernels on Multi- and Many-core Clusters

Ahmed E. Helal
Dept. of Elec. & Comp. Eng., Virginia Tech
ammhelal@vt.edu

Paul Sathre
Dept. of Computer Science, Virginia Tech
sath6220@cs.vt.edu

Wu-chun Feng
Dept. of Computer Science and Dept. of Elec. & Comp. Eng., Virginia Tech
feng@cs.vt.edu

***Abstract*—To attain scalable performance efficiently, the HPC community expects future exascale systems to consist of multiple nodes, each with different types of hardware accelerators. In addition to GPUs and Intel MICs, additional candidate accelerators include embedded multiprocessors and FPGAs. End users need appropriate tools to efficiently use the available compute resources in such systems, both within a compute node and across compute nodes. As such, we present MetaMorph, a library framework designed to (automatically) extract as much computational capability as possible from HPC systems. Its design centers around three core principles: abstraction, interoperability, and adaptivity. To demonstrate its efficacy, we present a case study that uses the *structured grids* design pattern, which is heavily used in computational fluid dynamics. We show how MetaMorph significantly reduces the development time, while delivering performance and interoperability across an array of heterogeneous devices, including multicore CPUs, Intel MICs, AMD GPUs, and NVIDIA GPUs.**

***Index Terms*—parallel libraries; performance portability; programmability; accelerators; accelerator-aware MPI; GPU; MIC; CUDA; OpenCL; OpenMP; MPI; structured grids; exascale**

I. Introduction

In the last decade, several parallel computing architectures that span a wide range of execution models have emerged to meet the increasing demand for high-performance applications driven by large-scale data sets. Examples of these architectures are multicore CPUs, many-core GPUs, and Intel Many Integrated Cores (MICs). To attain scalable performance efficiently — relative to power, energy, and cost — the high-performance computing (HPC) community expects future exascale HPC systems to consist of multiple nodes, each with different types of hardware accelerators, connected over a high-speed, low-latency network infrastructure. In addition to the expected hardware accelerators, namely GPUs and Intel MICs, that populate the top end of the Green500 and TOP500 Lists [1], additional candidate accelerators for these heterogeneous computing systems include low-power embedded multiprocessors, custom hardware accelerators (potentially emulated on FPGAs), and even FPGAs themselves [2].

Such heterogeneous systems require hybrid programming models to exploit their potential performance and energy efficiency, and dealing with such interoperation between different devices and programming models is a tedious and error-prone task. In addition, the abundance of parallel architectures has complicated the design and development of high-performance applications even more. Scientists face several design choices and have to decide which architecture, programming model, algorithm, and implementation technique are the most suitable for their applications. For example, NVIDIA GPUs can be programmed via CUDA, OpenCL, OpenACC, potentially OpenMP 4.0, PTX, and several other research programming models. Each option has a different learning curve and potential performance, but typically the best performance requires low-level implementation approach and significant architecture expertise, which is in short supply.

Further, parallel architectures change faster than parallel programming models and software, and scientists should not have to spend their time re-learning and rewriting code for the new architectures. Thus, to effectively use future exascale computing systems, end users need appropriate tools to make efficient use of the available compute resources, both within and across compute nodes, and to do so without needing to have extensive architectural expertise and with minimal development time.

In this paper, we present MetaMorph, a library framework designed to (automatically) extract as much computational capability as possible from exascale computing systems with three core design principles: *abstraction, interoperability and adaptivity.*

- *MetaMorph abstracts current and future hardware accelerators behind a single interface.* It does so without complicated installations or extensive application refactoring that existing solutions require. This, in turn, supports the development and upgrade of accelerated applications by end users with minimal development effort and time. MetaMorph provides not only high-level abstraction, but also high performance, comparable to hand-coded and manually-tuned accelerated kernels (Section II-B).
- *MetaMorph promotes interoperability across different accelerators and with existing software.* MetaMorph's communication interface supports data exchange between *different* hardware accelerators not only in a single node, but also across multiple nodes. Moreover, unlike many existing solutions that use proprietary data types, MetaMorph's APIs are designed to be as close to pure C as

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00 ©2016 IEEE

possible, using standard data types, and expose internal device contexts to promote interoperability with hardware vendors' libraries, domain-specific libraries, and existing code. This interoperability is crucial in the adaptation to any new library framework (Section II-A).

- *MetaMorph is designed to be adaptive to the capabilities of the execution environment.* It is built to be modular and allow users to only select the components relevant to the application and hardware in hand. Further, it provides simultaneous access to all accelerators present in a system, promoting the development of an overarching runtime scheduling system to map accelerated computations to the best hardware platform(s) available and reduce the execution time. Additionally, transparent interfaces to both intra-node and inter-node transfers provide an opportunity for intelligent partitioning and pipelining to increase overlap of computation and communication and to hide data transfer latency (Section II-C).

To demonstrate the efficacy of our MetaMorph library framework, we present a case study with the *structured grids* design pattern, which is heavily used in computational fluid dynamics (CFD). Specifically, we evaluate MetaMorph with benchmarks and a larger application, namely MiniGhost, which is a representative CFD application for solving partial differential equations with the finite difference method. We demonstrate that MetaMorph significantly reduces development time for heterogeneous systems without performance penalty and can be used to seamlessly utilize *all* the available hardware accelerators across multiple compute nodes, which include multicore CPUs, Intel MICs, AMD GPUs, and NVIDIA GPUs. In addition, we show MetaMorph's interoperability with hardware vendors' libraries and third-party libraries such as clBLAS [3], Intel MKL [4] and MAGMA libraries [5] (Section IV).

II. Design Philosophy

MetaMorph is designed to effectively utilize HPC systems that consist of multiple heterogeneous nodes with different hardware accelerators. Figure 1 shows the proposed library framework. MetaMorph acts as middleware between the application code and compute devices, such as CPUs, GPUs, Intel MIC and FPGAs. It hides the complexity of developing code for and executing on heterogeneous platform by acting as a unified "meta-platform." The application developer needs only to call MetaMorph's computation and communication APIs, and the operations are transparently mapped to the proper compute devices. MetaMorph uses a modular layered design, where each layer supports one of its core design principles and each module can be used relatively independently.

A. Interoperability Layer

When designing complex scientific applications for HPC clusters, two types of interoperability are critical: (1) interoperability between nodes that may have different hardware

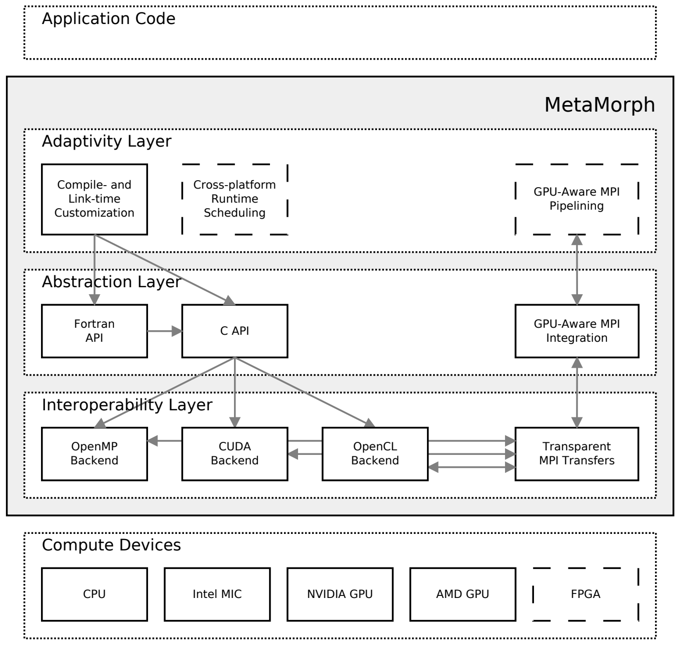


Fig. 1: MetaMorph uses a modular, layered design to hide the complexity of executing on and interoperating across a range of compute platforms.[1]

capabilities and (2) interoperability with existing code and external compute/communication libraries. Existing solutions only provide one of the two. Many frameworks provide multi-platform portability via complex data types or custom compute languages (often utilizing obtuse template metaprogramming), thus satisfying the first type of interoperability, but falling short of the second, as the entire application must be ported to the framework's types and/or language. Conversely, it is relatively easy to add a platform-specific library to an existing application that already uses that platform (e.g., adding cuBLAS [6] calls to a CUDA application), but these are *not* portable to other platforms, thus satisfying the second type of interoperability, but not the first.

MetaMorph, on the other hand, is designed from the ground up to support both types of interoperability. Figure 2 provides a sketch of this interoperatbility, and a concrete example is detailed in Section IV.

1) Interoperability Across Different Accelerators: MetaMorph satisfies this type of interoperability with its unified API design and transparent communication interface. When a user application calls a MetaMorph API, the call is transparently mapped to a back-end accelerator supported by the underlying platform and the MetaMorph library running on the node, which dramatically simplifies the programming required to coordinate multiple processes running on *different* hardware platforms. Further, MetaMorph provides a communication interface to transfer back-end-resident data, agnostic of the underlying execution platform, allowing for seamless interoperation across multiple nodes with a range of different hardware configurations.

[1]Dashed lines and boxes indicate components that are in development or not yet fully integrated into the prototype.

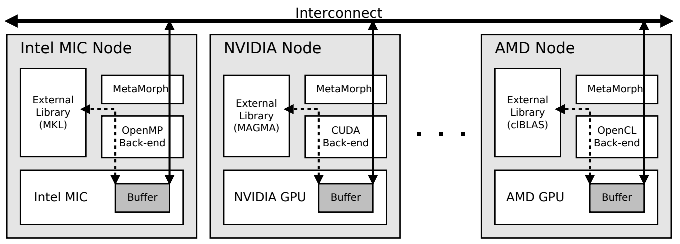


Fig. 2: MetaMorph provides interoperability both with external libraries in a node (dashed lines) and across nodes with varying hardware (solid lines).

2) Interoperability with existing software: MetaMorph satisfies this type of interoperability by careful design of its internal representation of data buffers and platform-specific back-ends. MetaMorph buffers are specified in the top-level APIs as simple C `void` pointers, with an enumerator specifying the primitive type that the back-end implementation should use. The back-end that a given void pointer actually resides on is inferred at run time from the state of the `run_mode` variable. To support incremental porting of existing applications developed for a specific platform, MetaMorph's internal context is exposed via extra API functions, so that back-end state can be shared directly with the host application (i.e., an application can share a context with MetaMorph via `meta_get_state` and `meta_set_state`). Thus, application developers can use MetaMorph, while they incrementally port and validate their applications, which significantly eases the transition process.

B. Abstraction Layer

Achieving (and more importantly, improving) performance in the real and changing world has become a function of portability; non-portable code stops gaining performance if (or more accurately, *when*) its target platform reaches end-of life. Code can be manually ported to new generations of hardware, but both functional and performance portability are often difficult to achieve without extensive expertise. Therefore, a need exists to future-proof programming solutions that obviate the demand for users to manually port performance-critical code to new hardware devices. The choice to provide this capability as a library framework was natural, given the proliferation of libraries in use in scientific applications and the vast range of domain- or platform-specific libraries for common operations, such as dense and sparse linear algebra.

MetaMorph provides programmability, functional portability, and performance portability by abstracting software back-ends (currently, OpenMP, CUDA and OpenCL) behind a single interface. As such, it bridges the performance-programmability gap by decomposing the problem space into two parts: high performance and high-level abstraction.

First, MetaMorph achieves high performance by providing low-level implementations of common operations, based on the best-known solutions for a given compute platform. Moreover, the software back-ends are instantiated and individually tuned for the different heterogeneous and parallel computing platforms (currently, multicore CPUs, Intel MICs, AMD GPUs, and NVIDIA GPUs).

Second, MetaMorph achieves high-level abstraction by hiding all device- and platform-specific details behind the unified interface, which enables the end-user to write the application once and run it on any supported device. Additionally, the portability criterion is further satisfied by providing an infrastructure for adding software back-ends for future compute devices — without end-user intervention or modifying the application. This provides the small population of early-adopter, architecture experts with a framework that enables them to dramatically extend the impact of their expertise to the wider community by expanding the library with new design patterns. So, rather than writing a kernel once for a single application, these experts can write that same kernel within the MetaMorph framework, provide it to the community, and allow it to be used across many applications.

Further, MetaMorph accelerates the development of new operations and computation/communication patterns, as shown in Figure 3. It provides a compilation infrastructure and helper APIs that handle the boilerplate initialization and compilation and simplify data exchange between the host and accelerators, such that MetaMorph developers can focus on developing the new kernels. In addition, we used our source-to-source translator [7], [8] to largely automate the generation of MetaMorph kernels, and in the future, we aim to release a family of such source-to-source translators and leverage the LLVM Just-In-Time compiler to simplify the expansion of the MetaMorph library with new design patterns.

Finally, existing kernels (e.g., CUDA kernels) can be included in MetaMorph without re-factoring by adding their implementation directly into the interoperability layer (e.g., CUDA backend) and their C/Fortran interface in the abstraction layer. However, advanced features, like seamless execution on different accelerators within a node and across nodes, will work only if these kernels are implemented in all the different back-ends. Contribution of a kernel for even a single back-end is valuable as it provides the architecture-expert community a baseline from which to implement and integrate kernels for the remaining back-ends.

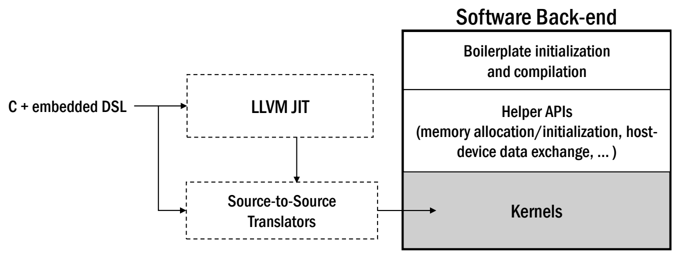

Software Back-end
Boilerplate initialization and compilation
Helper APIs (memory allocation/initialization, host-device data exchange, ...)
Kernels
C + embedded DSL
LLVM JIT
Source-to-Source Translators

Fig. 3: MetaMorph accelerates the development of new operations and kernels.

C. Adaptivity Layer

Software written for the heterogeneous and coming exascale era must be highly adaptive, as the expanding range of compute platforms and the increasing performance demands continually reshape the computing landscape. Accordingly, any library framework intended to provide high performance through such changes must itself be able to quickly respond to new opportunities to improve the performance. Therefore, MetaMorph has taken a modular layered approach that makes upgrading performance-critical components simple, without affecting user applications.

First, the adaptivity layer provides compile-time and link-time customization and optimization for performance-critical components. All performance-critical code for a given compute platform is encapsulated in a shared library object and separated from the core MetaMorph library and all other back-ends. This library-of-libraries construction promotes the optimization of back-ends to target specific compute devices, by making them disjoint code objects that can be tuned in isolation for new devices and then shared among the community. Further, the back-ends can be customized for different types of compute devices. For example, the OpenMP back-end can be customized for multicore CPUs and Intel MICs, and the OpenCL back-end can be customized for CPUs, AMD GPUs, NVIDIA GPUs and Intel MICs.

Second, the adaptivity layer provides run-time services to accelerate computations and data transfers. The accelerator-aware communication infrastructure is inspired by MPI-ACC [9], which allows automatic partitioning and pipelining of device-resident data buffers to hide data-transfer latencies. Another run-time service that is built into the MetaMorph design philosophy is cross-platform run-time scheduling, which intelligently maps the computations kernels to the best hardware platform(s) available, based on their relative performance, to reduce execution time. Our cross-platform scheduler is based on CoreTSAR [10], an adaptive run-time system. In summary, we highlight that our library framework facilitates the upgrading and development of such run-time services without significant modification to user applications, thus allowing end users to enjoy a “free ride” to better performance.

```
// Memory/Context Management
meta_alloc(void **ptr, size_t size);
meta_free(void *ptr);
meta_copy_h2d(void *dst, void *src, size_t size ...);
meta_copy_d2h(void *dst, void *src, size_t size ...);
meta_copy_d2d(void *dst, void *src, size_t size ...);
meta_set_acc(int acc, meta_mode mode);
meta_get_acc(int *acc, meta_mode *mode);
meta_flush(); //finish any outstanding work
// share meta_context with with existing software
meta_get_state(meta_platform *plat, meta_device *dev,
     meta_context *context, meta_command_queue *queue);
meta_set_state(meta_platform plat, meta_device dev,
     meta_context context, meta_command_queue queue);

// Communication Interface
meta_comm_init(int *argc, char *** argv);
meta_comm_finalize();
meta_packed_send(int dst, void *packed_buf, size_t len,
     meta_type_id type ...);
meta_packed_recv(int src, void *packed_buf, size_t len,
     meta_type_id type ...);
meta_pack_send(int dst, meta_face *face, void *buf, void *
     packed_buf, meta_type_id type ...);
meta_recv_unpack(int src, meta_face *face, void *buf, void
     *packed_buf, meta_type_id type ...);
//data marshaling
meta_pack_face(void *packed_buf, void *buf, meta_face *face
     , meta_type type ...);
meta_unpack_face(void *packed_buf, void *buf, meta_face *
     face, meta_type type ...);
meta_transpose_face(void *ind, void *outd, dim2 *size,
     meta_type type ...);
meta_face *make_slab(meta_slab_pos position, void *buf,
     dim3 *size, int thickness ...);

// Timers
meta_timers_init();
meta_timers_finalize();

// Compute Kernels
meta_kernel_name(...);
...
```

Fig. 4: Overview of the main user API exposed by MetaMorph

III. Prototype Implementation

A. A Library of Libraries

We realize MetaMorph[2] as a layered library of libraries. Each tier implements one of the core principles of abstraction, interoperability, and adaptivity. The top-level user APIs and platform-specific back-ends exists as separate shared library objects, with interfaces designated in shared header files. Primarily, this encapsulation supports custom tuning of back-ends to a specific device or class of devices, as we mentioned in Section II. In addition, it allows back-ends to be separately used, distributed, compiled, or even completely rewritten, without interference with the other components.

B. Programming Models

The core API, library infrastructure and communication interface are written in standard C for portability and performance. Individual accelerator back-ends are generated in C with OpenMP and optional SIMD extensions (for CPU and Intel MIC), CUDA C/C++ (NVIDIA GPUs), and C++ with OpenCL (AMD GPUs/APUs and other devices). In addition, a wrapper around the top-level API is written in polymorphic Fortran 2003 to simplify interoperability with Fortran applications prevalent in some fields of scientific computing.

C. Top-Level User API

The top-level API, shown in Figure 4, improves the programmability of user applications by abstracting the back-ends, which provide accelerated kernels for each platform. Specifically, it implements the offload/accelerator computation model[3], in which data is explicitly allocated, copied, and

[2]The prototype implementation of MetaMorph can be downloaded from https://github.com/vtsynergy/MetaMorph

[3]For API cohesiveness, the OpenMP back-end mimics the offload model. However, redundant data transfers can be eliminated, under the user’s control, via the *USE_UNIFIED_MEMORY* option.

manipulated via kernels within the MetaMorph context. In addition, the front-end interface supports seamless execution of user applications on different accelerators. It intercepts calls to the MetaMorph communication and computation kernels and transparently maps them to a back-end accelerator supported by the underlying platform. The only times that a user application needs to explicitly manage platforms are as follows:

- at compile time, when the library is advised what back-ends might be available (via conditional compilation with `-D WITH_`*`BACK-END`* definitions), and
- at run time, when the execution mode is set to one of the compiled-in back-ends (via the `METAMORPH_MODE` environment variable or a call to `meta_set_acc()`).

1) Memory Management: The top-level API contains memory management calls for allocating/freeing a MetaMorph buffer and transferring data to/from the host. MetaMorph implements its own implicit buffer types for a set of primitive data types — currently single-precision and double-precision floating points, signed and unsigned integers, and unsigned 64-bit integers — using a type enumerator and void pointer(s). Thus, individual API wrappers can dynamically map the provided void pointer to the correct type for the back-end implementation by inferring the backend-native type at run time from the global `run_mode` variable. In addition, these API wrappers take standard C types for scalars and `size_t[N]` vectors for N-D problem size variables, which are transmuted to appropriate types, when the back-end implementation performs the actual kernel launch.

2) Context Management: A number of functions are exposed to exert high-level control on MetaMorph, for example, getting/setting the current execution back-end, forcing outstanding asynchronous work to complete, and sharing MetaMorph context with exisiting software. To ensure compatibility and minimize performance overhead, when only a subset of capabilities is needed, the top-level API code uses conditional compilation, such that the library users only pay memory and performance overhead for the back-ends and options that are needed by "opting-in" at compile time.

3) Communication Interface: MetaMorph has been designed from the start for heterogeneous clusters, hence ensuring convenient data exchange between its processes is critical.

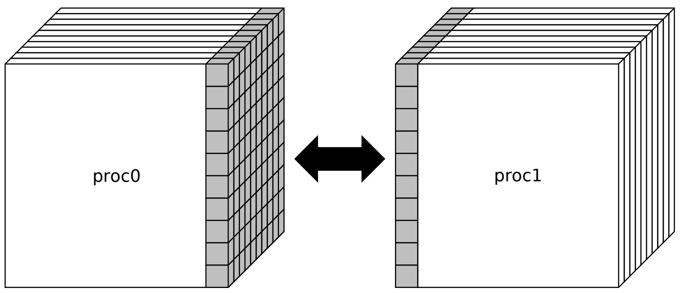


Fig. 5: Processes must exchange neighbor data every iteration.

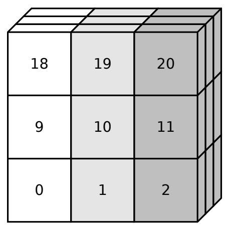
Domain-decomposition. In many computational science domains, including CFD, when the problem domain is sufficiently large and cannot fit on a single compute device or node, the domain is decomposed into smaller sub-domains that can be computed relatively independently. However, once decomposed, each sub-domain needs access to a current copy of data from its logical neighbors. This results in a compute-then-communicate iterative loop, where neighboring sub-domains residing in separate memory spaces must synchronize their edge data each iteration, i.e., exchange a face or slab of the sub-domain with neighbors, as shown in Figure 5.

Face Specification. As noted above, boundary (ghost) element exchange is an important communication pattern in many scientific domains. When N-dimensional grids are stored as standard C arrays, only the two faces at the low and high end of the N-2 dimensions are stored contiguously, and the remainder have their elements scattered at well-defined stride offsets. Therefore, a concise representation is needed for defining the set of memory addresses that make up a face, so that they can all be appropriately read and exchanged.

Rather than using a pointer list, we re-purpose the *gslice* data structure from `libstdc++` that is designed specifically for recording such structured offset information about an N-dimensional array. The data structure represents the offset computation as a tree of height N in which each successive level from the root is a finer stride through memory. That is, the leaf nodes represent the unit-stride dimension, their parents represent the dim_0-stride dimension, grandparents the $dim_0 * dim_1$-stride dimension, and so on. Figure 6 shows an example of this data structure, which requires $O(D)$ memory space (where D is the dimension of the data grid).

Data Marshaling. On-device packing and unpacking of a multidimensional dense grid can significantly reduce data transfer and synchronization overhead. So, instead of using host-side data marshaling, we provide per-back-end variants of parallel gather/scatter operations to support the exchange of portions of a back-end-resident data buffer. In these kernels, threads compute indices into the unpacked buffer in parallel from the face specification, then perform a direct copy between the packed and full buffers.

Communication Meta-Operations. We expose a family of four operations that provide data exchange with other MetaMorph processes and back-ends, in both blocking and non-blocking modes. We provide simple `packed_send` and `packed_recv` APIs, which transparently move data between the back-ends via a host-side staging buffer. In addition, two more operations are provided: `pack_send` and `recv_unpack`. These operations allow the user to specify face exchanges at a higher level of abstraction that concisely describes the exchange being performed. Currently, the user only provides the target process, the back-end-resident N-D data buffer and packed buffer, and a valid face specification as described above. Moreover, MetaMorph's communication interface has preliminary GPU-Direct support for exchanging

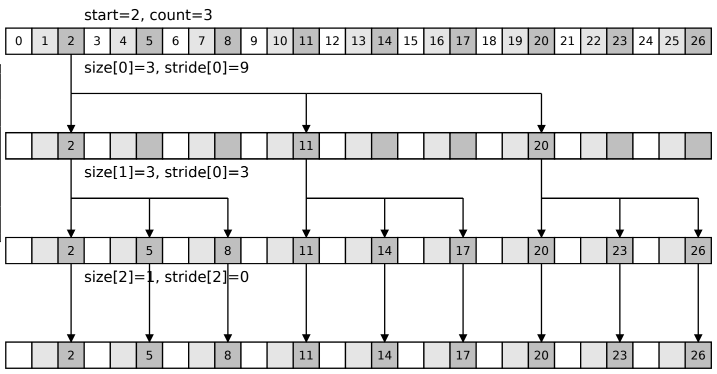


(a) A row-major 3D region, with unit-dimensional indicies

```
count=3
start=2

size[3]={3,3,1}
stride[3]={9,3,0}
```

(b) Specification of right-most 2D face with compressed indexing structure



(c) Hierarchical computation of strided layout of face indices

Fig. 6: Face description data structure and heirarchical index computation.

buffers directly between two MetaMorph processes with the CUDA back-end and a GPU-aware MPI, such as MVAPICH2.

Asynchronous Communication. To overlap computation with communication, MPI exchanges are frequently performed in asynchronous mode. As such, all exchange functions can operate in either blocking or non-blocking mode (via callbacks and helper functions). In practice, each high-level exchange operation consists of two to four asynchronous steps. For example a `pack_send` on a non-GPU Direct back-end must 1) run the pack kernel asynchronously, 2) whenever the pack kernel finishes, perform an asynchronous device-to-host copy of the packed buffer, 3) send the host buffer using MPI_ISend , and 4) free the temporary host buffer.

Therefore, an infrastructure is provided to coordinate such asynchronous pipelines, which is built from three main components: an MPI request queue, helper functions that trigger after requests finish, and callback functions that are triggered by the back-ends after kernels/transfers finish. When an asynchronous MPI operation is invoked, its corresponding request is registered on a queue alongside the type of meta-operation being performed, and the necessary data and helper function required to finish any remaining work. At this time, submitted requests are only checked for completion, when a `meta_flush` is called. However, there is an opportunity to check for completion more frequently. Each time an asynchronous device kernel or transfer is invoked, the necessary data and callback function can be specified such that the back-end run-time triggers the remaining work upon completion of the device operations.

4) Companion Features: The final software component in the front-end API is a set of companion features that provide optional capability, which may not be needed by all users. Currently, we provide the following features:

- Timing infrastructure that performs transparent timing of kernels and data transfers across the different software back-ends.
- Fortran compatibility that exposes both polymorphic Fortran 2003 and `ISO_C_BINDINGS`-compatible versions of the user-level API.

D. Accelerator Back-Ends

Back-ends are less uniformly constructed as a consequence of the dissimilarity of platform-specific programming. However, ultimately each is responsible for providing a standard C interface to the accelerated kernels. They are segregated from one another in order to allow separate compilation and encapsulation of platform-specific nuances. Consequently, if a given back-end requires special-purpose libraries or tools to build that are not present on the target machine, it can be easily excluded from a given build of the library as a whole without loss of function in the remaining back-ends.

1) OpenMP Back-End: The OpenMP back-end provides standard C variants of all API functions and should be considered the default back-end, as it provides functionally correct results on any CPU, regardless of whether the compiler respects OpenMP pragmas. For some kernels, additional compile-time options are provided to further accelerate code, such as the option to use AVX intrinsics.

2) CUDA Back-End: The CUDA back-end includes both the kernel functions and the host-side wrappers responsible for executing the kernel — and when necessary, auto-generating a CUDA grid/block configuration from the provided problem size. It uses the CUDA C *execution configuration* syntax and mixed device/host source. In most cases where a kernel supports multiple data types, simple templates are used in order to minimize code duplication.

3) OpenCL Back-End: The OpenCL back-end is similar to the CUDA back-end, with a few exceptions. OpenCL kernels and host code are stored in separate files, as is common in OpenCL development. The OpenCL host code includes functions for automatically performing the OpenCL initialization boilerplate. This includes selection of OpenCL platform and device, construction of a command queue and context for executing on the device, just-in-time compilation of kernel code, and management of the resulting program and kernel objects.

IV. A CASE STUDY WITH STRUCTURED GRIDS

We present a case study with the *structured grids* design pattern, which is heavily used in computational fluid dynamics. We demonstrate MetaMorph's capabilities with benchmarks and a representative CFD application. We show that a program written once using MetaMorph's abstraction layer can seamlessly utilize a wide range of hardware accelerators across multiple compute nodes. Moreover, we show MetaMorph's interoperability with platform-specific libraries. We evaluate MetaMorph on an experimental heterogeneous cluster with multicore CPUs, Intel MICs, NVIDIA GPUs, and AMD GPUs. In addition, we perform scalability analysis on a large-scale CPU-GPU cluster.

A. Applications

1) 3D Dot-Product Benchmark: We designed a benchmark that simulates the exchange of boundary or ”ghost” regions in a structured grid computation, followed by a global dot-product on a 3D grid. This benchmark represent a heavily-used pattern in iterative solvers for partial differential equations, e.g., conjugate gradient (CG), biconjugate gradient stabilized (Bi-CGSTAB), and generalized minimum residual (GMRES). Algorithm 1 describes our test benchmark that models a 3D structured grid computation on a domain of size Nx×Ny×Nz. After initialization, the global domain is decomposed into multiple regions (sub-domains) of size nx×ny×nz, which are logically connected in a torus along the X dimension, and each local domain is assigned to a process. Each MPI process exchanges boundary elements with neighbors, and performs a 3D dot-product on its local domains. The final result is computed by combining the partial results from all processes.

Algorithm 1 3D Dot-Product Benchmark

```
 1: INPUT: $Nx \times Ny \times Nz$                ▷ global domain size
 2: INPUT: $N$                                      ▷ number of sub-domains
 3: INPUT: $Iters$                                  ▷ number of iterations
 4: OUTPUT: $result$                                ▷ global 3D dot-product result
 5: Allocate and initialize global domain
 6: Perform domain-decomposition along the $X$ dimension
 7: Allocate and Initialize local domains of size $(nx+1) \times ny \times nz$
 8: for $i \in$ 0:Iters-1 do
 9:     Pack ghost cells into send_buffer
10:     Send send_buffer to $proc + 1$ process
11:     Receive ghost cells from $proc - 1$ process into recv_buffer
12:     Unpack ghost cells from recv_buffer into local domains
13:     Compute 3D dot-product on the local domains
14:     Perform global reduction to compute $result$
15: end for
16: return $result$
```

2) MiniGhost: MiniGhost [11], [12] is a representative (proxy) application for CTH [13], a multi-material shock hydrodynamics code developed at Sandia Lab to model hydrodynamic flow and dynamic deformation of solid materials. MiniGhost solves the partial differential equations (PDEs) of multiple variables, which represent a material state such as energy, mass and momentum, using the finite difference method. It implements a difference stencil (e.g., 3D seven-point stencil) and explicit time-stepping scheme on a homogenous 3D domain (grid). MiniGhost supports two communication modes: BSPMA (bulk synchronous parallel with message aggregation) and SVAF (single variable, aggregated face data). In BSPMA, the boundary data for all variables are transmitted in aggregated messages, while in SVAF the boundary data for each variable is sent in a dedicated message. When the number of variables is one, the two communication modes are equivalent. Algorithm 2 illustrates the main computations and communication patterns in MiniGhost with SVAF communication mode. After initialization, the global domain is decomposed along the X, Y, and Z dimensions into multiple sub-domains, and each sub-domain is mapped to a process. In each time step, the processes exchange boundary elements (2D faces) with neighbors that share a face in the X, Y, and Z dimensions, and apply a difference stencil on their sub-domains. Finally, the value of the global domain is computed using global summation.

Algorithm 2 MiniGhost with SVAF communication mode

```
 1: INPUT: $Nx \times Ny \times Nz$                ▷ global domain size
 2: INPUT: $N$                                      ▷ number of sub-domains
 3: INPUT: $Nvar$                                   ▷ number of variables
 4: INPUT: $TimeSteps$                              ▷ number of time steps
 5: INPUT: $Stencil$                                ▷ stencil type (3D7P, 3D27P,...)
 6: OUTPUT: $GridSum[Nvar]$                         ▷ global domain value
 7: $b$              ▷ number of neighbors per dimension (2 for 3D7P stencil)
 8: Allocate and initialize global domain
 9: Perform domain-decomposition along the $X$, $Y$ and $Z$ dimensions
10: Allocate and initialize local domains of size $(nx+b) \times (ny+b) \times (nz+b)$
11: for $i \in$ 0:TimeSteps-1 do
12:     for $j \in$ 0:Nvar-1 do
13:         Pack boundary data into send_buffer
14:         Send send_buffer to neighbors
15:         Receive boundary data from neighbors into recv_buffer
16:         Unpack boundary data from recv_buffer into local domain
17:         Apply boundary conditions on the local domain
18:         Compute finite-difference stencils on the local domain
19:         Perform global reduction to compute $GridSum[j]$
20:     end for
21: end for
22: return $GridSum$
```

B. Programmability and Productivity

Figure 7 shows the MetaMorph compute APIs that we used to accelerate the target applications, in addition to the core APIs (Figure 4). We implemented a MetaMorph version of the 3D dot-product and MiniGhost applications, and to show our interoperability with external libraries, we created a variant of the 3D dot-product benchmark using MetaMorph and the platform-specific BLAS libraries clBLAS, Intel MKL and MAGMA. Since the platform-specific BLAS libraries do not provide data marshaling primitives nor accelerator-aware communication interface, we use MetaMorph to do so, and only perform the dot-product operation with BLAS libraries.

```
// Compute Kernels
meta_stencil(void *ind, void *outd, dim3 *size, dim3 *start
    , dim3 *end, meta_type type ...);

meta_dotProd(void *ind1, void *ind2, dim3 *size, dim3 *
    start, dim3 *end, void *result, meta_type type ...);

meta_reduce(void *ind, dim3 *size, dim3 *start, dim3 *end,
    void *result, meta_type type ...);
```

Fig. 7: MetaMorph's compute APIs that are used to accelerate the target applications

Application	OpenMP	MetaMorph	MetaMorph+BLAS
3D dot-product	10	19	40
MiniGhost	23	38	NA

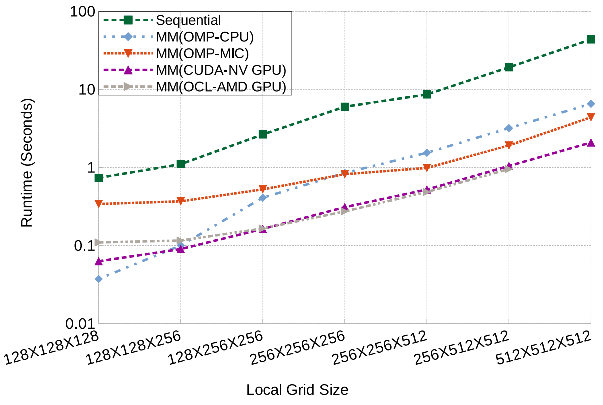
TABLE I: The number of effective code lines changed/added to accelerate the baseline MPI implementation

To evaluate the programmability of MetaMorph, we use the number of effective code lines changed or added to accelerate the baseline MPI version as our metric. We are interested in the source code lines that perform the core functionality. So, we don't consider any code lines used for profiling, timing, debugging or optional features. In addition, we consider that the applications use a single data type (double), although applications accelerated with MetaMorph can use five different data types without code modifications. Table I shows the lines of code changed or added to accelerate the sequential applications using MetaMorph, MetaMorph with platform-specific BLAS libraries, and directive-based programming models such as OpenMP. Due to its abstraction layer, MetaMorph has competitive programmability and productivity with OpenMP, and unlike OpenMP, it provides access to several heterogeneous accelerators. Moreover, through the interoperability layer, MetaMorph enables the user to utilize platform-specific libraries with an additional programming overhead.

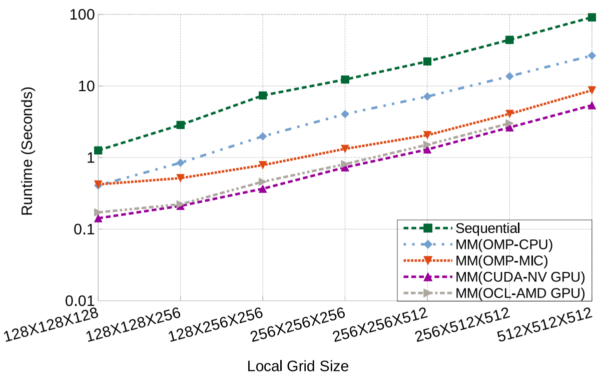
C. Experimental Results

Due to the lack of large-scale heterogeneous cluster with different accelerators, covering all currently-supported backends, we evaluate MetaMorph on an experimental heterogeneous cluster with multicore CPUs, Intel MICs, NVIDIA GPUs, and AMD GPUs. In addition, we perform scalability analysis on a large-scale CPU-GPU cluster. Details of the experimental nodes and the CPU-GPU cluster are provided in Table II and Table III, respectively. In the experiments, MiniGhost is configured to apply a 3D 7-point stencil on a global grid and to use an explicit time-stepping scheme with 100 time steps. We do not include sequential overheads such as data initialization and boundary conditions in the reported performance. While MetaMorph supports several datatypes, we use double-precision floating point only for brevity.

Figures 8, 9, and 10 show the performance of the 3D dot-product and MiniGhost applications on the experimental heterogeneous cluster. In the experiments, we launch four processes, each running on one of the supported back-end accelerators: multicore CPU (Intel Xeon E5-2697), Intel MIC (Intel Xeon Phi SC7120P), NVIDIA GPU (K20Xm) and AMD GPU (AMD Radeon 7970). We use weak scaling with local grid sizes that are typically used in CFD applications. However, AMD Radeon 7970 could not execute the problem size of 512x512x512, due to its limited global memory. Since the test resources are only interconnected with a high-traffic 1Gb ethernet, shared with approximately 30 other nodes, resulting in too much network noise for meaningful intercommunication performance characterization, we do not include the inter-node data transmission time in the reported performance.



(a) 3D dot-product benchmark



(b) MiniGhost

Fig. 8: Performance of the target applications with the different MetaMorph (MM) back-ends on the experimental cluster

Figure 8 shows the execution time of each process of the 3D dot-product and MiniGhost applications running on one of the supported back-ends in comparison with the sequential reference implementation running on Intel Xeon E5-2697. The results show that MetaMorph achieves up to 21x and 17x speedup over the serial implementation in 3D dot-product and MiniGhost, respectively. Since structured grids applications are characterized by regular memory access pattern and low computational intensity, their performance is limited by the memory system; hence, they are suitable for many-core accelerators (GPUs and Intel MICs) with large memory bandwidth. However, the problem size must be large enough such that the kernel launch overhead and the additional data transfers are effectively amortized.

Figure 9 shows the run-time distribution of MiniGhost with 256x256x512 local problem size on the different MetaMorph back-ends. The main computation and communication kernels of MiniGhost are stencil, reduction sum, data marshaling, and intra-node data transfer. On many-core accelerators, the intra-node data transfer includes host-to-device, device-to-host, and on-device data movement, while on CPUs it includes on-device data movement. MiniGhost has many on-device data transfers, as its finite-difference solver uses temporary work buffers to hold the intermediate results, while solving the

Machine Name	CPU(s)	Accelerator(s)	OS	MPI	Compiler(s)	Opt. Level
ht20	Intel Xeon E5-2697 v2 @ 2.70 GHz (2x)	NVIDIA Tesla K20Xm (2x)	Debian Jessie	MPICH 3.1.4	gcc 4.8.2 nvcc 6.0.1	-O3
dna1	Intel Core i5-2400 @ 3.10 GHz	AMD Radeon 7970	Debian Wheezy	MPICH 3.1.4	gcc 4.7.2	-O3
mic	Intel Xeon E5-2697 v2 @ 2.70 GHz (2x)	Intel MIC SC7120P (2x)	CentOS Linux 6	MPICH 3.1.4	icc 13.1.1	-O3

TABLE II: The experimental cluster's configurations

Cluster Name	CPU(s)	Accelerator(s)	OS	MPI (Interconnect)	Compiler(s)	Opt. Level
HokieSpeed	Intel Xeon E5645 @ 2.40 GHz (2x)	NVIDIA Tesla C2050 (2x)	CentOS Linux 6	OpenMPI 1.6.4 (QDR InfiniBand)	gcc 4.5/icc 13.1 nvcc 5	-O3

TABLE III: The CPU-GPU cluster's configurations

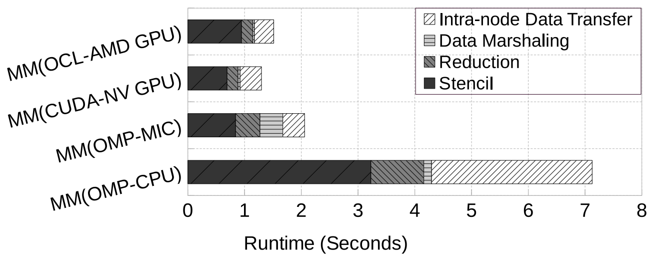


Fig. 9: The run-time distribution of MiniGhost using a 256X256X512 local grid with the different MetaMorph (MM) backends on the experimental cluster

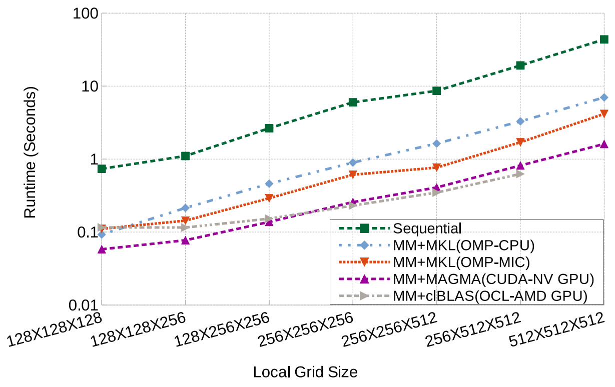
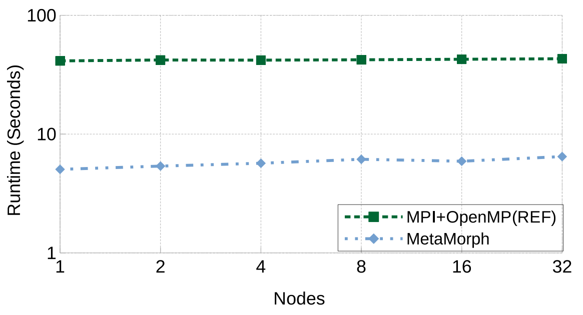


Fig. 10: Performance of the 3D dot-product benchmark on the different MetaMorph (MM) backends with platform-specific BLAS libraries on the experimental cluster

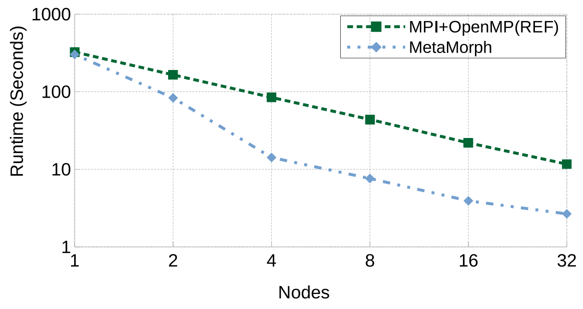
partial differential equations of the material state variables. On multicore CPUs, due to their limited memory bandwidth, the stencil and intra-node data transfer consume the majority of execution time. Many-core devices, with their large memory bandwidth, accelerate all kernels. However, data marshaling suffers from performance degradation on Intel MIC. The profiling data shows that data marshaling kernels do not utilize the vector units on Intel MIC efficiently, due to the non-unit stride memory access pattern and complex control flow. In addition, the hardware prefetcher in Intel MIC is less powerful than mainstream multicore CPUs, when the memory access stream is non-contiguous (scattered) [14].

Figure 10 shows the performance of the 3D dot-product benchmark, when accelerated using MetaMorph with platform-specific BLAS libraries. The results show that the MetaMorph variant (Figure 8a) has comparable performance to the MetaMorph with BLAS libraries version on multicore CPU, AMD GPU and NVIDIA GPU, although MetaMorph's 3D dot product is more flexible than the simple contiguous dot product available in BLAS libraries, as it allows the user to perform dot product on an arbitrary sub-region of the 3D grid; this is very useful for CFD applications, where neighbor ghost cells data is often stored contiguously with local data to simplify stencil operations. However, MetaMorph with MKL outperforms MetaMorph only on Intel MIC for small problem sizes (less than 256x256x256). Our hypothesis is that Intel MKL adapts to the different inputs and problem sizes, which is a feature currently in development in MetaMorph.

Figure 11 shows the scalability analysis of MiniGhost on the large-scale CPU-GPU cluster. Weak scaling experiments use a problem size of 512x512x512 per node, and strong scaling experiments use a global grid of size 1024x1024x1024. In comparison with the reference MPI+OpenMP implementation, MetaMorph achieves 7-8x speedup in the weak scaling problem. The main reason is that MetaMorph effectively utilizes all the available accelerators (CPUs and GPUs) within a node and across nodes. Unlike the traditional MPI+OpenMP approach, the MetaMorph version transparently maps the workload to accelerators from different vendors with different execution models and programming approaches. The workload is distributed based on the relative performance of the accelerators and the availability of on-device memory. In the strong scaling problem, the performance gap between MetaMorph and MPI+OpenMP decreases at lower node counts, as most of the workload is mapped to the host, due to the limited GPU memory. However, at larger node counts, MetaMorph achieves up to 6x speedup over the MPI+OpenMP implementation.



(a) Weak scaling: 512X512X512 local grid per node



(b) Strong scaling: 1024X1024X1024 global grid

Fig. 11: Scalability analysis of MiniGhost on the HokieSpeed cluster with MetaMorph vs. the reference MPI+OpenMP implementation

V. Related Work

With the end of frequency scaling and the shift to parallel and heterogeneous computing, developing software has become much more complex [15]. Many approaches have been proposed to address the challenges of programming parallel architectures by abstracting the hardware details. Here we discuss three related approaches: directive-based programming, domain-specific libraries, and portable run-time systems. MetaMorph is orthogonal to these approaches and subsumes both directive-based programming models and portable run-time systems in the back-end layer, while supporting interoperability with domain-specific libraries and existing accelerated code.

Directive-based programming models, such as OpenMP [16] and OpenACC [17], move the burden of explicit thread management, workload partitioning and scheduling, data movement across the memory hierarchy, and inter-thread synchronization/communication to the compiler. While OpenMP and OpenACC abstract away complex details, and provide a convenient interface to describe parallelism to the compiler, achieving acceptable performance requires deep understanding of the underlying architecture, run-time system, compiler limitations, and a number of complex clause specifications. Moreover, the programmer must use thread-safe functions, eliminate inter-thread data dependencies, avoid pointer aliasing, and manage access to shared variables. In addition, the high-level abstraction of directive-based programming can come with a performance penalty in comparison with low-level programming models such as OpenCL and CUDA [18], [19], [20], [21].

Domain-specific libraries, such as MAGMA, PARALUTION and ViennaCL, provide both abstraction and high performance for a set of computation kernels and algorithms in a specific domain. However, this often comes with the cost of complicated installations and extensive application refactoring. MAGMA [5] provides powerful intelligently scheduled BLAS and LAPACK algorithms, but due to the dependency on external libraries is difficult to install, configure, and tune, and does not yet provide unified or consistent capability across its CUDA, OpenCL, and Intel MIC implementations. Although MAGMA supports multi-GPU BLAS kernels, there is no built-in support for interoperability across different hardware accelerators, e.g. AMD GPU, NVIDIA GPU and Intel MIC. PARALUTION [22] and ViennaCL [23] provide iterative solvers and preconditioners that supports CPUs, GPUs, and Intel MICs. Although PARALUTION and ViennaCL are powerful solver frameworks, they require recasting the application to use complex cases and object types, which present a barrier to incremental porting and adaption.

Portable run-time systems, such as OpenCL [24] and OCCA [25], provide a kernel specification framework and run-time compilation and execution on multiple platforms. Although OpenCL and OCCA support functional portability, performance portability is not guaranteed, and the application developer need to modify the kernel implementation to achieve the required performance on the target hardware platform. Moreover, these approaches have relatively lower programmability in comparison with directive-based programming and domain-specific libraries, as the programmer must explicitly manage all hardware control operations.

In summary, directive-based programming models trade performance with high-level abstraction, while portable run-time systems can achieve high performance with the cost of low programmability and explicit hardware control. Domain-specific libraries can achieve high-level abstraction and high performance, but only for a set of algorithms in a specific domain and with a significant application re-factoring, complicated installations and steep learning curve.

On the other hand, MetaMorph addresses the challenges of programming heterogeneous architectures through its core design principles: abstraction, interoperability and adaptability. MetaMorph provides high performance and abstraction using highly-optimized back-end layer, and light-weight interface layer that does not require significant application re-factoring. In addition, it supports interoperability across different hardware accelerators and with existing code. Moreover, MetaMorph is designed to eventually exploit run-time information to improve computation and data transfer latency even more.

VI. Conclusion

In this paper, we introduced MetaMorph, a library framework designed to simplify the process of extracting high performance from a range of current and future accelerator architectures, without significant time investment in development and learning platform-specific nuances. We showed how the core principles of adaptivity, abstraction, and interoperability are instantiated in our prototype. Further, we demonstrated that through these principles, MetaMorph is able to transparently and efficiently map common communication and computation patterns across several nodes bearing accelerators from different vendors with different programming approaches.

The results show that while MetaMorph has comparable programmability and productivity to directive-based programming models, it provides performance and interoperability across an array of heterogeneous devices, including multicore CPUs, Intel MICs, AMD GPUs, and NVIDIA GPUs. In addition, high performance similar to domain- and accelerator-specific approaches is achievable through the MetaMorph library. Further, by effectively utilizing all the available accelerators within a compute node and across compute nodes, MetaMorph achieves an order of magnitude scalable speedup over the traditional MPI+OpenMP approach.

There remain many opportunities to expand on the MetaMorph prototype, some of which are already in the pipeline. For example, the unified accelerator meta-platform provided by MetaMorph is a perfect candidate for a run-time scheduling system capable of intelligently mapping code across all available accelerator back-ends, providing further abstraction of device-specific nuances and increasing performance. The communication abstractions provided by MetaMorph can be expanded to take more advantage of accelerator-aware MPI implementations, providing an opportunity for communication pipelining, DMA-based transfers, and other optimizations. Finally, we plan to expand MetaMorph's reach to other accelerator devices — such as FPGAs and DSPs— as well as other application domains.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and constructive feedback. This work was supported in part by the Air Force Office of Scientific Research (AFOSR) Computational Mathematics Program via Grant No. FA9550-12-1-0442.

References

[1] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, "Top 500 supercomputers," 2016. [Online]. Available: http://www.top500.org

[2] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

[3] AMD, "Accelerated Parallel Processing Math Libraries (APPML)," Oct. 2015. [Online]. Available: http://developer.amd.com/tools/heterogeneous-computing

[4] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel Math Kernel Library," in *High-Performance Computing on the Intel® Xeon Phi*. Springer, 2014, pp. 167–188.

[5] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki, "Accelerating numerical dense linear algebra calculations with GPUs," in *Numerical Computations with GPUs*. Springer, 2014, pp. 3–28.

[6] NVIDIA, CUDA, "CuBLAS library," *NVIDIA Corporation, Santa Clara, California*, vol. 15, 2008.

[7] G. Martinez, M. Gardner, and W.-c. Feng, "Cu2cl: A cuda-to-opencl translator for multi-and many-core architectures," in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*. IEEE, 2011, pp. 300–307.

[8] P. Sathre, M. Gardner, and W.-c. Feng, "Lost in translation: Challenges in automating cuda-to-opencl translation," in *2012 41st International Conference on Parallel Processing Workshops*. IEEE, 2012, pp. 89–96.

[9] A. Aji, L. Panwar, F. Ji, K. Murthy, M. Chabbi, P. Balaji, K. Bisset, J. Dinan, W. Feng, J. Mellor-Crummey, X. Ma, and R. Thakur, "MPI-ACC: Accelerator-Aware MPI for Scientific Applications," *Parallel and Distributed Systems, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

[10] T. Scogland, W. Feng, B. Rountree, and B. de Supinski, "CoreTSAR: Core Task-Size Adapting Runtime," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, no. 11, pp. 2970–2983, Nov 2015.

[11] R. F. Barrett, C. T. Vaughan, and M. A. Heroux, "Minighost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing."

[12] R. F. Barrett, S. D. Hammond, C. T. Vaughan, D. W. Doerfler, M. A. Heroux, J. P. Luitjens, and D. Roweth, "Navigating an evolutionary fast path to exascale," in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, ser. SCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 355–365. [Online]. Available: http://dx.doi.org/10.1109/SC.Companion.2012.55

[13] E. S. Hertel, Jr., R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. Mcglaun, S. V. Petney, S. A. Silling, P. A. Taylor, and L. Yarrington, "Cth: A software family for multi-dimensional shock physics analysis," in *in Proceedings of the 19th International Symposium on Shock Waves, held at*, 1993, pp. 377–382.

[14] J. Fang, H. Sips, L. Zhang, C. Xu, Y. Che, and A. L. Varbanescu, "Test-driving intel xeon phi," in *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '14. New York, NY, USA: ACM, 2014, pp. 137–148. [Online]. Available: http://doi.acm.org/10.1145/2568088.2576799

[15] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobbs journal*, vol. 30, no. 3, pp. 202–210, 2005.

[16] OpenMP, ARB, "Openmp 4.0 specification," May 2013.

[17] OpenACC Working Group and others, "The OpenACC Application Programming Interface," 2011.

[18] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACCfirst experiences with real-world applications," in *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 859–870.

[19] W. Wang, L. Xu, J. Cavazos, H. H. Huang, and M. Kay, "Fast acceleration of 2D wave propagation simulations using modern computational accelerators," *PloS one*, vol. 9, no. 1, 2014.

[20] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki, "Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, May 2013, pp. 136–143.

[21] M. Daga, Z. S. Tschirhart, and C. Freitag, "Exploring parallel programming models for heterogeneous computing systems," in *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, Oct 2015, pp. 98–107.

[22] D. Lukarski, "PARALUTION project," Oct. 2015. [Online]. Available: http://www.paralution.com

[23] K. Rupp, F. Rudolf, and J. Weinbub, "ViennaCL-a high level linear algebra library for GPUs and multi-core CPUs," *Proc. GPUScA*, pp. 51–56, 2010.

[24] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

[25] D. S. Medina, A. St.-Cyr, and T. Warburton, "OCCA: A unified approach to multi-threading languages," *CoRR*, vol. abs/1403.0968, 2014. [Online]. Available: http://arxiv.org/abs/1403.0968