IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 4 December 2023, accepted 7 February 2024, date of publication 9 February 2024, date of current version 16 February 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3364672

RESEARCH ARTICLE

OpenH: A Novel Programming Model and API for Developing Portable Parallel Programs on Heterogeneous Hybrid Servers

SIMON FARRELLY, RAVI REDDY MANUMACHU, (Member, IEEE), AND ALEXEY LASTOVETSKY, (Member, IEEE)
School of Computer Science, University College Dublin, Belfield, Dublin 4, D04 C1P1 Ireland

Corresponding author: Ravi Reddy Manumachu (ravi.manumachu@ucd.ie)

This work was supported in part by the Science Foundation Ireland (SFI) through the SFI Frontiers for the Future Program under Grant 20/FFP-P/8683, and in part by the Sustainable Energy Authority of Ireland (SEAI) under Grant 21/RDD/664.

ABSTRACT Heterogeneous nodes composed of a multicore CPU and accelerators are today's norm in high-performance computing (HPC) platforms due to their superior performance and energy efficiency. Tools such as OpenCL and hybrid combinations such as OpenMP plus OpenACC are used for developing portable parallel programs for such nodes. However, these tools have some drawbacks, including a lack of compiler support for nested parallelism, performance portability, automatic heterogeneous workload distribution, user-friendly thread placement, and processor affinity essential to the portable performance of hybrid programs executing on such nodes. In this paper, we propose OpenH, a novel programming model and library API for developing portable parallel programs on heterogeneous hybrid servers composed of a multicore CPU and one or more different types of accelerators. OpenH integrates Pthreads, OpenMP, and OpenACC seamlessly to facilitate the development of hybrid parallel programs. An OpenH hybrid parallel program starts as a single *main* thread, creating a group of Pthreads called *hosting* Pthreads. A hosting Pthread then leads the execution of a software component of the program, either an OpenMP multithreaded component running on the CPU cores or an OpenACC (or OpenMP) component running on one of the accelerators of the server. The OpenH library provides API functions that allow programmers to get the configuration of the executing environment and bind the hosting Pthreads (and hence the execution of components) of the program to the CPU cores of the hybrid server to get the best performance. We illustrate the OpenH programming model and library API using two hybrid parallel applications based on matrix multiplication and 2D fast Fourier transform for the most general case of a hybrid hyperthreaded server comprising p computing devices. Finally, we demonstrate the practical performance and energy consumption of OpenH for the hybrid parallel matrix multiplication application on a server comprising an Intel Icelake multicore CPU and two Nvidia A40 GPUs.

INDEX TERMS Parallel computing, parallel programming, heterogeneous platform, hybrid platform, accelerators, OpenMP, OpenACC, Pthreads.

I. INTRODUCTION

Heterogeneous nodes/servers featuring multicore CPU processors hosting multiple accelerators (GPUs and FPGAs) dominate the computing landscape due to their superior performance and energy efficiency. Such nodes compose the supercomputers topping the TOP500 [1] and Green500 [2] lists and delivering exascale computing.

The software development tools and programming models employed to develop parallel programs on heterogeneous servers can be broadly classified into two categories: *low-level vendor-specific* and *high-level vendor-agnostic*. We use the terms *heterogeneous server* and *hybrid server* interchangeably to refer to a server that contains multicore CPU

The associate editor coordinating the review of this manuscript and approving it for publication was Jie Tang.

processors hosting one or more identical or different types of accelerators. We use the term *hybrid parallel program* to signify a program that divides the workload heterogeneously between the multicore CPU processors and accelerators of a server to provide maximum resource utilization and performance on the server.

Vendor-specific programming tools such as CUDA [3] and ROCm [4] provide low-level APIs to the programmer to fine-tune their parallel programs for performance for Nvidia and AMD GPUs, respectively. These tools are typically combined with OpenMP to develop heterogeneous parallel programs. However, using these tools introduces vendor lock-in, hampering the portability of the programs.

Several high-level vendor-agnostic programming tools are proposed to simplify heterogeneous programming by providing high-level abstractions that automate handling low-level details and architecture-specific optimization without significantly sacrificing performance. The most notable research works in this category employ directive-based, C++-based, and skeleton-based approaches.

The directive-based approaches (OpenMP [5], OpenACC [6]) provide compiler directives that are applied to loops, allowing compilers and runtime systems to automate the tasks of offloading to accelerators and parallelization of loops.

The C++-based approaches (OpenCL [7], SYCL [8], Kokkos [9], OCCA [10], RAJA [11]) allow different heterogeneous devices to be used in a single application using standard C++ programming language.

Finally, the skeleton-based (SkelCL [12], SkePU [13]) approaches are based on the concept of skeletons, which are higher-order functions such as map, reduce, scan, farm, and pipeline that implement a typical pattern of computation and data dependence. The SkelCL and SkePU backends translate the codes using the skeletons into OpenCL kernels and C++ library calls that execute on multicore CPUs and GPUs.

However, while the C++-based tools (OpenCL, SYCL, Kokkos, OCCA and RAJA) provide code portability by adhering to modern C++ standard and providing support for various devices (CPU, GPU, and FPGA), they do not have a dominant mainstream support since they compete with the most popular mainstream solutions for these devices - for example, OpenMP for multicore CPUs and CUDA for Nvidia GPUs.

While directive-based approaches such as OpenMP plus OpenACC are ideally suited for portable hybrid parallel programming, they suffer from limitations such as a lack of compiler support for nested parallelism, heterogeneous workload distribution, and user-friendly thread placement and processor affinity of CPU and accelerator computations. We elaborate on these limitations in Section III.

Research works (Xu et al. [14], Komoda et al. [15], Yan et al. [16], [17], Cho et al. [18], Torres et al. [19], Kale et al. [20]) propose extensions to OpenMP and OpenACC to automate the complex process of distributing the computations and data of parallel loops between CPUs and accelerators. However, these works focus on the homogeneous distribution of loop iterations across multiple GPUs to achieve load balance. In the research works (Xu et al. [14], Komoda et al. [15], Cho et al. [18], Torres et al. [19], Kale et al. [20])), the host CPU is only used for synchronizing the results from the accelerators. Furthermore, the research works above do not consider the binding of the loop iterations to the underlying CPU cores, which is essential to obtaining the best performance.

To summarize, developing portable parallel programs on heterogeneous hybrid servers remains challenging. The existing tools suffer from many limitations, which include a lack of compiler support for nested parallelism, performance portability, automatic heterogeneous workload distribution, user-friendly thread placement and processor affinity that are essential to the portable performance of hybrid programs.

In this paper, we propose OpenH, a novel programming model and library API for developing portable parallel programs on heterogeneous hybrid servers composed of a multicore CPU and one or more different types of accelerators. OpenH integrates Pthreads, OpenMP, and OpenACC seamlessly to develop hybrid parallel programs.

An OpenH hybrid parallel program starts as a single *main* thread, which then creates a group of Pthreads called *hosting* Pthreads. A hosting Pthread then leads the execution of a software component of the program, which is either an OpenMP multithreaded component running on the CPU cores or an OpenACC (or OpenMP) component running on one of the accelerators of the server. The OpenH library provides API functions that allow programmers to get the configuration of the executing environment and bind the hosting Pthreads (and hence the execution of components) of the program to the CPU cores of the hybrid server to get the best performance.

The important feature of OpenH is that it does not compete with mainstream solutions widely supported by the community and vendors. Instead, it relies on these solutions and integrates them. OpenH differs from traditional APIs designed for hybrid parallel programming, such as OpenCL. Unlike OpenH, OpenCL provides a unique self-contained API for programming both CPU and accelerator program components and their integration into a single hybrid application. Thus, OpenCL competes with OpenMP in programming CPU components of the hybrid program and with OpenACC and vendor-specific APIs in programming the accelerator components. In contrast, OpenH uses OpenMP to program CPU components, OpenACC and vendor APIs to program accelerator components, and Pthreads and OpenH-specific API to integrate the CPU and accelerator components into a single hybrid program. This approach allows OpenH to achieve better performance portability as it relies on state-of-the-art mainstream solutions in programming individual devices of the hybrid server.

We illustrate the OpenH programming model and library API using two hybrid parallel applications based on matrix multiplication and 2D fast Fourier transform for the most general case of a hybrid hyperthreaded server comprising *p* computing devices.

We demonstrate the practical performance and energy consumption of OpenH for the hybrid parallel matrix multiplication application on a server comprising an Intel Icelake multicore CPU and two Nvidia A40 GPUs. Specifically, we compare the performance and dynamic energy consumption of OpenH matrix multiplication application against matrix multiplication application based on OpenMP, employing only the multicore CPU and matrix multiplication application based on OpenACC and employing only the two Nvidia A40 GPUs. The OpenH matrix multiplication application outperforms the other applications in terms of performance. Furthermore, the OpenH application, when optimized for dynamic energy, performs the best and consumes the least dynamic energy by employing the most energy-efficient processor (multicore CPU in this case).

The main original contributions of this work are:

- A comprehensive survey of foundational tools for developing portable parallel programs on modern heterogeneous hybrid HPC nodes/servers.
- A comprehensive overview of issues and challenges in developing portable parallel programs on modern heterogeneous hybrid HPC nodes/servers using OpenMP and OpenACC.
- A novel programming model for developing portable parallel programs on heterogeneous hybrid servers.
- A library API for programmers to get the configuration of the executing environment and bind the hosting Pthreads (and hence the execution of components) of the program to the CPU cores of the hybrid platform to get the best performance.
- Experiments comparing the practical performance and energy consumption of an OpenH hybrid matrix multiplication application with an OpenMP and OpenACC matrix multiplication application, respectively, employing CPU only and two Nvidia A40 GPUs only.

The rest of the paper is organized as follows. Section II presents foundational tools for portable parallel programming on modern HPC servers. Section III describes the design issues in employing OpenMP and OpenACC in synergy to develop a programming model for portable parallel programs on heterogeneous hybrid servers. Then, Section IV illustrates the OpenH programming model and library API for developing portable parallel programs on heterogeneous hybrid servers. The following Section V describes two hybrid parallel applications, matrix multiplication and 2D fast Fourier transform, developed using OpenH and executing on a hybrid server comprising *p* computing devices. Section VI contains the experimental results for the hybrid parallel matrix multiplication application on a hybrid server comprising a multicore CPU and two Nvidia A40 GPUs. Section VII presents the related work. Finally, we conclude the paper in Section VIII.

II. FOUNDATIONAL TOOLS FOR PORTABLE PARALLEL PROGRAMMING

This section describes the foundational tools for developing portable parallel programs on modern heterogeneous hybrid HPC nodes/servers. We will evaluate how mature these tools are to develop portable parallel programs on platforms comprising multicore CPUs and one or more accelerators belonging to different vendors.

A. PTHREADS

Pthreads, or Portable Operating System Interface (POSIX) threads, have been a popular choice for implementing multithreading in C and C++ programs since their introduction in 1995. Pthreads provide a standard Unix interface for creating and manipulating threads, making it easier for developers to write portable multithreaded programs across different Unix-like operating systems.

One of the primary advantages of using Pthreads is the level of control they offer to the developer. Pthreads allow developers to specify the number of threads, the scheduling policy, and the synchronization mechanisms used to coordinate access to shared resources. This level of control can lead to more efficient use of system resources and better performance.

However, this level of control also comes with a cost. Pthreads are relatively complex, requiring developers to manage issues such as race conditions, deadlocks, and thread synchronization. This complexity can make writing correct and reliable multithreaded programs more complicated.

Another potential disadvantage of Pthreads is their portability. While Pthreads provide a standardized interface, there are still differences between implementations on different operating systems. Therefore, it can lead to subtle bugs and performance differences that may take time to be apparent. Additionally, Pthreads are only available on Unix-like operating systems, limiting their portability to other platforms.

Given the complexity mentioned above of Pthreads, it would be advantageous to leverage higher-level abstractions where feasible. Such abstractions can offer a more intuitive programming model and automated handling of low-level details, reducing the likelihood of errors and simplifying development. Furthermore, Pthreads do not have any API to allow them to run on accelerators, restricting any programming model relying solely on them to utilize CPUs only.

B. OPENMP

OpenMP is a popular high-level threading API for C, C++, and Fortran that aims to simplify the implementation of parallelism in code. Like Pthreads, OpenMP can parallelize compute-intensive applications by distributing the work across multiple threads. However, there are some notable differences between the two approaches.

One of the primary benefits of OpenMP is its ease of use. OpenMP uses a simple pragma-based syntax that allows developers to specify parallel regions, loop parallelization, and task-based parallelism with minimal modifications to existing code. This simplicity can make writing and maintaining parallel code easier, especially for those less familiar with low-level threading constructs.

Another advantage of OpenMP is its portability. OpenMP is available on various platforms and operating systems, including Windows and Linux, making it a versatile option for parallelization. Furthermore, newer versions of OpenMP, such as OpenMP 4.5 and 5.0, allow code offloading to accelerators like GPUs, providing additional performance and flexibility to developers.

However, as with Pthreads, OpenMP has some potential drawbacks. One issue is that the ease of use provided by OpenMP comes at the cost of some degree of control over the parallelism. Unlike Pthreads, OpenMP provides limited control over scheduling and synchronization, which can limit the fine-tuning of parallel code. Additionally, while OpenMP is portable across different platforms, performance can vary between implementations, and developers may need to consider implementation-specific optimizations.

Nonetheless, given the simplicity and portability offered by OpenMP, it may be an excellent option for handling parallel processing tasks executed on the CPU. Moreover, with its recent support for offloading code to accelerators, OpenMP presents a promising tool for handling parallel code on various computing architectures.

While OpenMP offers the advantage of portability and simplicity, it still needs to address the challenge of developing hybrid portable code. Although OpenMP allows for the offloading of code to accelerators like GPUs, its inherent initial focus on running parallel code on CPUs and the loss mentioned above of control present obstacles for effective hybrid programming. Specifically, ensuring parallel execution of CPU kernels and the hosting threads of the GPU kernels on disjoint groups of CPU cores becomes a critical concern in leveraging OpenMP as the sole solution for hybrid programming. It will become more apparent once we discuss OpenMP's approach to affinity, "places" and their shortcomings in this area, likely stemming from their design when the model focused solely on CPU parallelism.

C. OPENACC

OpenACC is a popular parallel programming model for accelerating compute-intensive tasks on accelerator architectures, such as GPUs. It is designed to simplify the implementation of parallelism in code written in C, C++, and Fortran, and it offers an alternative to tools such as CUDA. In the OpenACC programming model, a hosting thread running on a CPU core manages and coordinates the execution of a kernel on an accelerator. This approach allows for a division of labour between the CPU and the accelerator, where the CPU handles the control flow and manages data transfers. In contrast, the accelerator focuses on computationally intensive kernel execution.

One of the primary benefits of OpenACC is its ease of use. OpenACC employs a set of directives embedded in the code to specify parallelism. These directives are similar to OpenMP pragmas and can be easily added to existing code with minimal modification. Therefore, it makes it easier for developers to write and maintain parallel code, especially those less familiar with low-level parallelization constructs.

Another advantage of OpenACC is its portability. OpenACC is designed to be platform-independent, meaning that code written using OpenACC can run on various computing architectures, including CPUs, GPUs, and other accelerators. This versatility is advantageous for developers who require flexibility in deployment and want to avoid vendor lock-in that can sometimes be associated with proprietary tools like CUDA.

While OpenACC shares many similarities with CUDA, there are some notable differences between the two approaches. One of the primary differences is that OpenACC is a high-level programming model that provides more abstraction and easier-to-use directives than CUDA. Hence, it can be advantageous for developers who require a more straightforward and flexible approach to parallel programming. Still, it may come at the cost of some degree of control over the parallelism. Additionally, OpenACC may be less performant than CUDA in certain use cases, such as those requiring fine-tuned control over the parallelism.

Considering the benefits above, OpenACC is a promising component within a comprehensive toolkit for developing portable and user-friendly heterogeneous code. Nevertheless, it is essential to note that OpenACC does not inherently utilize idle CPU cores for computational tasks beyond running hosting threads. Consequently, to achieve a hybrid programming model that effectively employs both CPU and accelerator resources, OpenACC needs to be supplemented by complementary tools such as Pthreads or OpenMP.

D. CUDA

CUDA is a parallel programming model designed to accelerate compute-intensive tasks on Nvidia GPUs. The core of CUDA is based on three abstractions exposed to the programmer: a hierarchy of thread groups, shared memories and barrier synchronization. They allow the programmer to partition the problem into coarse sub-problems solved by thread groups in parallel, and each sub-problem is further partitioned into chunks that are solved by threads within a thread group.

One of the primary benefits of CUDA is its level of control. With CUDA, developers have direct access to the GPU hardware, allowing them to fine-tune the parallelism and optimize performance for their specific application. CUDA also provides a comprehensive set of libraries for common mathematical operations, making it easier to write high-performance code without sacrificing control. Because it provides low-level control over the hardware, CUDA can achieve very high levels of parallelism and throughput, which can be beneficial in applications that require intensive calculations.

However, one of the main drawbacks of CUDA is its lack of portability. CUDA is tightly coupled with NVIDIA GPUs and is not compatible with other types of hardware. It can limit the flexibility of developers who require heterogeneous computing solutions or want to avoid vendor lock-in.

Another notable disadvantage of CUDA is its complexity. CUDA requires developers to have a deep understanding of parallel programming concepts, GPU architecture, and memory management. It can make it challenging for those less familiar with low-level parallelization constructs to write and maintain parallel code. Furthermore, debugging CUDA applications can be challenging due to the complexity of the underlying hardware and the need for comprehensive debugging tools. As a result, CUDA may not be the best choice for developers who prioritize ease of use and maintainability over raw performance.

Finally, CUDA would need to be paired up with another tool, such as OpenMP or Pthreads, to utilize the host CPU in a hybrid application.

E. OPENCL

OpenCL (Open Computing Language) [7] is an open standard designed to support portable parallel application development on heterogeneous platforms comprising multicore CPUs and accelerators (such as GPUs and FPGAs).

OpenCL's platform model represents the topology of a platform by a host CPU connected to one or more OpenCL compute devices.

It contains two APIs: platform layer API and runtime API. The platform layer API functions run on the host CPU and allow users to detect the available computing devices. The runtime API enables the application's kernel programs to be compiled, loaded and executed on the computing devices in parallel.

Like CUDA, it utilizes a low-level approach to developing code for the accelerator. Therefore, it requires the user to have a high level of understanding and a tailored implementation for the architecture the code is targeting.

The OpenCL's uniform programming environment provides maximum code portability for its heterogeneous applications composed of software components executing parallelly on compute devices. However, it has yet to hold dominant mainstream support since it competes with other mainstream solutions widely used in each software component category. For example, OpenCL competes with OpenMP, the most popular programming tool for developing software components for multicore CPUs, with CUDA and OpenACC for accelerator software components for Nvidia GPUs.

F. MPI

MPI (Message Passing Interface) is the de facto standard and library for developing parallel programs for distributed memory systems. In the MPI programming model, processes with separate address spaces execute in parallel and communicate using messages.

MPI is usually combined with one of the tools above to develop parallel programs for heterogeneous clusters where MPI is used between processes across nodes and a shared memory tool inside a process within a node.

G. SUMMARY

OpenMP and OpenACC are promising tools for developing portable parallel programs on heterogeneous hybrid server platforms. However, a critical concern in leveraging OpenMP as the sole solution for hybrid programming is ensuring parallel execution of CPU kernels and the hosting threads of the GPU kernels on disjoint groups of CPU cores. The lack of facility to address this concern will become more apparent once we discuss OpenMP's approach to affinity.

OpenACC does not inherently utilize idle CPU cores for computational tasks beyond running hosting threads. Consequently, to achieve a hybrid programming model that effectively employs both CPU and accelerator resources, OpenACC needs to be supplemented by complementary tools such as Pthreads or OpenMP.

Therefore, in their current state, none of these tools can facilitate the development of portable parallel programs on heterogeneous hybrid server platforms.

III. PORTABLE PARALLEL PROGRAMS ON HETEROGENEOUS HYBRID SERVERS USING OPENMP+OPENACC: DESIGN ISSUES

This section describes our initial attempt at leveraging the widely adopted high level programming models, OpenMP and OpenACC, in synergy to develop a programming model for developing portable parallel programs on heterogeneous hybrid servers.

Leveraging OpenMP and OpenACC in synergy is deemed favorable due to OpenMP's widespread usage in parallel CPU code and OpenACC's capability to offload computations to accelerators, complemented by its relatively advanced stage of development for GPU offloading compared to OpenMP. By adopting a directive-based paradigm, the proposed model aimed to alleviate the burden of system intricacies from developers, enabling them to focus primarily on the logic of the code and to ensure system agnosticism of the code-base.

We discuss the limitations and challenges that surfaced during the attempt, prompting the identification of areas that require improvement. In response, we design and implement a novel programming model, presented in the next section, to empower developers with greater control over addressing these challenges when crafting hybrid code.

A. OPENMP AND OPENACC COMPILERS

Our first goal is to find compilers that support both OpenMP and OpenACC. Several compilers implement the OpenMP specification [21]. However, the options for OpenACC are much more limited.

Among the compilers supporting OpenMP, Nvidia's NVC compiler [22] and GCC [23], the GNU compiler collection, support OpenACC for Nvidia GPUs. GCC supports OpenACC for Nvidia through a tool called nvptx-tools and also supports AMD Radeon GPUs [24]. However, both compilers have issues compiling hybrid codes using OpenMP and OpenACC.

GCC currently does not allow users to compile projects targeting both OpenMP and OpenACC. A workaround is for GCC to utilise the newer OpenMP specification [5] to offload GPU work, removing OpenACC from the process. However, this feature is not yet fully functional on the platforms we tested.

The issue with the NVC compiler is that it fails to compile more complex code that employs nested parallelism supported in OpenMP. Therefore, unlike GCC, the NVC compiler has no integrated support for nested parallel regions in OpenMP, despite being supported in the OpenMP specification. Nevertheless, a workaround can be employed for NVC, utilising Pthreads to replace the top-level parallelism in the code. This workaround would marginally increase complexity and decrease portability (limiting it to POSIX compliant systems).

In summary, we choose the NVC compiler due to its OpenACC implementation's maturity over the newer accelerator offloading specifications of OpenMP. However, it would be ideal to use an open-source compiler like GCC when it can better support hybrid codes developed using OpenMP and OpenACC.

B. PROCESSOR AFFINITY CHALLENGES

This section describes the processor binding and thread affinity features provided by OpenMP. High-level programming tools such as OpenMP often abstract and limit this affinity configuration, likely in an attempt to reduce complexities. Whilst these abstractions are often reasonable when using the API in isolation, once a developer looks to combine multiple APIs, their implied benefits quickly become obstructions. As such, any hybrid model will need to provide developers with much more control over the final configuration of where hosting and worker threads are located on the CPU. Note that OpenACC has no support for processor binding and thread affinity.

1) OPENMP PROCESSOR BINDING AND THREAD AFFINITY

OpenMP 4.0 provides two environment variables, *OMP_PLACES* and *OMP_PROC_BIND*, to specify how the OpenMP threads are bound to logical cores on the machine. The logical cores will be the same as physical CPU cores if hyperthreading is absent.

In addition to the two environment variables, OMP_PLACES and OMP_PROC_BIND, OpenMP 4.0 provides the *proc_bind* clause, which can appear on a parallel pragma directive. The *proc_bind* clause specifies how the team of threads executing the parallel region are bound to processors. The value set in OMP_PROC_BIND environment variable is a global setting and applies to all the parallel teams launched in the OpenMP program. However, the global setting can be overridden locally using the *proc_bind* clause.

The OpenMP approach (using environment variables and options in pragma parallel directive) is different from multi-threaded applications written using Pthreads that make use of POSIX/Unix calls (*pthread_setaffinity_np*, *sched_setaffinity*), to pin the thread to a specific logical core.

OMP_PLACES specifies the places on the machine to which the threads are bound and is called a *partition* (of the physical/logical set of core IDs). OMP_PROC_BIND specifies the binding policy that prescribes how the threads are assigned to places. Therefore, OMP_PROC_BIND must be used in conjunction with OMP_PLACES to enable binding.

The OMP_PLACES environment variable allows two types of settings, high-level and low-level. In the high-level version, OMP_PLACES permits three values: *threads, cores,* or *sockets*. When set to *threads*, each place corresponds to a single hardware thread. When set to *cores*, each place corresponds to a single core on the target machine. Finally, when set to *sockets*, each place corresponds to a single socket. If OMP_PLACES is not set, the default value is *cores*.

For example, the setting, *export OMP_PLACES="threads (48)"*, specifies the number of hardware threads for possible places (including hyper-threads). The number of places in parentheses is optional. The setting, *export OMP_PLACES="cores(24)"*, specifies the number of cores for possible places (each core may have a certain number of hardware threads). The setting, *export OMP_PLACES="sockets(2)"*, specifies the number of sockets for possible places where each socket may consist of a certain number of cores.

The low-level version allows the user to specify an explicit list of places described by non-negative numbers. A place is defined by an unordered set of comma-separated non-negative numbers enclosed by braces. Generally, the numbers represent the smallest unit of execution exposed by the execution environment, typically a hardware thread. For example, the setting, *export OMP_PLACES="{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"*, specifies four places. The first place includes CPU cores, {0,1,2,3}. The second place, {4,5,6,7}, and so on. Intervals may also be used to define places. For example, the settings, *export OMP_PLACES="{0:4},{4:4},{8:4},{12:4}"* and *export OMP_PLACES="{0:4}:4:4"*, are the same as the setting above.

The *OMP_PROC_BIND* environment variable provides fine-grained control of how threads are bound and distributed to places and whether OpenMP threads can be moved between places.

When set to *true*, the thread affinity is enabled, and the threads are bound to places specified in OMP_PLACES environment variable. When set to *false*, thread affinity is disabled, and threads are allowed to move between places.

The value *master* assigns the threads in the team to the same place as the master thread. The *master thread* in OpenMP is an OpenMP thread with thread number 0. A *parent thread* is the thread that encounters the *parallel* construct and generated a parallel region. It is the parent thread of each of the threads in the team of that parallel region. The master thread of a parallel region is the same as its parent thread.

Consider the following example:

```
export OMP_NUM_THREADS=4;
export OMP_PROC_BIND=master;
export OMP_PLACES='{0:4},{4:4},{8:4},
{12:4},{16:4},{20:4},{24:4},{28:4}'
```

The four OpenMP threads defined by the OMP_NUM_THREADS environment variable are assigned to place 0. Place 0 is the list, {0, 1, 2, 3} of core IDs.

The value *close* assigns the threads in the team to places close to the place of the parent thread. Suppose T is the number of threads in the team, and P is the number of places in the parent's place partition. If $T \leq P$, the master thread (with ID 0) executes in the place of the parent thread. Thread 1 executes on the next place in the place partition, and so on.

If $T > P$, each place is assigned at least $S = \lfloor \frac{T}{P} \rfloor$ consecutive threads. The first S threads, including the master thread, are assigned to the place of the parent thread. The following S threads are assigned to the next place in the place partition, and so on. The wrap-around is to the place partition of the master thread.

Consider the following example:

```
export OMP_NUM_THREADS=4;
export OMP_PROC_BIND=close;
export OMP_PLACES='{0:4},{4:4},{8:4},
{12:4},{16:4},{20:4},{24:4},{28:4}'
```

The value of T and P are 4 and 8, respectively. The 8 places are, {0,1,2,3}, …, {28,29,30,31}. The OpenMP thread 0 is assigned to place 0, thread 1 to place 1, and so on.

The value *spread* distributes a set of T threads as evenly as possible among P places of the parent thread's place partition. If $T \leq P$, the parent thread place partition is divided into T subpartitions, each containing at least $S = \lfloor \frac{T}{P} \rfloor$ consecutive places. A single thread is assigned to each subpartition. The master thread executes in the place of the parent thread and is assigned to the subpartition that includes that place. Thread 1 is assigned to the first place in the next subpartition, and so on.

Consider the following example:

```
export OMP_NUM_THREADS=4;
export OMP_PROC_BIND=spread;
export OMP_PLACES='{0:4},{4:4},{8:4},
{12:4},{16:4},{20:4},{24:4},{28:4}'
```

Since the number of threads ($T = 4$) is less than the number of places $P = 8$, four subpartitions are formed, each containing two places. The thread assignment is as follows: OpenMP thread 0 is assigned to place 0, thread 1 is assigned to place 2, thread 2 is assigned to place 4, and thread 3 is assigned to place 6.

If $T > P$, the parent thread's place partition is divided into P subpartitions, each corresponding to a single place. Then, each place contains at least $S = \lfloor \frac{T}{P} \rfloor$ consecutive threads. The first S threads with the smallest thread number (including the master thread) are assigned to the subpartition that contains the place of the parent thread. The subsequent S threads with the following smallest thread numbers are assigned to the next place in the place partition, and so on.

Consider the following example:

```
export OMP_NUM_THREADS=8;
export OMP_PROC_BIND=spread;
export OMP_PLACES='{0:4},{4:4},
{8:4},{12:4}'
```

Since the number of threads ($T = 8$) is greater than the number of places ($P = 4$), four subpartitions are formed, each corresponding to a single place and containing two threads. The thread assignment is as follows: OpenMP threads 0 and 1 to place 0, threads 2 and 3 to place 1, and so on.

2) OPENMP NESTED PARALLELISM FOR HYBRID PARALLEL APPLICATION EXECUTION

The Nested Parallelism feature in OpenMP is essential to developing hybrid parallel applications. To understand why, we present here the anatomy of a hybrid parallel application.

A hybrid parallel application is composed of several software components (kernels) executing in parallel. There is a one-to-one mapping between the components and computing devices of the hybrid platform on which the application is executed. The execution of an accelerator component involves a dedicated CPU core, running the hosting thread, and the accelerator itself, performing the accelerator code. The execution of the accelerator component includes data transfer between the CPU and accelerator memory, computations by the accelerator code, and data transfer between the accelerator memory and CPU. The execution of a CPU component only involves the CPU cores performing the multithreaded CPU code.

Therefore, a hybrid parallel application developed using OpenMP will have two levels of nesting. At the top level, a team of threads is created whose size is equal to the number of software components in the application. At the second level, each thread from the top level will create a team of threads to execute the software component. There will be a one-to-one mapping between the teams and the software components. All the teams will now execute the components in parallel. Moreover, note that different teams will have different numbers of threads due to the nature of the component decomposition of the hybrid parallel application.

Furthermore, the software components must be mapped to the physical/logical cores considering the affinity of the accelerators to the CPU cores. *Therefore, the programmer*

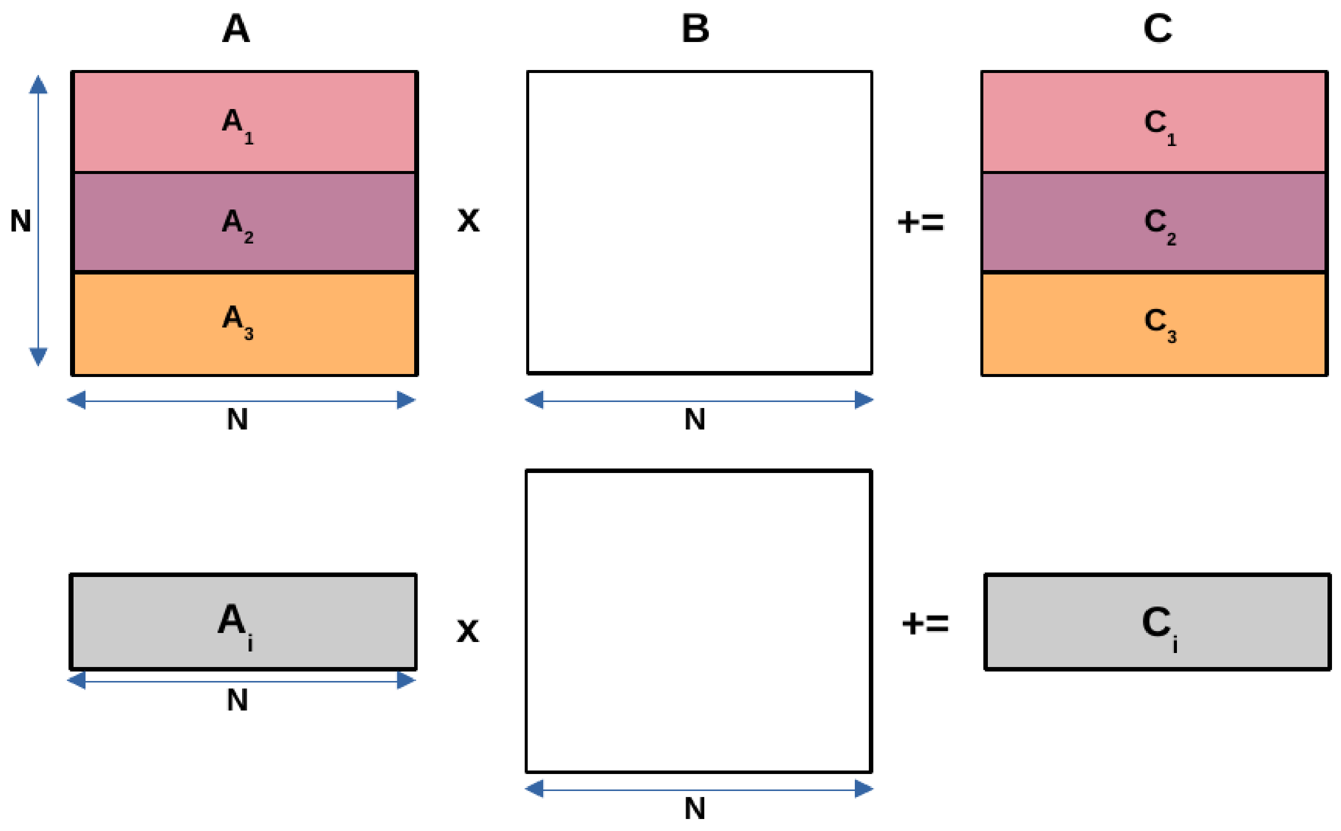


FIGURE 1. **A hybrid parallel matrix multiplication application computing the matrix product ($C+=A\times B$) of two dense square matrices A and B of size $N\times N$. It is executed on a hybrid server comprising a multicore CPU, an Nvidia k40c GPU, and an Intel Xeon Phi 3120P. The application comprises 3 software components (one CPU component, one GPU component, and one Xeon Phi component) executed in parallel. The matrix B is shared by all the teams. Each software component i is assigned a number of rows of A and C proportional to its performance.**

must carefully understand the hardware topology of the server and map the execution of software components to partitions of CPU cores.

We briefly overview the complexity of developing a hybrid parallel matrix multiplication application using the nested parallelism feature of OpenMP. Figure 1 illustrates the application. It computes the matrix product ($C+=A\times B$) of two dense square matrices A and B of size $N\times N$. The application is executed on a hybrid server comprising a dual-socket multicore CPU with each socket containing 24 logical cores, and two accelerators, one Nvidia k40c GPU and an Intel Xeon Phi 3120P.

The hardware topology of the hybrid server and the CPU affinity of the accelerators are shown in Figure 2. The Nvidia k40c GPU in the hybrid server is closest to the cores in NUMA node 1 (12-23,36-47) on this hybrid server.

The application contains three software components executing in parallel, a CPU component and two accelerator components. The execution of the accelerator component involves a dedicated CPU core, running the hosting thread, and the accelerator itself, performing the accelerator code. The CPU component involves the remaining 46 CPU cores executing the multithreaded CPU code. Thus, three OpenMP threads – the master thread of the CPU component and the hosting threads of the accelerator components – are running in parallel, each leading the execution of one software component. The matrices A and C are partitioned horizontally among the three components such that the number of rows of A and C assigned to each component is proportional to its performance.

Figure 3 shows the two principal parts of the application. *The first part involves environment variable settings external to the OpenMP program essential to extracting optimal performance on the hybrid platform.*

While OpenMP allows the setting of *OMP_PROC_BIND* variable for each nesting level, it does not provide a similar system for *OMP_PLACES*. That is, *OMP_PLACES* setting for nested levels is not permitted. Therefore, each child thread will inherit the same list of places (specified in OMP_PLACES) of the parent thread to distribute its child threads if *OMP_PROC_BIND* on the current level is set to *"master"* or *"close"*. If using the *spread* option for *OMP_PROC_BIND* on the current level, the parent thread place partition is divided into subpartitions as previously outlined.

Lines 1-3 contain the environment variable settings. The master or parent thread partition comprises three places specified in OMP_PLACES. The first and second places

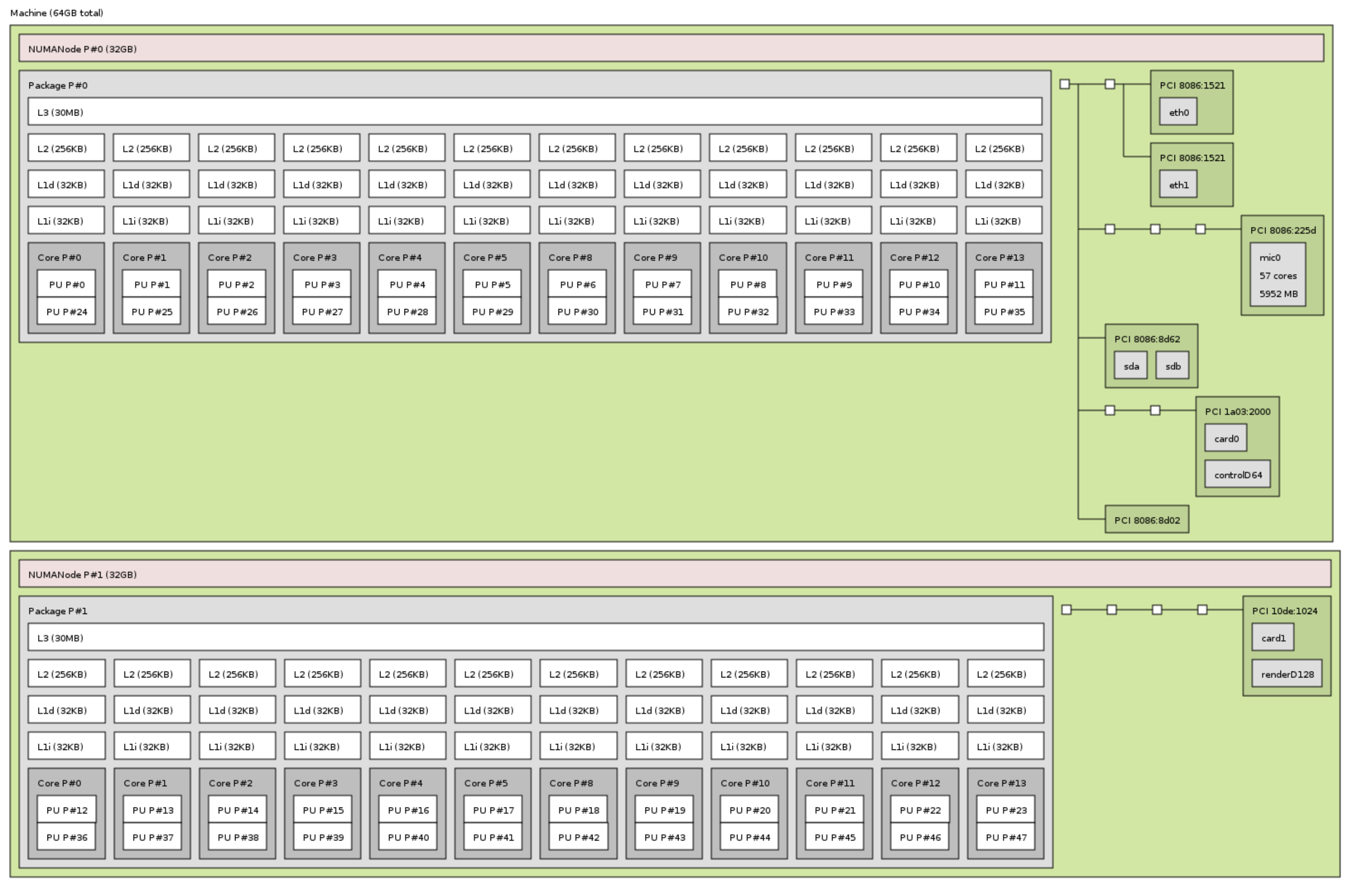


FIGURE 2. The hardware topology of the hybrid server comprising a dual-socket multicore CPU with each socket containing 24 logical cores, and two accelerators, an Nvidia K40c GPU and an Intel Xeon Phi 3120P. The graphic is generated using the *lstopo* tool of the *hwloc* package. While the Intel Xeon Phi card is closest to cores in NUMA node 0, the Nvidia k40c GPU is closest to the cores in NUMA node 1.

represent cores 0 and 12. The third place represents the list of cores, $\{1, \cdots, 11, 13, \cdots, 47\}$.

The second principal part of the application comprises OpenMP and OpenACC pragmas and library functions to execute the CPU and accelerator software components. In Line 5, three team leads are created using *proc_bind* value of *spread*. Therefore, the parent thread partition is divided into three subpartitions, where each subpartition gets a place. So, team lead 0 gets place 0 (core 0), team lead 1 gets place 12 (core 12), and team lead 2 gets the place, $\{1, \cdots, 11, 13, \cdots, 47\}$, which is the set of remaining cores.

Lines 7-20 implement the execution of the Xeon Phi software component with OpenMP thread 0 as its hosting thread leading the execution. The Xeon Phi component invokes Intel MKL dgemm routine to compute the matrix product $C_1 \mathrel{+}= A_1 \times B_1$. The OpenMP thread 1 leads the execution of the GPU software component in Lines 22-33. This software component employs OpenACC to execute the *cublas Dgemm* library routine to compute the matrix product $C_2 \mathrel{+}= A_2 \times B_2$. Finally, the OpenMP thread 2 creates a team of 46 threads bound to CPU cores $\{1, \cdots, 11, 13, \cdots, 47\}$ to execute the CPU component. The CPU component invokes Intel MKL dgemm routine to compute the matrix product $C_3 \mathrel{+}= A_3 \times B_3$.

3) OPENMP NESTED PARALLELISM FOR HYBRID PARALLEL APPLICATION EXECUTION: LIMITATIONS AND CHALLENGES

The hybrid program presented in Figure 3 fully complies with the official OpenMP and OpenACC specifications. It comprises two principal parts. The first part involves environmental variable settings to map the CPU and accelerator components to the hybrid platform to extract optimal performance. The second part contains the execution of the CPU and accelerator components using OpenMP and OpenACC constructs.

However, limitations in the state-of-the-art compilers prohibit the compilation of this program, and challenges include the need for code portability. We will start with limitations before discussing portability.

The first limitation pertains to the lack of compiler support for the program. We will look at two prominent compilers, NVC and GCC. While the NVC compiler allows the compilation of OpenACC constructs to execute the GPU accelerator component, it does not support OpenMP's nested parallelism and execution of the Xeon Phi accelerator

```
export OMP_NESTED='true'
export OMP_PROC_BIND='true'
export OMP_PLACES='{0},{12},{1:11,13:47}'
#pragma omp parallel num_threads(3) proc_bind(spread)
{
    int tlead = omp_get_thread_num();
    if (tlead == 0) {
        // Xeon Phi software component execution
        acc_set_device_num(0, acc_device_xeonphi);
        #pragma acc data copyin(N, N1, A1, B1) copy(C1)
        {
            #pragma acc host_data use_device(A1, B1, C1)
            {
                cblas_dgemm(CblasRowMajor, CblasNoTrans,
                        CblasNoTrans, N1, N, N, 1.0, A1,
                        N1, B1, N, 1.0, C1, N1);
            }
        }
    }

    if (tlead == 1) {
        // GPU software component execution
        acc_set_device_num(1, acc_device_nvidia);
        cublasHandle_t handle;
         cublasStatus_t status = cublasCreate(&handle);
        #pragma acc data copyin(N, N2, A2, B2) copy(C2)
        {
            #pragma acc host_data use_device(A2, B2, C2)
            {
                double alpha = 1.0, beta = 1.0;
                cublasDgemm(handle, CUBLAS_OP_N,
                        CUBLAS_OP_N, N2, N, N,
                        &alpha, A2, N2, B2, N, &beta, C2, N2);
            }
        }
        cublasDestroy(handle);
    }

    if (tlead == 2) {
        // CPU software component execution
        cblas_dgemm(CblasRowMajor, CblasNoTrans,
                    CblasNoTrans, N3, N, N, 1.0,
                    A3, N3, B3, N, 1.0, C3, N3);
    }
}
```

FIGURE 3. A hybrid parallel matrix multiplication application executing on a server comprising a dual-socket multicore CPU of 48 logical cores and two accelerators, an Nvidia k40c GPU, and an Intel Xeon Phi 3120P. The application contains three software components (kernels) executing in parallel, one Intel Xeon Phi component, one GPU component, and one CPU component. The application uses the OpenMP's nested parallelism feature and OpenACC to execute the accelerator components. The team leads 0 and 1 will lead the execution of the Xeon Phi and GPU components on cores 0 and 12, respectively. The team lead 2 will lead the execution of the CPU component on the set of cores, $\{1, \cdots, 11, 13, \cdots, 47\}$. *It is important that the programmer know the affinity of the accelerators to the CPU cores for mapping the software components for optimal performance..*

component. Therefore, it would fail to compile the program. Note that Intel no longer supports the Intel Xeon Phi family of coprocessors. Therefore, we will not consider this family of coprocessors further in this article.

The GCC compiler does not support using OpenMP and OpenACC in a single program. Therefore, it will fail to compile the OpenACC constructs executing the accelerator components. Hence, the programmer must use vendor-specific API to perform the accelerator computations hindering the portability of the code.

Finally, one can rewrite the hybrid program using OpenMP only to execute the CPU and accelerator components since the latest OpenMP specification provides constructs targeting accelerators. However, the OpenMP version of the hybrid program fails to compile on our hybrid servers, highlighting a lack of comprehensive support for accelerators.

We focus on portability challenges now. The programmer must employ the environment variable settings (OMP_PLACES and OMP_PROC_BIND) to map the execution of the program to the underlying CPU cores for optimal performance. The environment variable approach renders the program non-portable to other platforms because the environment variable settings will differ from one platform to another.

Furthermore, the portability of the presented hybrid program is hindered by the fact that the hybrid application developer must understand the hardware topology of the hybrid server and the CPU affinity of the accelerators. To extract the maximum performance, the developers must carefully map the CPU threads of their hybrid programs to partitions of cores. To the author's knowledge, there is no library automating this feature crucial to the optimal performance of hybrid programs.

4) NESTED PARALLELISM PROBLEM: SOLUTION USING PTHREADS AND OPENMP

One can utilize Pthreads to replace the top-level OpenMP threads as the team leads to solving the limitations in the NVC compiler related to nested parallelism.

Consider the execution of the hybrid matrix multiplication application developed using Pthreads, OpenMP, and OpenACC on the same server platform. Three Pthreads will be created at the top level, out of which two Pthreads will lead the execution of the two accelerator components, and one Pthread will lead the execution of the CPU component using the OpenMP parallel loop construct.

To extract optimal performance on the server platform, the Pthreads executing the GPU components must be bound to CPU cores 0 and 1. The Pthread executing the CPU component will need to be bound to the set of CPU cores, $\{2, 3, \cdots, 63\}$.

Hence, it is necessary to explore how mixing thread affinities between the Pthreads and OpenMP worked. To the author's knowledge, there is no documentation for this as it is not a common use case to need both in traditional parallel models. Consequently, we developed several test programs to assess the impact on affinity utilizing both Pthreads and OpenMP.

When creating an OpenMP parallel region, a Pthread parent thread with no set affinity will utilize the *OMP_PLACES* environment variable setting. However, once a Pthread parent thread's affinity is set using *pthread_setaffinity_np()* or *sched_setaffinity()* call, all OpenMP threads inside the parallel region will share the same affinity as the parent thread and ignore the value set in *OMP_PLACES* environment variable.

The summary of challenges when mixing Pthreads and OpenMP in the same program follows:

- The programmer must avoid mixing the Pthread API or Unix system calls, *pthread_setaffinity_np()* or *sched_setaffinity()*, and OpenMP (OMP_PLACES and OMP_PROC_BIND) for processor binding and thread affinity to ensure the correctness and reliability of hybrid parallel applications.
- The programmer must carefully analyze the hardware topology of the server and bind the top-level Pthreads (or team leads) to partitions of cores to execute the software components of the hybrid application in parallel for optimal performance. Moreover, the programmer must consider the additional layer of complexity introduced by hyperthreading.

C. SUMMARY

The development of portable hybrid parallel programs is hampered by limitations in the state-of-the-art NVC and GCC compilers and lack of tools to map the software components of the hybrid program to the CPU cores of the hybrid platform for optimal performance.

When attempting to integrate programming models of OpenMP and OpenACC to develop a hybrid approach, significant challenges arise in effectively controlling the assignment of threads to specific cores. This control is essential for ensuring code efficiency and avoiding scenarios of oversubscription or underutilization. There should be a unified and transparent method of mapping process threads to logical or physical cores as the developer desires.

Furthermore, the utilization of hyperthreading introduces another layer of complexity due to the mapping of logical cores to physical cores within the system. This mapping discrepancy can confuse developers, as it impacts the determination of the available number of physical cores for parallelization and which logical cores to use to maximize efficiency. Addressing these issues becomes imperative to provide developers with the available resources for parallelization efficiently.

In conclusion, the limitations in OpenMP and OpenACC require a solution using Pthreads. Furthermore, there is a need for a tool to empower developers with complete control to bind the hybrid parallel program's CPU threads to the hybrid platform's CPU cores.

IV. OPENH: A NOVEL PROGRAMMING MODEL FOR DEVELOPING PORTABLE PARALLEL PROGRAMS ON HETEROGENEOUS HYBRID SERVERS

We propose *OpenH*, a novel programming model that enables users to seamlessly combine Pthreads, OpenMP, and OpenACC and a library API to address the portability challenges mentioned above to aid the development of portable hybrid parallel programs.

The OpenH library contains critical missing features in the current models to allow developers to write portable hybrid parallel programs. It provides API functions for obtaining the configuration of the executing environment and binding the hybrid program's software components to the CPU cores to get the best performance.

A. OPENH PROGRAMMING MODEL

An OpenH hybrid parallel program starts as a single *main* thread, which then creates a group of Pthreads called *hosting* Pthreads.

Unlike OpenMP, OpenACC, and Pthreads, the OpenH programming model explicitly abstracts not only program threads but also the physical and logical CPU cores and the accelerators of the hybrid executing platform. The physical CPU cores are represented by IDs that are integers in the array, $\{0, \ldots, npc - 1\}$, where npc is the total number of physical cores in the platform. Similarly, the logical CPU cores are represented by integer identifiers given by the array, $\{0, \ldots, nlc - 1\}$, where nlc is the total number of logical cores in the platform. Finally, the accelerators are signified by IDs taking values in the array, $\{0, \ldots, nacc - 1\}$, where $nacc$ is the total number of accelerators in the platform. Apart from an ID, each accelerator also has an OpenH type. However, all CPU cores are assumed to be the same type and do not possess an OpenH type. Therefore, the OpenH programming model employs a simple abstraction of CPU cores and accelerators and hides the intricate hardware topology of the hybrid server platform.

The OpenH program execution model is a fork-join model illustrated in Figure 4 where the main thread creates a group of hosting Pthreads responsible for leading the execution of the software components in parallel.

A CPU hosting Pthread leads the execution of a multithreaded CPU software component employing either OpenMP or a multithreaded library routine. There can be one or more CPU software components and, therefore, one or more CPU hosting Pthreads. For a CPU component employing an OpenMP parallel region, the hosting Pthread of the component becomes the *master* thread of the region.

An accelerator hosting Pthread leads the execution of an accelerator component, which is an OpenACC (or OpenMP) component running on one of the accelerators of the server.

The hybrid program's main thread joins with the group after completing the computations, forming a barrier or synchronization point. The results of the computations can be safely used for further processing after the *join* synchronization.

In addition, the hosting Pthreads can synchronize their actions during the execution of the software components by employing the Pthread synchronization API.

Finally, the OpenH library provides API functions that allow programmers to get the configuration of the executing environment. Furthermore, the library provides API functions for binding the hosting Pthreads (and hence the execution of the software components) to the CPU cores of the hybrid server to get the best performance.

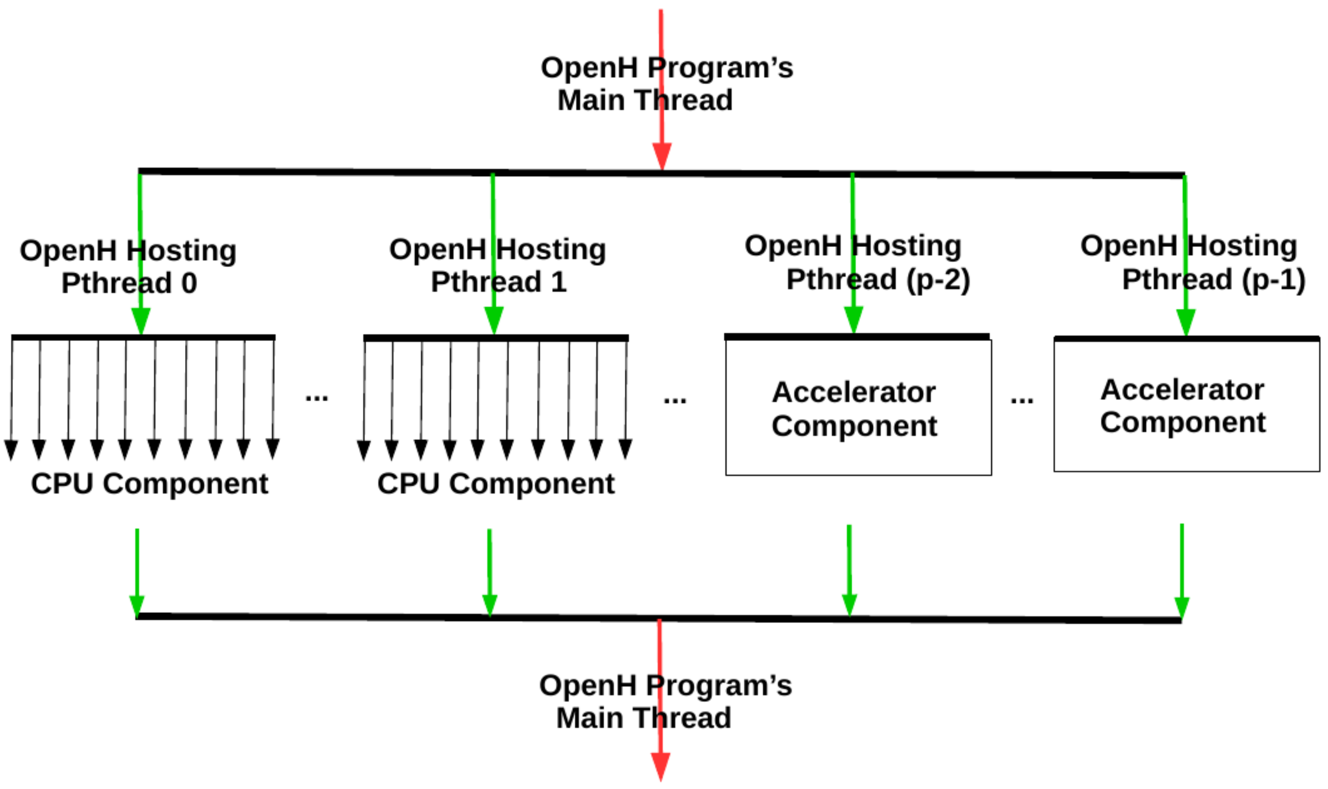


FIGURE 4. **The OpenH fork-join programing model comprising a group of *p* Pthreads called *hosting Pthreads*. Each *hosting Pthread* leads the execution of either a CPU software component or an accelerator software component of the program. The CPU components are shown as multithreaded OpenMP components with OpenMP threads indicated by dark solid arrows.**

B. OPENH LIBRARY API

The OpenH library provides sets of API functions for environment, thread placement and processor binding.

1) API FOR ENVIRONMENT AND TOPOLOGY

The OpenH library runtime is initialized and destroyed using the following API functions:

```
int openh_init();
int openh_finalize();
```

The above functions must only be invoked by the *main* thread. It is erroneous to call any other OpenH library function before *openh_init()*.

The main thread and the hosting Pthreads can invoke the rest of the API functions in this category.

The API function, *openh_is_hyperthreaded()*, returns 1 if hyperthreading is enabled on the platform and 0 otherwise. The function is *reentrant* and *thread-safe*.

```
int openh_is_hyperthreaded();
```

The API functions, *openh_get_num_pcores()* and *openh_get_num_lcores()*, return the number of physical and logical CPU cores in the platform, respectively. Both functions are reentrant and thread-safe.

```
int openh_get_num_pcores();
int openh_get_num_lcores();
```

If hyperthreading is not present, the number of physical CPU cores will be the same as the number of logical CPU cores.

If hyperthreading is present, the number of logical cores will be greater than the number of physical cores. For this case, the OpenH library provides two API functions for obtaining the mapping between the OpenH logical CPU core IDs to the OpenH physical CPU core IDs.

The first API function, *openh_get_mapping_scheme()*, returns the underlying hardware vendor's scheme for mapping logical CPU core IDs to physical CPU core IDs. The function is *reentrant* and *thread-safe*.

```
int openh_get_mapping_scheme();
```

The current version of the OpenH library provides the enum, *openh_mapping_scheme*, containing two well-known mapping schemes, *round-robin* (*OPENH_L2P_ROUNDROBIN*) and *linear* (*OPENH_L2P_LINEAR*), supported by mainstream hardware vendors and a mapping scheme (*OPENH_L2P_NWK*) covering the rest of the cases, for example, platforms where logical CPU core IDs are mapped to physical CPU core IDs with a stride.

```
enum openh_mapping_scheme {
  OPENH_L2P_ROUNDROBIN = 0,
  OPENH_L2P_LINEAR,
  OPENH_L2P_NWK
};
```

The round-robin scheme for mapping logical CPU core IDs to physical CPU core IDs is the default in Intel and AMD multicore CPU processors. The mapping scheme can be changed to linear in BIOS in AMD processors.

The second API function, *openh_get_mapping_function()*, returns the mapping function given the input mapping scheme, *mapping_scheme*. It is *reentrant* and *thread-safe*.

```
typedef int ( *openh_pcoreid_func )(
         int openh_lcoreid);
openh_pcoreid_func
    openh_get_mapping_function(
        int mapping_scheme);
```

The programmer uses the mapping function pointer, *openh_pcoreid_func*, provided by the second API function, *openh_get_mapping_function()*, to obtain the OpenH physical CPU core ID associated with the input OpenH logical CPU core ID, *openh_lcoreid*. Therefore, the programmer first obtains the mapping scheme using *openh_mapping_scheme()*, then uses this scheme to obtain the mapping function pointer via the API function call, *openh_get_mapping_function()*, and finally obtains the physical CPU core ID mapped to a logical CPU core ID using the mapping function pointer.

The programmers can use their own mapping functions, but this will hinder the portability of their OpenH programs.

We illustrate the two well-known mapping schemes, *round-robin* and *linear*. Consider a dual-socket multicore CPU with eight physical cores per socket and two logical cores per physical core. Then the OpenH physical CPU core IDs are $\{0, 1, \cdots, 15\}$ and the OpenH logical CPU core IDs are $\{0, 1, \cdots, 31\}$.

Suppose the mapping scheme is *round-robin*. The API function call, *openh_get_mapping_scheme()*, returns *OPENH_L2P_ROUNDROBIN*. The OpenH logical CPU core IDs, $\{0, 1, \cdots, 31\}$, map to the array of OpenH physical CPU core IDs, $\{0, \cdots, 15, 0, \cdots, 15\}$. Therefore, for example, the function pointer, *openh_pcoreid_func()*, obtained using the mapping function, *openh_get_mapping_function()*, will return the OpenH physical CPU core ID of 0 for input OpenH logical CPU ID of 16 and 1 for input 17.

Suppose the mapping scheme is *linear*. The API function call, *openh_get_mapping_scheme()*, returns *OPENH_L2P_LINEAR*. the OpenH logical CPU core IDs, $\{0, 1, \cdots, 31\}$, map to the array of OpenH physical CPU core IDs, $\{0, \cdots, 7, 0, \cdots, 7, 8, \cdots, 15, 8, \cdots, 15\}$. Therefore, for example, the function pointer, *openh_pcoreid_func()*, obtained using the mapping function, *openh_get_mapping_function()*, will return the OpenH physical CPU core id of 8 for input OpenH logical CPU ID of 16 and 9 for input 17.

The API function, *openh_get_ptol_function()*, allows the programmer to obtain the list of OpenH logical CPU core IDs associated with an OpenH physical CPU core ID. It is *reentrant* and *thread-safe*.

```
typedef int ( *openh_ptol_func )(
           int openh_pcoreid, int**
               lcpuids, int* nlids);
openh_ptol_func
    openh_get_ptol_function(
           int mapping_scheme);
```

The programmer first obtains the mapping scheme using *openh_mapping_scheme()* and then uses this scheme to obtain the physical-to-logical function pointer, *openh_ptol_func*, using the function *openh_get_ptol_function()*. Finally, the programmer obtains the list of OpenH logical CPU core IDs in the array, *lcpuids*, using the function pointer, *openh_ptol_func*, for the input physical CPU core ID, *openh_pcoreid*. The size of the array *lcoreids* is *nlids*. Memory for *lcoreids* is obtained using *malloc()* and can be freed with *free()*.

The following code snippet demonstrates the use of the API function, *openh_get_ptol_function()*, to obtain and print the list of OpenH logical CPU core IDs for input OpenH physical CPU core ID, *pcpuid*.

```
int i, mscheme =
    openh_get_mapping_scheme();
openh_ptol_func optolf =
    openh_get_ptol_function(mscheme);
int* lcpuids, nlids;
int rc = optolf(pcpuid, &lcpuids,
    &nlids);
printf(
  "Logical IDs for physical CPU ID %d:
      ",
pcpuid);
for (i = 0; i < nlids; i++) {
   printf("%d ", lcpuids[i]);
}
printf("\n");
free(lcpuids);
```

2) API FUNCTIONS FOR ACCELERATORS

We now present the API functions concerning the accelerators.

The API function, *openh_get_num_accelerators()*, returns the number of accelerators in the execution platform.

```
int openh_get_num_accelerators();
```

The API function, *openh_get_acc_type()*, returns the type of the accelerator given the *id*.

```
int openh_get_acc_type(int id);
```

The current version of the OpenH library provides the following enum, *openh_acc_type*, containing the supported accelerator types.

```
enum openh_acc_type {
  OPENH_CUDA_GPU = 0,
  OPENH_AMD_GPU,
  OPENH_FPGA,
  OPENH_UNKNOWN
};
```

The API functions, *openh_get_num_accelerators()* and *openh_get_acc_type()*, together allow the programmer to obtain a mapping between the accelerator IDs and types. Both functions are *reentrant* and *thread-safe*.

The OpenH physical CPU core IDs closest to an accelerator can be obtained using the API function *openh_get_accelerator_pcpuaffinity*.

```
int openh_get_accelerator_pcpuaffinity(
        int accnum, int** closestPcpuids,
        int* npcpuids);
```

The above function returns the closest OpenH physical CPU core IDs to the accelerator, *accnum*, in the array, *closestPcpuids*. The size of the output array is *npcpuids*. Memory for *closestPcpuids* is obtained using *malloc()* in the API function, and can be freed with *free()*.

Similarly, the closest OpenH logical CPU core IDs can be obtained using the API function *openh_get_accelerator_lcpuaffinity*.

```
int openh_get_accelerator_lcpuaffinity(
       int accnum, int** closestLcpuids,
       int* nlcpuids);
```

Both *openh_get_accelerator_pcpuaffinity()* or *openh_get_accelerator_lcpuaffinity()* are *reentrant* and *thread-safe*.

If hyperthreading is not present, the set of physical CPU core IDs in the array, *closestPcpuids*, will be the same as the logical CPU core IDs in the array, *closestLcpuids*. In this case, to get optimal performance, the programmmer must bind different accelerator hosting Pthreads to different CPU core IDs in the output array from *openh_get_accelerator_pcpuaffinity()* or *openh_get_accelerator_lcpuaffinity()*.

If hyperthreading is present, we would recommend the programmer bind different accelerator hosting Pthreads to different physical CPU core IDs using the API function, *openh_get_accelerator_pcpuaffinity()*. Using *openh_get_accelerator_lcpuaffinity()* API call to bind different accelerator hosting Pthreads to different logical CPU core IDs may lead to overloading or oversubscribing a physical CPU core if the logical CPU cores happen to share the same physical CPU core. Such a case can arise when the hardware vendor follows a linear enumeration scheme for logical CPU core IDs.

Furthermore, if there is one single-socket multicore CPU and multiple accelerators on a platform, all the accelerators will likely be closest to all the logical cores of the multicore CPU. Therefore, the output arrays, *closestPcpuids* and *closestLcpuids*, will be the same for all the accelerators. Hence, to get optimal performance, the programmer must bind different accelerator hosting Pthreads to different physical CPU core IDs or logical CPU core IDs that do not share the same physical CPU core.

Therefore, the OpenH library provides two high-level functions that allow the programmer to assign unique OpenH physical CPU core IDs or OpenH logical CPU core IDs that map to unique OpenH physical CPU core IDs.

```
int openh_get_unique_pcore(int accId);
int openh_get_unique_lcore(int accId);
```

The API function, *openh_get_unique_pcore()*, assigns a unique OpenH physical CPU core ID to pin the hosting Pthread for the accelerator *accId*. Finally, the API function, *openh_get_unique_lcore()*, assigns a unique OpenH logical CPU core ID to pin the hosting Pthread for the accelerator *accId*. Furthermore, different accelerator hosting Pthreads are assigned unique OpenH logical CPU core IDs that map to a unique OpenH physical CPU core ID when using the API function, *openh_get_unique_lcore()*.

The two API functions, *openh_get_unique_pcore()* and *openh_get_unique_lcore()*, are not *thread-safe*. It is recommended that programmers invoke these functions only in the *main thread*. Furthermore, only one of the functions must be used for obtaining the unique OpenH CPU core IDs to assign for binding the accelerator hosting Pthreads.

The two high-level API functions above are implemented on top of the basic API functions, *openh_get_accelerator_pcpuaffinity()* and *openh_get_accelerator_lcpuaffinity()*. They are high-level helpers provided for programmers who prefer the defaults employed by OpenH library for their OpenH programs.

The two API functions, *openh_get_accelerator_pcpuaffinity()* and *openh_get_accelerator_lcpuaffinity()*, cater to advanced programmers who would like to design and implement a different portable solution to the one the OpenH library provides that is optimal for their platform.

We illustrate the use cases for the affinity API functions for accelerators in our description of an OpenH hybrid parallel matrix multiplication application in Section V-A.

3) ASSIGNMENT AND BINDING API FOR HOSTING THREADS AND SOFTWARE COMPONENTS

The OpenH library provides two sets of API functions for assigning the CPU core IDs and binding the main thread and hosting Pthreads and the execution of software components to the assigned CPU core IDs, called *affinity* and *binding* APIs.

Like OpenMP, the *affinity* API assign the places (or CPU core IDs) for binding (or pinning) the main thread and the hosting Pthreads. The *binding* API bind the hosting Pthreads to the CPU core IDs assigned using the *affinity* API. For the CPU component, the *affinity* and *binding* API functions assign the CPU core IDs for binding the CPU hosting Pthread and the execution of the CPU component on the CPU core IDs.

Therefore, an OpenH program will always have the affinity API function invocations followed by binding API function invocations. Furthermore, the *affinity* API must be invoked only by the main thread.

The primary reason for division into *affinity* and *binding* APIs is to avoid locking and synchronization between Pthreads to get and set the CPU core IDs for binding. Furthermore, both *affinity* and *binding* API functions are not *thread-safe*.

The following API functions assign the OpenH physical CPU core IDs for binding the main and hosting Pthreads:

```
void openh_assign_main_pcpuid(int
    pcpuid);
void openh_assign_acc_pcpuids(int
    accId, int* pcpuids, int size);
void openh_assign_cpu_pcpuids(int
    cpuComponentId, int* pcpuids, int
    size);
```

The API function, *openh_assign_main_pcpuid*, assigns the OpenH physical CPU core ID, *pcpuid*, for binding the main thread. It must be invoked by the main thread only.

The API function, *openh_assign_acc_pcpuids*, assigns the OpenH physical CPU core IDs provided in the array, *pcpuids*, for binding the hosting Pthread for the accelerator *accId*. The size of the array is provided in the *size* argument.

The API function, *openh_assign_cpu_pcpuids*, assigns the OpenH physical CPU core IDs provided in the array, *pcpuids*, for binding the CPU hosting Pthread and the execution of the CPU component *cpuComponentId*. The size of the array is provided in the *size* argument.

Similarly, the following API functions assign the OpenH logical CPU core IDs for binding the main and hosting Pthreads:

```
void openh_assign_main_lcpuid(int
    lcpuid);
void openh_assign_acc_lcpuids(int
    accId, int* lcpuids, int size);
void openh_assign_cpu_lcpuids(int
    cpuComponentId, int* lcpuids, int
    size);
```

Finally, the following API functions are helpers for automatically assigning the OpenH physical and logical CPU core IDs for binding the CPU hosting Pthread and the execution of the CPU component *cpuComponentId*:

```
void openh_assign_cpu_free_pcpuids(
    int cpuComponentId);
void openh_assign_cpu_free_lcpuids(
    int cpuComponentId);
```

After assigning the OpenH physical CPU core IDs for the accelerator hosting Pthreads, the programmers can use *openh_assign_cpu_free_pcpuids()* to assign the remaining free OpenH physical CPU core IDs for binding the hosting Pthread and therefore for the execution of the CPU component *cpuComponentId*. Similarly, *openh_assign_cpu_free_lcpuids()* assigns the remaining free OpenH logical CPU core IDs. Note that once the CPU hosting Pthread is assigned physical CPU core IDs for binding using *openh_assign_cpu_free_pcpuids()*, there will be no free physical CPU core IDs left. Similarly, once the CPU hosting Pthread is assigned logical CPU core IDs for binding using *openh_assign_cpu_free_lcpuids()*, there will be no free logical CPU core IDs left.

The following *binding* API functions actually bind the hosting Pthreads to the CPU core IDs that have been assigned using the *affinity* API functions.

```
int openh_bind_main_self();
int openh_bind_acc_self(int accId);
int openh_bind_cpu_self(int
    cpuComponentId);
```

The function *openh_bind_main_self()* must be called by the main thread only. The function *openh_bind_acc_self()* is called by the hosting Pthread for the accelerator *accId*. It binds the accelerator hosting Pthread to the core IDs assigned for binding in the API function calls, *openh_assign_acc_pcpuids()* or *openh_assign_acc_lcpuids()*.

The function *openh_bind_cpu_self()* is called by the hosting Pthread for the CPU component *cpuComponentId*. It binds the CPU hosting Pthread and the execution of the CPU component to the CPU core IDs assigned for binding in the API function calls, *openh_assign_cpu_pcpuids()* or *openh_assign_cpu_lcpuids()*. Therefore, the CPU component executes only on the CPU core IDs assigned using the affinity assignment API.

Finally, these functions are typically the first invocations in the Pthread function execution of the corresponding component. For example, the function *openh_bind_cpu_self()* would be invoked before any processing in the Pthread function associated with a CPU component.

V. HYBRID PARALLEL APPLICATIONS USING OPENH

We present the design and implementation of two hybrid parallel applications based on matrix multiplication and 2D fast Fourier transform (2D-FFT), respectively. We then discuss interesting use cases of the OpenH library employing the hybrid parallel matrix multiplication application as an example.

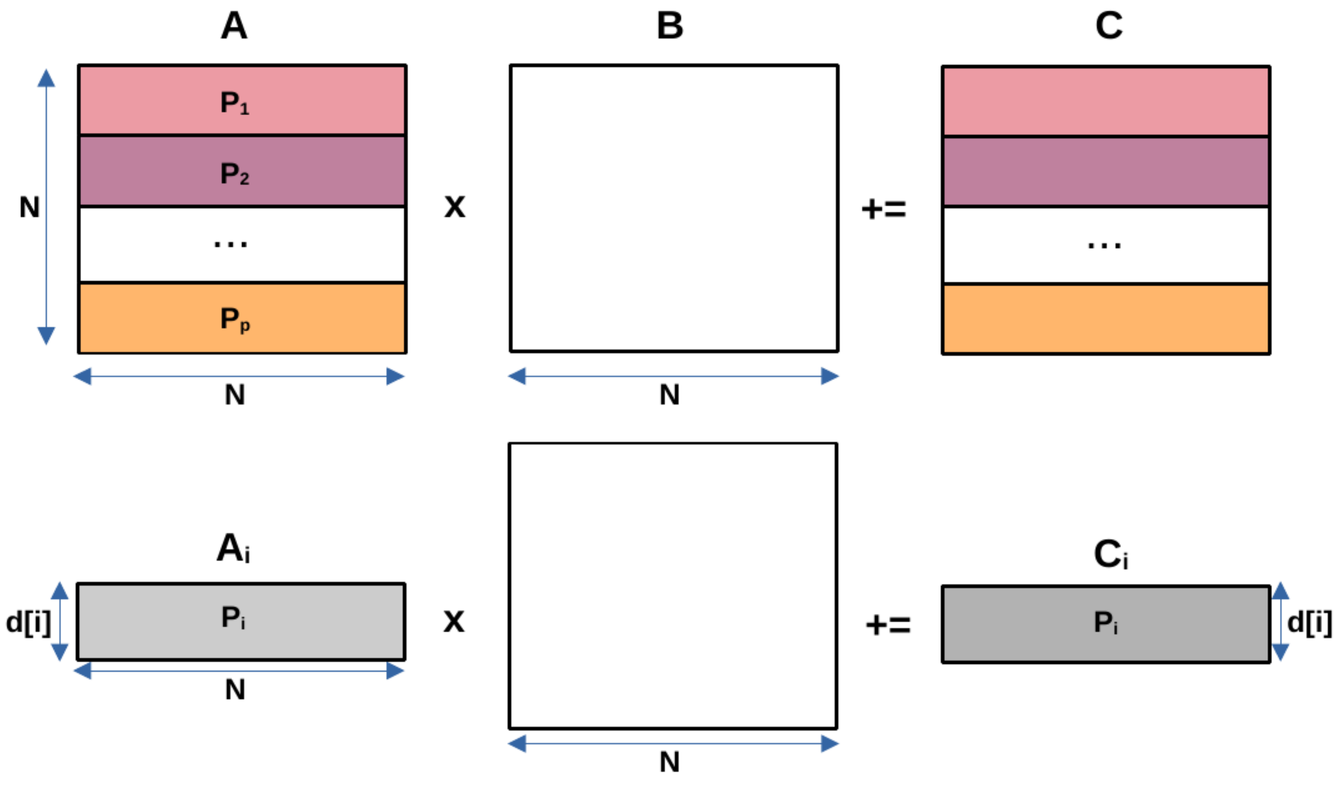


FIGURE 5. An OpenH hybrid parallel matrix multiplication application computing the matrix product ($C+ = A \times B$) of two dense square matrices A and B of size $N \times N$ using p computing devices (one multicore CPU and $p-1$ accelerators). It comprises a group of p hosting Pthreads leading the execution of p software components, one CPU component and $p-1$ accelerator components. The matrix B is shared by all the components. Each component i is assigned a number of rows of A and C proportional to its performance and given by $d[i]$.

A. HYBRID PARALLEL MATRIX MULTIPLICATION APPLICATION USING OPENH

This section illustrates the OpenH library API to develop a hybrid parallel matrix multiplication application on a server comprising p computing devices, one multicore CPU and $p-1$ accelerators.

We first present the design and implementation of the application for the most general case of a hybrid server, which is hyperthreaded and comprises *nlc* number of logical CPU cores. The design employs only the group of API functions provided for dealing with OpenH logical CPU core IDs. Therefore, the application is designed to be portable to all hyperthreaded and non-hyperthreaded hybrid platforms without any code changes.

1) DESIGN AND IMPLEMENTATION

Figure 5 illustrates the application. It computes the matrix product ($C+ = A \times B$) of two dense square matrices A and B of size $N \times N$. It comprises p software components, $\{S_1, \ldots, S_p\}$, one CPU component and $p-1$ accelerator components. The main thread creates the group of p hosting CPU and accelerator Pthreads to lead the execution of the CPU and the accelerator software components in parallel.

The matrices A and C are partitioned horizontally among the p components such that each component is assigned a number of rows of A and C proportional to its performance. The distribution of rows of A and C is provided in the array, d. The matrix B is shared by all the components. So, each component P_i computes its horizontal partition C_{P_i} using the matrix product, $C_{P_i} + = A_{P_i} \times B$.

In the supplemental, we provide the implementation of our hybrid program that shows how the workload distribution array, d, is automatically determined using the performances of the software components estimated at runtime. We do not present the OpenH API functions employed for determining the performances and the workload distribution since they are a work in progress.

We first determine the relative performances of the software components using the OpenH API function, *openh_perf_benchmark()*, which executes small representative benchmark codes of the software components solving the same workload size in parallel. The execution times of all the benchmark codes are measured simultaneously, thereby considering the influence of resource contention. The API function implementation essentially executes a mini-version of the hybrid matrix multiplication application employing the same affinity and binding settings for the Pthreads executing the benchmark codes as the hosting Pthreads and the same library settings for the library routines invoked in the software component implementations.

```
1  #include <openh.h>
2  typedef struct _ComponentPartition_t {
3      int cId; int M; int N; int K;
4      double* A; double* B, double* C;
5  } ComponentPartition_t;
6
7  int main(int argc, char *argv[]) {
8      openh_init();
9      /* Get the number of accelerators */
10     int nacc = openh_get_num_accelerators();
11     /* Set the number of software components */
12     int p = nacc+1;
13     /* Assign the CPU core IDs for the accelerator hosting
          Pthreads */
14     for (i = 0; i < nacc; i++) {
15         int lcoreid = openh_get_unique_lcore(i);
16         openh_assign_acc_lcpuids(i, &lcoreid, 1);
17     }
18     /* Assign the remaining CPU core IDs that execute the
          CPU component */
19     openh_assign_cpu_free_lcpuids(0);
20     int start = 0; ComponentPartition_t cData[p];
21     pthread_t compT[p];
22     for (i = 0; i < p-1; i++) {
23         cData[i].cId = i;
24         cData[i].M = d[i]; cData[i].N = N;
25         cData[i].K = N; cData[i].A = &A[start];
26         cData[i].B = B; cData[i].C = &C[start];
27         pthread_create(&compT[i], NULL,
28         (openh_get_acc_type(i) == OPENH_CUDA_GPU)
29             : accCudaMxmComponent ?
                   accNonCudaMxmComponent,
30             (void*)(&cData[i]));
31         start += d[i]*N;
32     }
33
34     cData[p-1].cId = 0; cData[p-1].M = d[p-1];
35     cData[p-1].N = N; cData[p-1].K = N;
36     cData[p-1].A = &A[start]; cData[p-1].B = B;
37     cData[p-1].C = &C[start];
38     pthread_create(&compT[p-1], NULL, cpuMxmComponent,
           (void*)(&cData[p-1]));
39     for (i = 0; i < p; i++) {
40         pthread_join(compT[i], NULL);
41     }
42     openh_finalize();
43 }
```

FIGURE 6. **The OpenH parallel matrix multiplication code illustrating the OpenH fork-join model of execution. The OpenH library calls are highlighted in bold. The main thread creates the group of hosting CPU and accelerator Pthreads to lead the execution of the CPU and the accelerator software components in parallel. If the accelerator is a CUDA-enabled GPU signified by the enum value OPENH_CUDA_GPU, then the accelerator hosting Pthread will lead the execution of the Nvidia CUDA GPU component, *accCudaMxmComponent*. Otherwise, the accelerator hosting Pthread will lead the execution of the *accNonCudaMxmComponent*.**

```
1  void *cpuMxmComponent(void *arg) {
2      ComponentPartition_t cData = *(ComponentPartition_t*)arg;
3      int cpuComponentId = cData.cId;
4      /* Bind the CPU hosting Pthread */
5      openh_bind_cpu_self(cpuComponentId);
6      /* Set the number of threads to execute DGEMM */
7      cblas_set_num_threads(openh_get_num_lcores() -
8                 openh_get_num_accelerators());
9      int M = cData.M, N = cData.N;
10     int K = cData.K; double* dA = cData.A;
11     double* B = cData.B; double* dC = cData.C;
12     cblas_dgemm(CblasRowMajor, CblasNoTrans,
13                 CblasNoTrans, M, N, K, 1.0,
14                 dA, K, B, N, 1.0, dC, N);
15 }
16 void *accNonCudaMxmComponent(void *arg) {
17     ComponentPartition_t cData = *(ComponentPartition_t*)arg;
18     int accId = cData.cId;
19     /* Bind the non-GPU accelerator hosting Pthread */
20     openh_bind_acc_self(accId);
21     int i, j, k, M = cData.M, N = cData.N;
22     int K = cData.K; double* dA = cData.A;
23     double* B = cData.B; double* dC = cData.C;
24 #pragma acc data copyin(A[0:M*K], B[0:K*N]) copyout(C[0:M*N]) {
25 #pragma acc parallel loop gang
26     for (j = 0; j < N; j++) {
27 #pragma acc loop vector
28         for (i = 0; i < M; i++) {
29             double sum = 0.0;
30             for (k = 0; k < K; k++)
31                 sum += A[i*K + k] * B[k*N + j];
32             C[i*N + j] = sum;
33         }
34     }
35 }
36 void *accCudaMxmComponent(void *arg) {
37     ComponentPartition_t cData = *(ComponentPartition_t*)arg;
38     int gpuId = cData.cId;
39     /* Bind the GPU accelerator hosting Pthread */
40     openh_bind_acc_self(gpuId);
41     /* Set the Nvidia GPU device num */
42     acc_set_device_num(gpuId, acc_device_nvidia);
43     int M = cData.M, N = cData.N;
44     int K = cData.K; double* dA = cData.A;
45     double* B = cData.B; double* dC = cData.C;
46     cublasHandle_t handle;
47     cublasStatus_t status = cublasCreate(&handle);
48 #pragma acc data copyin(M, N, K, dA, B) copy(dC) {
49 #pragma acc host_data use_device(dA, B, dC) {
50         double alpha = 1.0, beta = 1.0;
51         cublasDgemm(handle, CUBLAS_OP_N,
52                 CUBLAS_OP_N, M, N, K, &alpha,
53                 dA, M, B, K, &beta, dC, M);
54     }
55     }
56     cublasDestroy(handle);
57 }
```

FIGURE 7. **The OpenH parallel matrix multiplication application is decomposed into p software components (one CPU and $p - 1$ accelerator components) and executing on a heterogeneous platform comprising p devices, one multicore CPU and $p - 1$ accelerators. The OpenH library calls are highlighted in bold. The CPU software component implementation is in the function, *cpuMxmComponent*. The software component implementation specific to Nvidia CUDA GPU is in the function, *accCudaMxmComponent*. For any other accelerator, the software component implementation is in the function, *accNonCudaMxmComponent*.**

The performances are then used to determine the workload distribution using the API function, *openh_get_wd()*. Note that the sum total of the execution times of the benchmark codes, the determination of the performances and workload distribution are insignificant compared to the total execution time of the hybrid program. Furthermore, the OpenH API function for benchmarking performances essentially reuses the structure of the hybrid program and the software component implementations.

Figures 6 and 7 illustrate the main steps of execution of the OpenH parallel matrix multiplication application.

We will first describe the execution steps of the *main* thread in the application (Figure 6). Line 8 initializes the OpenH library runtime using the API function *openh_init()*. The API function, *openh_get_num_accelerators*, returns the number of accelerators (Line 10).

The variable p stores the number of hosting Pthreads in the program, which is equal to the number of software components. Lines 14-17 determine and assign the physical CPU core IDs closest to the accelerators for binding the accelerator hosting Pthreads. The API function, *openh_get_unique_lcore(i)*, returns the unique OpenH logical CPU core ID closest to the input accelerator, *i*. The hosting Pthread for the accelerator *i* is assigned the place using the API function, *openh_assign_acc_lcpuids* (Line 16). The OpenH library functions ensure that different accelerator hosting Pthreads are set to OpenH logical CPU core IDs that map to different OpenH physical CPU core IDs for optimal performance.

The CPU software component with ID 0 is assigned the remaining OpenH logical CPU core IDs using the API function, *openh_assign_cpu_free_lcpuids()*, that execute the component (Line 19). The number of logical CPU core IDs for binding the CPU hosting Pthread is $nlc - nacc$ since nlc is the number of logical cores in the platform and $nacc$ logical CPU core IDs are assigned to accelerator hosting Pthreads.

Lines 22-32 show the creation of the $p - 1$ accelerator hosting Pthreads responsible for executing the accelerator components. The partition data for an accelerator software component is filled in the Lines 23-26. The accelerator type for the accelerator *i* is returned by *openh_get_acc_type()*. If the accelerator type is OPENH_CUDA_GPU, then the accelerator hosting Pthread will lead the execution of the Nvidia CUDA GPU component, *accCudaMxmComponent*. This component invokes the CUBLAS DGEMM routine to compute the matrix product. Otherwise, the accelerator hosting Pthread will invoke the software component implementation provided in the function, *accNonCudaMxmComponent*.

Lines 34-37 contain the filling of the partition data for the CPU component and the creation of the hosting Pthread leading the execution of the CPU component. This component invokes the OpenBLAS DGEMM routine to compute the matrix product.

After completing the computations, the main thread synchronizes/joins with the *p* hosting Pthreads in Lines 39-41. Finally, the OpenH runtime is destroyed using the API function, *openh_finalize()*.

Figure 7 show the main code fragments of the software components. Lines 1-15 contain the CPU software component code. The hosting Pthread leading the execution of the CPU component with ID, *cpuComponentId*, is first bound using the API function, *openh_bind_cpu_self()*. Then, it executes the CBLAS DGEMM routine to compute the matrix product, $dC = dA \times B$, multiplying its horizontal slice in A with B.

The DGEMM routine is executed using threads bound to OpenH logical CPU core IDs that are set using the API function, *openh_assign_cpu_free_lcpuids()*, in the main thread. The number of threads employed by the DGEMM routine is set using the CBLAS library API function, *cblas_set_num_threads()*, which will be different for different CBLAS library implementations. For OpenBLAS implementation, the library API function is *oblas_set_num_threads()*. For the Intel MKL library implementation, the library API function is *mkl_set_num_threads()*. The number of threads passed to the function is equal to $nlc - nacc$ where nlc is the number of logical cores in the platform given by the API function, *openh_get_num_lcores()*, and $nacc$ is the number of accelerators given by the API function, *openh_get_num_accelerators()*.

Therefore, the OpenH *main* thread shares a core with one of the hosting Pthreads. However, this sharing does not affect the performance since the main thread is not involved in the computations in this application and waits in the *pthread_join()* loop to synchronize with the hosting Pthreads (Figure 6, Lines 39-41).

Lines 16-35 demonstrate the execution of a component employing an accelerator that is not an Nvidia GPU. The hosting Pthread leading the execution of *accId* is bound using the API function, *openh_bind_acc_self()*. Lines 24-35 show the use of OpenACC pragmas to compute the matrix product, $dC = dA \times B$. The *pragma acc data copyin* (Line 24) directive allocates memory on the accelerator for A, B, and C, respectively, of sizes $M * K$, $K * N$, and $M * N$ and copies data from the host to the accelerator when entering the data region enclosed by the directive.

The OpenACC execution model has three levels: *gang*, *worker*, and *vector*. For Nvidia GPUs, an OpenACC gang is a threadblock, a worker is a warp, and a vector is a CUDA thread. Therefore, the *pragma acc parallel loop gang* directive (Line 25) marks the loop for gang parallelism, which maps to grid-level parallelism (for example, CUDA grid) for the accelerator. The *pragma acc loop vector* (Line 27) marks the loop for vector parallelism (for example, CUDA threads).

Finally, Lines 36-57 demonstrate the execution of a software component employing an Nvidia GPU. The hosting Pthread leading the execution of *gpuId* is bound using the API function, *openh_bind_acc_self()*. Lines 48-55 show the use of OpenACC pragma directives to execute the CUBLAS DGEMM routine to compute the matrix product, $dC = dA \times B$, on the GPU device, *gpuId*. The GPU device ID is set using the OpenACC library function, *acc_set_device_num*.

The *pragma acc host_data use_device* directive allows to get the device addresses of dA, B, and dC within host code that can then be input to CUDA library functions (such as the CUBLAS DGEMM routine) that expects CUDA device pointers. The directive essentially allows the compiler to generate code to use device copies of dA, B, and dC as arguments to the CUBLAS DGEMM routine, which is invoked on the CPU side. Lines 51-53 execute the CUBLAS DGEMM routine to compute the matrix product, $dC = dA \times B$, on the GPU.

B. HYBRID 2D FAST FOURIER TRANSFORM USING OPENH

This section illustrates the OpenH library API to develop a hybrid parallel 2D fast Fourier transform application (2D FFT) on a server comprising p computing devices, one multicore CPU and $p - 1$ accelerators.

Following in the same lines as the hybrid matrix multiplication example, we present the design and implementation of

```
#include <openh.h>
typedef struct _ComponentPartition_t {
    int cId; int M; int N; int direction ;
    fftw_complex* myS;
} ComponentPartition_t;
int openhfft2d( int direction , fftw_complex* signalMatrix ) {
    openh_init();
    int nacc = openh_get_num_accelerators();
    int p = nacc+1;
    for (i = 0; i < nacc; i++) {
        int lcoreid = openh_get_unique_lcore(i);
        openh_assign_acc_lcpuids(i, &lcoreid, 1);
    }
    openh_assign_cpu_free_lcpuids(0);
    int start = 0; ComponentPartition_t cData[p];
    pthread_t compT[p];
    for (i = 0; i < p-1; i++) {
        cData[i].cId = i; cData[i].direction = direction ;
        cData[i].M = d[i]; cData[i].N = N;
        cData[i].myS = &signalMatrix[ start ];
        pthread_create (&compT[i], NULL,
        (openh_get_acc_type(i) == OPENH_CUDA_GPU)
            : accCudaFFTComponent
            ? accNonCudaFFTComponent,
            (void*)(&cData[i]));
        start += d[i]*N;
    }
    cData[p-1].cId = 0; cData[i].direction = direction ;
    cData[p-1].M = d[p-1]; cData[p-1].N = N;
    cData[p-1].myS = &signalMatrix[start ];
    pthread_create (&compT[p-1], NULL, cpuFFTComponent,
            (void*)(&cData[p-1]));
    for (i = 0; i < p; i++)
        pthread_join (compT[i], NULL);
    openh_finalize();
    return 0;
}
int openhtranspose(fftw_complex* signalMatrix ) {
    openh_init();
    int lcpuids [nlc];
    for (i = 0; i < nlc; i++)
        lcpuids [i] = i;
    openh_assign_cpu_pcpuids(0, lpcuids, nlc);
    pthread_t compT;
    pthread_create (&compT, NULL, cpuTranspose, (void*)transposeData);
    pthread_join (compT, NULL);
    openh_finalize();
}
int main(int argc, char *argv []) {
    int direction = atoi (argv [1]);
    openhfft2d( direction , signalMatrix );
    openhtranspose( signalMatrix );
    openhfft2d( direction , signalMatrix );
    openhtranspose( signalMatrix );
}
```

FIGURE 8. The OpenH parallel 2D FFT application executing on a heterogeneous platform comprising *p* devices, one multicore CPU and *p* − 1 accelerators. The 2D FFT is accomplished in four steps: (1) Computing 1D FFTs on rows of the signal matrix (using *openhfft2d()*), (2) Tranpose of the signal matrix (using *openhtranspose()*), (3) 1D FFTs on rows of the signal matrix (*openhfft2d()*), and finally (4) Tranpose of the signal matrix (*openhtranspose*). In the *openhfft2d* function, the main thread creates a group of *p* hosting CPU and accelerator Pthreads to lead the execution of the CPU and the accelerator software components in parallel. In the *opentranspose* function, the main thread creates one hosting CPU Pthread to lead the execution of the CPU software component. The software component employs all the CPU cores during its execution.

```
void *cpuFFTComponent(void *arg) {
    ComponentPartition_t cData = *(ComponentPartition_t*)arg;
    int cpuComponentId = cData.cId;
    /* Bind the CPU hosting Pthread */
    openh_bind_cpu_self(cpuComponentId);
    /* Set the number of threads to execute FFT */
    fftw_plan_with_nthreads (openh_get_num_lcores() -
            openh_get_num_accelerators());
    int M = cData.M; int N = cData.N;
    fftw_complex* myS = cData.myS;
    int rank = 1, howmany = M, s[] = {N};
    int idist = N, odist = N, istride = 1, ostride = 1;
    int *inembed = s, *onembed = s;
    fftw_plan my_plan = fftw_plan_many_dft(
                        rank, s, howmany,
                        myS, inembed, istride , idist ,
                        myS, onembed, ostride , odist ,
                        sign , FFTW_ESTIMATE);
    fftw_execute (my_plan);
    fftw_destroy_plan (my_plan);
}
void *accNonCudaFFTComponent(void *arg) {
    ComponentPartition_t cData = *(ComponentPartition_t*)arg;
    int gpuId = cData.cId, direction = cData.direction ;
    /* Bind the non-GPU accelerator hosting Pthread */
    openh_bind_acc_self(accId);
    int i, j, k, M = cData.M, N = cData.N;
    cufftDoubleComplex *myS = (cufftDoubleComplex*)cData.myS;
#pragma acc data copyin( dft [0:N]) copyout(myS[0:M*N]) {
#pragma acc parallel loop gang
      for (row = 0; row < M; row++) {
#pragma acc loop vector
          for (p = 0; p < N; p++) {
              for (q = 0; q < N; q++) {
                  int t = (p*q) % N;
                  if (t == 0) {
                      myS[row*N+p].x += myS[row*N+q].x;
                      myS[row*N+p].y += myS[row*N+q].y;
                  } else {
                    double delta = dft[t].y / dft[t].x;
                    double m1 = myS[row*N+q].x - delta * myS[row*N+q].y;
                    double m2 = myS[row*N+q].y + delta * myS[row*N+q].x;
                    myS[row*N+p].x += dft[t].x * m1;
                    myS[row*N+p].y += dft[t].x * m2;
                  }
              }
          }
      }
}
void *accCudaFFTComponent(void *arg) {
    ComponentPartition_t cData = *(ComponentPartition_t*)arg;
    int gpuId = cData.cId, direction = cData.direction ;
    /* Bind the GPU accelerator hosting Pthread */
    openh_bind_acc_self(gpuId);
    /* Set the Nvidia GPU device num */
    acc_set_device_num(gpuId, acc_device_nvidia );
    int M = cData.M, N = cData.N;
    cufftDoubleComplex *myS = (cufftDoubleComplex*)cData.sMatrix;
#pragma acc data copyin(M, N) copy(myS) {
#pragma acc host_data use_device(myS) {
        int rank = 1, batch = M, s[] = {N}, idist = N, odist = N;
        int istride = 1, ostride = 1, *inembed = s, *onembed = s;
        cufftHandle plan;
        cufftPlanMany(&plan, rank, s,
                inembed, istride , idist , onembed, ostride , odist ,
                cuFFT_Z2Z, batch));
        cufftExecZ2Z(plan, myS, myS, direction );
        cufftDestroy (plan);
      }
    }
}
```

FIGURE 9. The CPU software component implementation computing the 1D FFTs of size N is in the function, *cpuFFTComponent*. The software component implementation specific to Nvidia CUDA GPU is in the function, *accCudaFFTComponent*. The CUDA implementation calls cuFFT API functions to compute a batch of 1D FFTs of size N. For any other accelerator, the software component implementation is in the function, *accNonCudaFFTComponent*. The transpose of the signal matrix employing all the CPU cores is presented in the supplemental.

the application for the most general case of a hybrid server, which is hyperthreaded and comprises *nlc* number of logical CPU cores. However, the hybrid 2D FFT implementation is more complex than the hybrid matrix multiplication implementation.

1) DESIGN AND IMPLEMENTATION

Figure 10 illustrates the OpenH hybrid parallel 2D FFT application computing the 2D-FFT of a signal matrix S of size $N \times N$ using p computing devices (one multicore CPU with *nlc* logical cores and $p - 1$ accelerators).

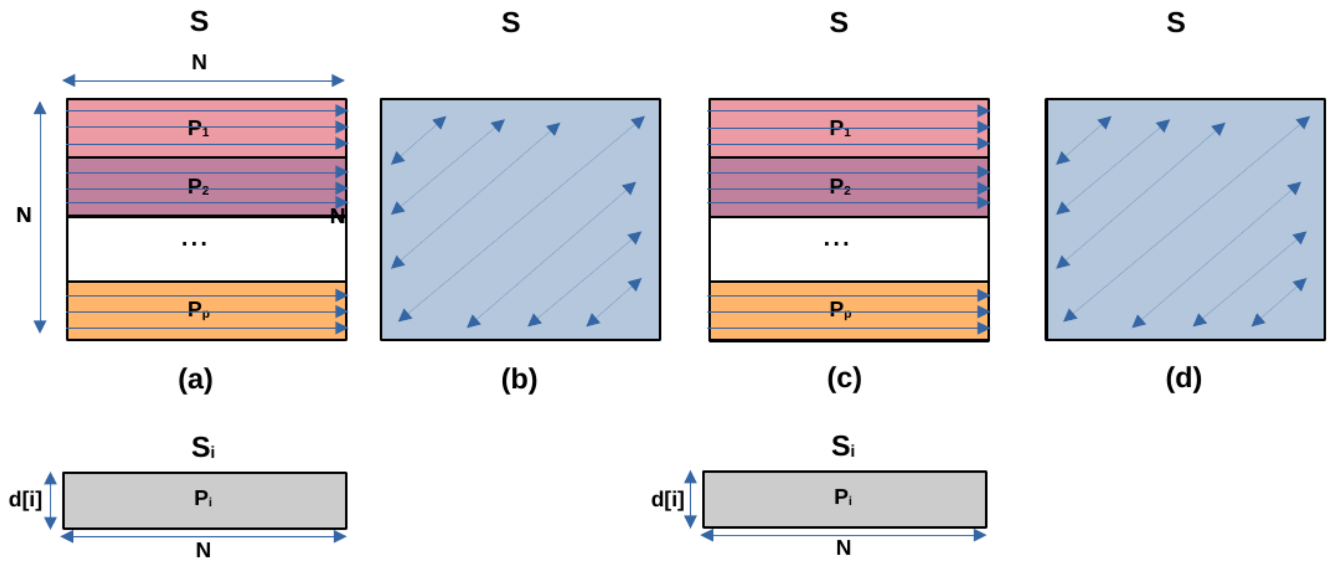


FIGURE 10. An OpenH hybrid parallel 2D FFT application computing the 2D-FFT of a signal matrix S of size $N \times N$ using p computing devices (one multicore CPU and $p-1$ accelerators). It comprises four main steps. (a). A group of p hosting Pthreads lead the execution of p software components, one CPU component and $p-1$ accelerator components, that perform N row 1D-FFTs of size N indicated by dotted arrows. Each component i is assigned a number of rows of S proportional to its performance and given by $d[i]$. (b). One hosting CPU Pthread leads the execution of a CPU software component that transposes the signal matrix S. Steps (c) and (d) are the repetition of steps (a) and (b), respectively.

It comprises four main steps: (a). Function *openhfft2d()* performing N row 1D FFTs of size N (Line 51). (b). Function *openhtranspose()* tranposing the signal matrix S (Line 52). Steps (c) and (d) are the repetition of steps (a) and (b), respectively. Note that both *openhfft2d()* and *openhtranspose()* are 2D-FFT application-specific functions and not OpenH library functions.

Unlike the matrix multiplication application presented earlier, where the whole computation is accomplished by just one group of OpenH hosting Pthreads, the main thread in 2D-FFT creates different groups of OpenH hosting Pthreads to execute a batch of 1D FFTs and transpose. The composition of the groups leading the execution of the batch of 1D FFTs and transpose differ.

In our hybrid implementation of 2D-FFT, executing the batch of 1D FFTs involves all the p computing devices, whereas the transpose is executed by all the nlc logical cores of the multicore CPU only. Note that one can have a 2D-FFT hybrid implementation containing a parallel hybrid transpose that employs all the p computing devices. In such an implementation, one group of OpenH hosting Pthreads will lead the execution of a batch of 1D FFTs and then the transpose. However, we found that this implementation complicated the exposition. Furthermore, the cost of creating and destroying a group of OpenH hosting Pthreads and setting their affinities is not a concern since it is insignificant compared to the total execution time of the application.

Figures 8 and 9 illustrate the main steps of execution of the OpenH parallel 2D FFT application.

In the function *openhfft2d()*, the main thread creates a group of p hosting Pthreads that lead the execution of p software components, one CPU component and $p-1$ accelerator components. The signal matrix S is partitioned horizontally among the p components such that each component is assigned a number of rows of S proportional to its performance. The distribution of rows of S is provided in the array, d. The workload distribution array, d, is automatically determined using the performances of the software components estimated at runtime.

Similar to the matrix multiplication application, we first determine the relative performances of the software components using the OpenH API function, *openh_perf_benchmark()*, which executes small representative benchmark codes of the software components solving the same workload size in parallel. The API function implementation essentially executes a mini-version of the hybrid 2D-FFT application employing the same affinity and binding settings for the Pthreads executing the benchmark codes as the hosting Pthreads and the same library settings for the library routines invoked in the software component implementations. The performances are then used to determine the workload distribution using the API function, *openh_get_wd()*. We do not present the OpenH API functions employed for determining the performances and the workload distribution since they are a work in progress.

In the function *openhfft2d()*, Line 7 initializes the OpenH library runtime using the API function *openh_init()*. The API function, *openh_get_num_accelerators*, returns the number of accelerators (Line 9). The variable p stores the number of hosting Pthreads, which is equal to the number of software components. Lines 10-13 assign the physical CPU core IDs closest to the accelerators for binding the accelerator hosting Pthreads. The API function, *openh_get_unique_lcore(i)*, returns the unique OpenH logical CPU core ID closest to the input accelerator, i. The hosting Pthread for the accelerator i is assigned the place using the API function, *openh_assign_acc_lcpuids* (Line 12).The CPU software component with ID 0 is assigned the remaining OpenH logical CPU core IDs using the API function, *openh_assign_cpu_free_lcpuids()*, that execute the component (Line 15).

Lines 17-27 show the creation of the $p - 1$ accelerator hosting Pthreads responsible for executing the accelerator components. The partition data for an accelerator software component is filled in the Lines 18-20. The accelerator type for the accelerator i is returned by *openh_get_acc_type()*. If the accelerator type is OPENH_CUDA_GPU, then the accelerator hosting Pthread will lead the execution of the Nvidia CUDA GPU component, *accCudaFFTComponent*. This component invokes cuFFT routines to compute the 1D FFTs. Otherwise, the accelerator hosting Pthread will invoke the software component implementation provided in the function, *accNonCudaFFTComponent*.

Lines 28-30 contain the filling of the partition data for the CPU component and the creation of the hosting Pthread leading the execution of the CPU component. This component invokes the Intel MKL FFT routines to compute the 1D FFTs.

After completing the computations, the main thread synchronizes/joins with the p hosting Pthreads in Lines 32-33. Finally, the OpenH runtime is destroyed using the API function, *openh_finalize()*.

In the function *openhtranspose()*, the main thread creates a hosting Pthread that leads the execution of a CPU software component computing the transpose of the signal matrix S (Lines 43-45). The hosting Pthread is assigned all the logical CPU cores using the API function, *openh_assign_cpu_lcpuids* (Line 42). The transpose is multithreaded and employs a number of threads equal to the number of logical CPU cores in the platform (*nlc*).

Figure 9 shows the main code fragments of the software components involved in executing the N 1D FFTs of the signal matrix S. Lines 1-21 contain the CPU software component code. The hosting Pthread leading the execution of the CPU component with ID, *cpuComponentId*, is first bound using the API function, *openh_bind_cpu_self()*. Then, it executes the Intel MKL FFT routines (using the FFTW interface) to compute M 1D FFTs of size N in its partition, *cData.myS*.

The Intel MKL FFT routine (*fftw_plan_many_dft()*) plans multiple multidimensional complex DFTs to compute *howmany* transforms, each having rank *rank* and size *s*. The Intel MKL FFT routine (*fftw_execute()*) is executed using threads bound to OpenH logical CPU core IDs that are set using the API function, *openh_assign_cpu_free_lcpuids()*, in the main thread. The number of threads employed by the routine is set using the library API function, *fftw_plan_with_nthreads()*. The number of threads passed to the function is equal to $nlc - nacc$ where *nlc* is the number of logical cores in the platform given by the API function, *openh_get_num_lcores()*, and *nacc* is the number of accelerators given by the API function, *openh_get_num_accelerators()*.

Therefore, the OpenH *main* thread shares a core with one of the hosting Pthreads. However, this sharing does not affect the performance since the main thread is not involved in the computations in this application and waits in the *pthread_join()* loop to synchronize with the hosting Pthreads (Figure 8, Lines 32-33).

Lines 22-49 demonstrate the execution of a component employing an accelerator that is not an Nvidia GPU. Note that the implementation of 2D FFT presented here is basic and unoptimized. The hosting Pthread leading the execution of *accId* is bound using the API function, *openh_bind_acc_self()*.

Lines 29-48 show the OpenACC pragmas employed to compute M 1D FFTs of size N in *myS*. The *pragma acc data copyin* (Line 29) directive allocates memory on the accelerator for *dft* and *myS*, respectively, of sizes N and $M * N$ and copies data from the host to the accelerator when entering the data region enclosed by the directive. The *pragma acc parallel loop gang* directive (Line 30) marks the loop for gang parallelism, which maps to grid-level parallelism (for example, CUDA grid) for the accelerator. The *pragma acc loop vector* (Line 32) marks the loop for vector parallelism (for example, CUDA threads). The loop (Line 31) computes M 1D FFTs of size N. Lines 33-47 compute an 1D FFT of size N.

Finally, Lines 50-71 demonstrate the execution of a software component employing an Nvidia GPU. The hosting Pthread leading the execution of *gpuId* is bound using the API function, *openh_bind_acc_self()*. Lines 50-60 show the use of OpenACC pragma directives to execute the cuFFT routine computing the 1D FFTs on the GPU device, *gpuId*. The GPU device ID is set using the OpenACC library function, *acc_set_device_num*.

The *pragma acc host_data use_device* directive (Lines 50-51) allows to get the device address of *myS* within host code that can then be input to CUDA library functions that expects CUDA device pointers. The directive essentially allows the compiler to generate code to use a device copy of *myS* as an argument to the cuFFT routine (*cufftExecZ2Z()*), which is invoked on the CPU side.

Lines 54-59 execute the cuFFT routines to compute M 1D FFTs of size N on the GPU. The cuFFT function *cufftPlanMany()* creates a FFT plan configuration of dimension *rank*, with sizes specified in the array *s*. The *batch* input parameter tells cuFFT how many transforms to configure. The cuFFT routine *cufftExecZ2Z()* executes a double-precision complex-to-complex transform plan in the transform direction as specified by *direction* parameter.

C. DISCUSSION OF THE USE CASES

We discuss here a few use cases for OpenH library API functions employing the hybrid parallel matrix multiplication above as an example.

1) HYPERTHREADING

We assume the hybrid server is hyperthreaded in the hybrid matrix multiplication application presented above. The application code invoked the OpenH library API functions that deal solely with OpenH logical CPU core IDs. For example, API functions *openh_get_unique_lcore()* and *openh_assign_acc_lcpuids()* are used to get the closest OpenH logical CPU core IDs to the accelerators to pin the accelerator hosting Pthreads and *openh_assign_cpu_free_lcpuids()* to select the OpenH logical CPU core IDs for binding the CPU hosting Pthread.

For a server that is not hyperthreaded, the programmer can use either of the two groups of API functions provided for dealing with OpenH physical or logical CPU core IDs since the set of the OpenH physical CPU core IDs is the same as the set of OpenH logical CPU core IDs.

For use cases where programmers find that hyperthreading hampers the performance of their applications, they can employ the group of API functions dealing with OpenH physical CPU core IDs only and desist from using the API functions involving the OpenH logical CPU core IDs. For example, the API functions *openh_get_unique_pcore()* and *openh_assign_acc_pcpuids()* can be used to get the closest OpenH physical CPU core IDs to the accelerators and *openh_assign_cpu_free_pcpuids()* to select the OpenH physical CPU core IDs for binding the CPU hosting Pthread and the execution of the CPU component.

2) AFFINITY API FOR THE ACCELERATORS

The OpenH library provides two high-level API functions, *openh_get_unique_pcore()* and *openh_get_unique_lcore()*, that allow programmers to obtain unique OpenH physical CPU core IDs or OpenH logical CPU core IDs that map to unique OpenH physical CPU core IDs for binding the accelerator hosting Pthreads.

However, these two functions are high-level helpers that provide one solution from a set of solutions for the problem, which is to determine unique OpenH physical CPU core IDs or OpenH logical CPU core IDs that map to unique OpenH physical CPU core IDs for binding the accelerator hosting Pthreads. They are provided for programmers who prefer the defaults employed by OpenH library for their OpenH programs.

The two API functions, *openh_get_accelerator_pcpuaffinity()* and *openh_get_accelerator_lcpuaffinity()*, cater to advanced programmers who would like to design and implement a different portable solution to the one the OpenH library provides that is optimal for their platform. Suppose a hybrid platform with a single-socket multicore CPU, sixteen physical cores, and two accelerators. Then, the OpenH physical CPU core IDs are $\{0, 1, \cdots, 15\}$. Also, consider that all the physical CPU cores are closest to both accelerators. The programmer can use the following solution to assign unique OpenH physical CPU core IDs for binding the accelerator hosting Pthreads.

```
int nacc = openh_get_num_accelerators();
for (i = 0; i < nacc; i++) {
  int* closestPcpuids; int npcpus;
  openh_get_accelerator_pcpuaffinity(i,
      &closestPcpuids, &npcpus);
  openh_assign_acc_pcpuids(i,
      &closestPcpuids[i], 1);
  free (closestPcpuids);
}
```

In the code snippet above, the OpenH physical CPU core IDs closest to the accelerator i are obtained in the array, *closestPcpuids*, using the API function call, *openh_get_accelerator_pcpuaffinity()*. The accelerator i is assigned the OpenH physical CPU core ID at index i in the array, *closestPcpuids*. Hence, the accelerator with ID 0 is assigned the OpenH physical CPU core ID 0, and the accelerator with ID 1 is assigned the OpenH physical CPU core ID 0. The programmer may assign the OpenH physical CPU core ID at index $nacc - 1 - i$ in the array, *closestPcpuids* to binding the hosting Pthread for accelerator i. In this case, the accelerator with ID 0 is assigned the OpenH physical CPU core ID 15, and the accelerator with ID 1 is assigned the OpenH physical CPU core ID 14.

However, suppose the programmers choose to use the API functions, *openh_get_accelerator_lcpuaffinity()* and *openh_assign_acc_lcpuids()*, to assign unique OpenH logical CPU IDs that map to unique OpenH physical CPU core IDs for binding the accelerator hosting Pthreads. The solution to this problem involves considering the mapping scheme for OpenH logical CPU IDs to OpenH physical CPU core IDs. We describe one solution in the supplemental (Section IV).

VI. EXPERIMENTAL RESULTS AND DISCUSSION

We experimentally analyze the practical performance and dynamic energy consumption of three matrix multiplication applications, OpenH, OpenMP, and OpenACC, on our research hybrid server whose specifications are given in Table 1 (Topology given in Supplemental, Section II). The two Nvidia A40 GPUs are closest to all the cores (0-63) in the Intel Icelake multicore CPU of the hybrid server.

We employ system-level physical measurements using external power meters for component-level measurement of energy consumption. The measurements obtained this way are considered ground truth [25].

The hybrid server has one WattsUp Pro power meter between the wall A/C outlets and the node's input power sockets. The power meter captures the total power consumption of the node. It has a data cable connected to one USB port of the node. A Perl script collects the data from the power meter using the serial USB interface. The execution

TABLE 1. Specifications of the Intel hybrid server containing a single-socket Icelake multicore CPU and two Nvidia A40 GPUs.

Intel Platinum 8362 Icelake	
No. of cores per socket	32
No. of threads per core	2
Socket(s)	2
L1d cache, L1i cache	1.5 MiB, 1 MiB
L2 cache, L3 cache	40 MiB, 48 MiB
Total main memory	62 GB DDR4-3200
TDP	265 W
NVIDIA A40 GPU	
No. of GPUs	2
No. of Ampere cores	10,752
Total board memory	48 GB GDDR6 (with ECC)
Memory bandwidth	696 GB/sec
TDP	300 W

of these scripts is non-intrusive and consumes insignificant power. The power meters are periodically calibrated using an ANSI C12.20 revenue-grade power meter, Yokogawa WT210. The maximum sampling speed of the power meters is one sample every second. The accuracy specified in the data sheets is $\pm 3\%$. The minimum measurable power is 0.5 watts. The accuracy at 0.5 watts is ± 0.3 watts. The static power consumption of the server is 146 W.

To ensure the reliability of our results, we follow a statistical methodology where a sample average for a response variable (execution time and energy) is obtained from multiple experimental runs. The sample average is calculated by executing the application repeatedly until it lies in the 95% confidence interval and a precision of 0.05 (5%) is achieved. For this purpose, Student's t-test is used, assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions using Pearson's chi-squared test.

A. OPENH PARALLEL MATRIX MULTIPLICATION APPLICATION

The OpenH application comprises three hosting Pthreads leading the execution of three software components, {S_CPU,S_GPU1,S_GPU2}. It computes the matrix product, $C+ = A \times B$, where A, B, and C are square matrices of size $N \times N$. The application (where $p = 3$) is illustrated in Figures 5, 6 and 7. The workload size is $2 \times N^3$.

The accelerator hosting Pthreads leading the execution of the GPU components, {S_GPU1,S_GPU2}, are pinned to two different CPU cores (0 and 1) on the hybrid server. The software component *S_CPU* invokes the OpenBLAS DGEMM routine and is executed on the Intel multicore CPU using 62 OpenBLAS threads. It is pinned to the CPU cores, {2,3,…,63}. The accelerator components, $\{S_{GPU1}, S_{GPU2}\}$, are implemented using the thread function, *accCudaMxmComponent*, which invokes the CUBLAS DGEMM routine. We observed that the performance of this component is significantly better than the pure OpenACC implementation in *accNonCudaMxmComponent* (Figure 7). The experimental finding informs that pure OpenACC implementations have yet to match the performance of the hardware vendor's scientific libraries highly optimized for their accelerators.

The matrices A and C are partitioned horizontally among the three OpenH software components such that each component is assigned a number of rows of A and C proportional to its performance. The distribution of rows of A and C is provided in the array d. All the components share the matrix B.

B. PERFORMANCE FUNCTIONS AND WORKLOAD DISTRIBUTION

We present the workload partitioning algorithm to determine the workload distribution, d, optimizing the OpenH matrix multiplication application for performance.

The decomposition of the matrix A is computed using a model-based workload partitioning algorithm [26], [27]. The inputs to the algorithm are N and execution time functions of the three software components, $\{S_{CPU}, S_{GPU1}, S_{GPU2}\}$ of matrix dimension M. Each data point in a execution time function contains the execution time of the component computing a matrix product of two matrices of dimensions $M \times N$ and $N \times N$, respectively. The workload partitioning algorithm finds the partitioning, $d = \{d[0], \ldots, d[2]\}, d[0] + d[1] + d[2] = N$, of the workload of size N using the three software components to minimize the computation time of parallel execution of the workload.

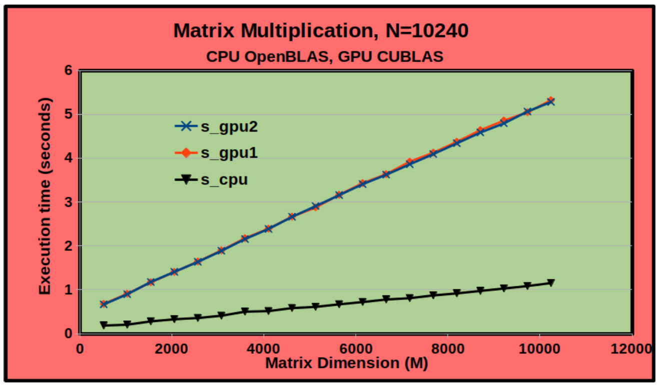
Therefore, the output by the algorithm is an array d of three elements, where $d[0]$, $d[1]$, and $d[2]$, respectively, contain the number of rows of A and C assigned to $\{S_{CPU}, S_{GPU1}, S_{GPU2}\}$.

We present the methodology to construct the execution time profiles of the three components ($\{S_{CPU}, S_{GPU1}, S_{GPU2}\}$). The execution time profiles are experimentally built using an automated build procedure employing a group of three hosting Pthreads. Figure 11 shows the execution time profiles. The workload size of a data point in a profile is $2 \times M \times N^2$ and is proportional to M since N is a constant.

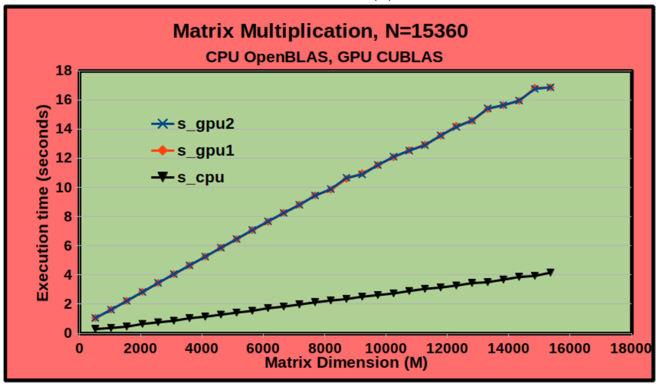
The execution times of all the components executing the same workload are measured simultaneously, thereby considering the influence of resource contention. The execution time for an accelerator component includes the time to transfer data between the host device and the accelerator. For example, the execution time of the execution of a workload on GPU_1 includes the time to transfer data from the host multicore CPU core to the accelerator and back and the computations on the accelerator.

C. PERFORMANCE OF OPENH PARALLEL MATRIX MULTIPLICATION APPLICATION

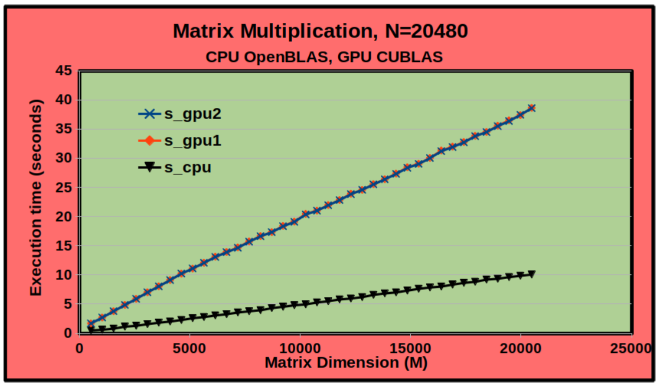
Figure 12 compares the execution times of three matrix multiplication applications experimented on our server platform (Table 1).



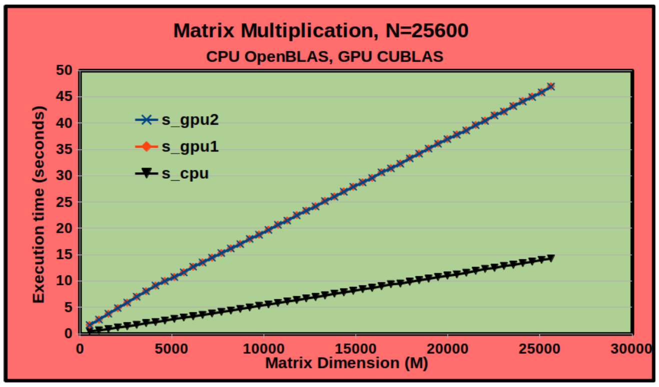
(a) Execution time versus matrix dimension M. $N = 10240$.



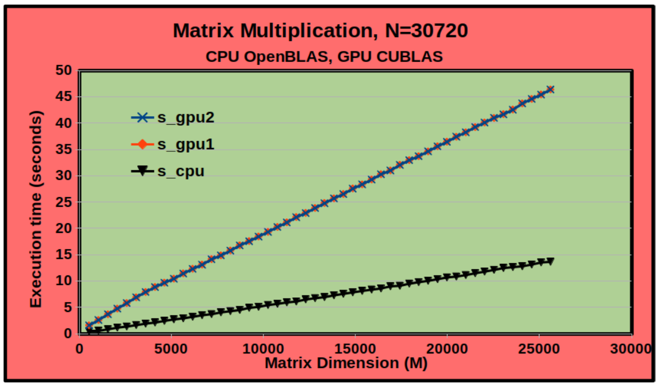
(b) Execution time versus matrix dimension M. $N = 15360$.



(c) Execution time versus matrix dimension M. $N = 20480$.



(d) Execution time versus matrix dimension M. $N = 25600$.



(e) Execution time versus matrix dimension M. $N = 30720$.

FIGURE 11. The execution time profiles of the three software components, $\{S_{CPU}, S_{GPU1}, S_{GPU2}\}$, executing the matrix multiplication application computing a matrix product of two matrices, respectively, of dimensions $M \times N$ and $N \times N$. The workload size of a data point in each figure is $2 \times M \times N^2$ and therefore is proportional to M since N is a constant. The CPU component employs the DGEMM routine provided by OpenBLAS 0.3.23.dev. The GPU components employ CUBLAS DGEMM provided by CUDA 12.0 toolkit. The profiles are linear functions of workload size. The performances of the two accelerator components are indistinguishable.

The matrix multiplication application (*openh_hybrid*) is the OpenH application presented in the Figures 6 and 7 comprising three software components, $\{S_{CPU}, S_{GPU1}, S_{GPU2}\}$.

The matrix multiplication application (*openmp_cpu_only*) is based on OpenMP only and employs only the multicore CPU. It executes the OpenBLAS DGEMM routine running 64 OpenBLAS threads equal to the total number of logical CPU cores in the server platform. Finally, the matrix multiplication application (*openacc_2_a40_gpus*) is based on OpenACC and employs only the two Nvidia A40 GPUs. The workload is divided equally between the GPUs since their performance functions are identical (Figure 11).

The OpenH matrix multiplication application performs the best since it employs the performance-optimal workload distribution and mapping of the software components to the CPU cores of the hybrid server. Furthermore, the performance-optimal workload distribution is the load-balanced distribution between the three software components [27], [28] since their execution times are linear functions of workload size.

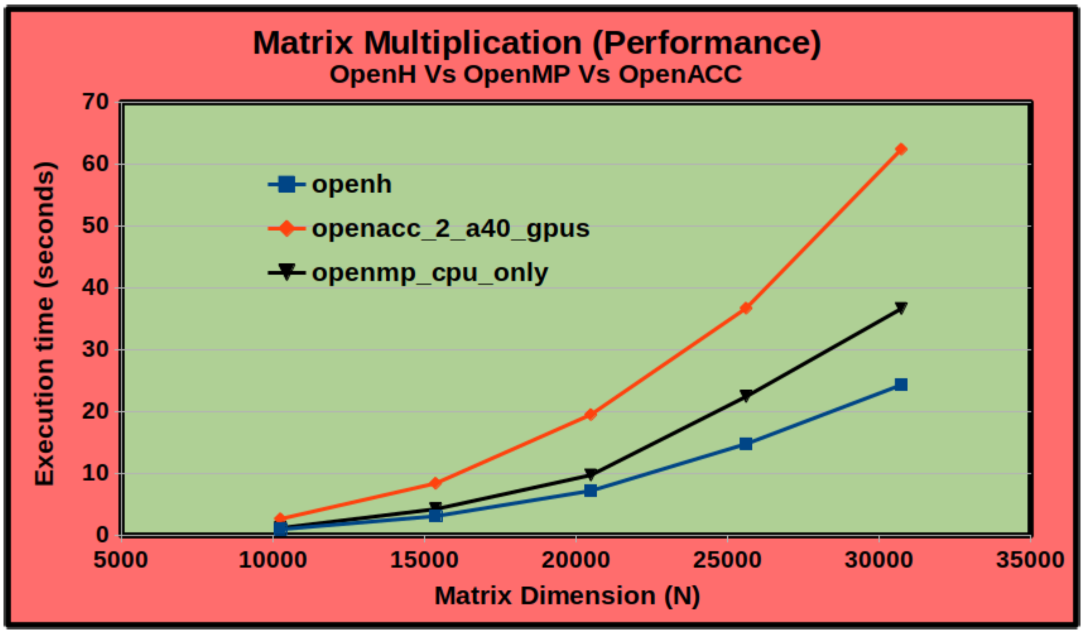


FIGURE 12. **The comparison of execution times of OpenH matrix multiplication application against OpenMP matrix multiplication application employing CPU only and OpenACC matrix multiplication application employing only the two GPU components. The matrix multiplication applications compute a matrix product of two matrices of dimensions $N \times N$ and $N \times N$. The performance of the OpenH matrix multiplication application is the best.**

D. ENERGY CONSUMPTION OF OPENH PARALLEL MATRIX MULTIPLICATION APPLICATION

Briefly, we introduce some terminology on energy consumption. The total energy consumption during an application execution is the sum of dynamic and static energy consumption. We define static energy consumption as the energy consumed by the platform without the application execution. Dynamic energy consumption is calculated by subtracting this static energy consumption from the total energy consumed by the platform during the application execution.

For the OpenH matrix multiplication application, we employ the workload partitioning algorithms proposed in [27], [28], and [29] to determine the partitioning of the matrices between A and C that minimizes the total dynamic energy consumption of the application. Specifically, the matrices A and C are partitioned horizontally among the three software components to minimize the application's total dynamic energy consumption.

The inputs to the algorithm are N and dynamic energy functions of the three software components, $\{S_{CPU}, S_{GPU1}, S_{GPU2}\}$ of matrix dimension M. Each data point in a function contains the dynamic energy consumption of the component computing a matrix product of two matrices, respectively, of dimensions $M \times N$ and $N \times N$. The workload partitioning algorithm finds the partitioning, $d = \{d[0], \ldots, d[2]\}, d[0] + d[1] + d[2] = N$, of the workload of size N using the three software components that minimize the total dynamic energy consumption of parallel execution of the workload. The output by the algorithm is an array d of three elements where $d[0]$, $d[1]$, and $d[2]$ contain the number of rows of A and C assigned to $\{S_{CPU}, S_{GPU1}, S_{GPU2}\}$, respectively.

The dynamic energy profiles of the three components are constructed using the additive approach [28]. The dynamic energy profiles are provided in the supplemental, Section III. In the additive approach, the dynamic energy profiles of the three components are constructed serially. The combined profile where the individual dynamic energy consumptions are totalled for each data point is then obtained. Then, the dynamic energy profile employing all the components in parallel is built. The difference between the parallel and combined dynamic energy profiles is observed. The parallel and combined profiles follow the same pattern, and the average difference between the corresponding data points in the profiles is around 2.5% (within the statistical accuracy threshold set in our experiments). Therefore, since the component profiles satisfy the additive hypothesis, we can use the dynamic energy profiles constructed serially as input to the workload partitioning algorithm for dynamic energy optimization.

The dynamic energy profiles (Section III, Supplemental) are linear functions of M. Therefore, the profiles are linear functions of the workload size since the workload size $2 \times M \times N^2$ of each data point is proportional to M (N is a constant). The authors [27] and [30] show that for the case of linear dynamic energy profiles that the workload distribution optimizing the application for dynamic energy has the total workload assigned to the most energy-efficient processor, the processor with linear energy function with the lowest slope (multicore CPU in this case).

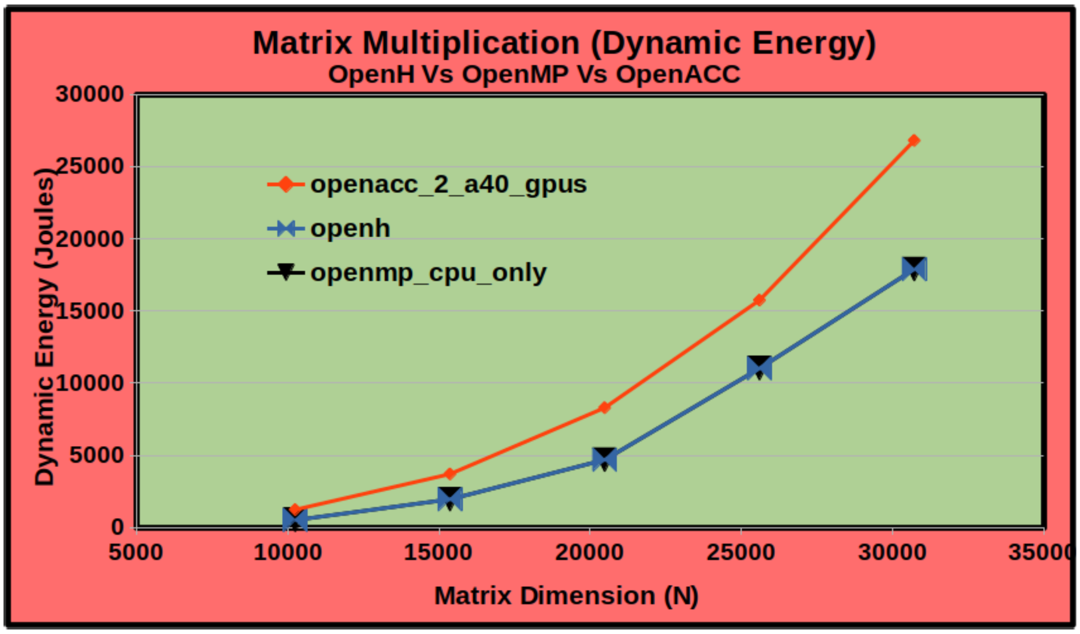


FIGURE 13. The comparison of dynamic energy consumptions of OpenH matrix multiplication application against OpenMP matrix multiplication application employing CPU only and OpenACC matrix multiplication application employing only the two accelerator components. The matrix multiplication application computes the matrix product of two matrices of dimensions $N \times N$ and $N \times N$, respectively. The OpenH application uses the energy-optimal workload distribution, which involves only the multicore CPU. Therefore, both OpenH and OpenMP applications consume the least dynamic energy.

Figure 13 compares the dynamic energy consumptions of the three matrix multiplication applications. The OpenMP matrix multiplication application (*openmp_cpu_only*) employs only the multicore CPU. It executes the OpenBLAS DGEMM routine running 64 OpenBLAS threads equal to the total number of logical CPU cores in the server platform.

The OpenACC matrix multiplication application (*openacc_2_a40_gpus*) comprises the GPU components, {S_GPU1,S_GPU2}, employing the two Nvidia A40 GPUs. The workload is divided equally between the GPUs. Moreover, the accelerator hosting Pthreads leading the execution of the GPU components are pinned to two different CPU cores (0 and 1) on the hybrid server.

The OpenH application employs the energy-optimal workload distribution that assigns the total workload to the most energy-efficient processor, the multicore CPU. Therefore, the OpenH application has one hosting Pthread that leads the execution of the CPU component, which executes the OpenBLAS DGEMM routine running 64 OpenBLAS threads equal to the total number of logical CPU cores in the server platform. It will have no accelerator components.

Both OpenH and OpenMP matrix multiplication applications consume the least dynamic energy.

VII. RELATED WORK

We start with an overview of research works on programming models and runtimes supporting hybrid parallel programming. Then, we present research works proposing hybrid parallel applications.

Research works (Diaz et al. [31], Fang et al. [32]) comprehensively survey the leading families of high-level programming models in heterogeneous many-core architectures. Therefore, we focus only on research works that propose extensions to the mainstream tools to support parallel programming on heterogeneous hybrid servers with a multicore CPU hosting multiple accelerators.

Komoda et al. [15] and Yan et al. [16] propose extensions to OpenACC and OpenMP to execute parallel loops across multiple accelerators (GPUs). The loop iterations are divided into chunks of equal size and executed across different GPUs. Matsumura et al. [33] propose a transpiler that takes an OpenACC code and generates a hybrid OpenMP and OpenACC code capable of multi-GPU execution. The outermost loop of OpenACC kernels is distributed equally between the different GPUs in the hybrid code. However, the host CPU is not considered one of the devices in the above research works.

Yan et al. [17] propose HOMP, an extension of OpenMP that automates the complex process of distributing the computations and data of parallel loops between CPUs and accelerators. HOMP offers language extensions, workload distribution algorithms, and runtime support to facilitate parallel loop execution across heterogeneous devices. While HOMP focuses on distributing the iterations of parallel loops, OpenH is based on the parallel execution of software components (kernels) that compose a hybrid program.

Cho et al. [18] propose an array programming interface to distribute parallel loops between different nodes and different devices inside each node using MPI and

OpenCL, respectively. Torres et al. [19] propose extensions of OpenMP to distribute workload between multiple devices. Kale et al. [20] propose extensions to OpenMP task constructs to schedule loop computations across multiple GPUs.

All the above research works focus on the homogeneous distribution of loop iterations across multiple GPUs using extensions of OpenMP and OpenACC.

Zhong et al. [34], [35] propose a novel hybrid parallel matrix multiplication application manually optimized for performance on a hybrid platform consisting of four single-socket AMD multicore CPUs and two Nvidia GPUs. The application comprises several CPU and GPU software components (kernels) executing in parallel. The CPU components employ the DGEMM routine offered by the AMD Core Math Library, and the GPU components invoke the CUBLAS DGEMM routine offered by the Nvidia CUDA library. The workload is divided heterogeneously between the components based on their functional performance models.

Xu et al. [14] employ OpenMP and OpenACC to develop hybrid applications for heterogeneous platforms with multiple GPUs. They propose extensions to OpenACC to support multiple GPUs. Synchronization between kernels executing on the devices is handled via special OpenACC directives using the host CPU. Therefore, the host CPU thread synchronizes updates to the kernels' shared data. Vitali et al. [36] use OpenMP and OpenACC to develop a hybrid molecular docking application employed in drug discovery.

While the research works reviewed previously propose language extensions to automate the parallel loop execution across heterogenous devices, the research works [14], [34], [35], [36] manually optimize the parallel applications on heterogeneous hybrid platforms for performance.

OpenH differs from the above research works in several ways. It is the first programming tool for developing portable parallel programs on heterogeneous hybrid servers composed of a multicore CPU and one or more different types of accelerators. It comprises a novel programming model and library API that allow programmers to get the configuration of the executing environment and bind the hosting Pthreads of the program to the CPU cores of the hybrid server to get the best performance.

OpenH employs Pthreads, OpenMP, and OpenACC seamlessly to address the limitations and challenges in state-of-the-art hybrid parallel program development. It provides a library API, which is missing in the prior art, to place and bind software components for optimal performance and energy. Moreover, OpenH allows programmers to employ heterogeneous workload distribution between the software components, optimizing the program for performance and energy. In addition, OpenH targets different types of accelerators. Finally, unlike the above research works, we also analyze the energy consumption of a hybrid matrix multiplication application based on OpenH.

VIII. CONCLUSION

Heterogeneous architectures featuring multicore CPU processors hosting multiple accelerators, such as GPUs, top the TOP500 and Green500 lists and dominate the computing landscape due to their superior performance and energy efficiency. However, developing parallel programs providing portable performance on such platforms remains challenging.

Vendor-specific programming tools such as CUDA and ROCm provide low-level APIs to the programmer to fine-tune their parallel programs to extract the best performance on a specific GPU architecture, Nvidia and AMD GPUs, respectively. These tools are combined with OpenMP to develop heterogeneous parallel programs. However, using these tools introduces vendor lock-in, hampering the portability of the programs.

Several vendor-agnostic high-level programming tools are proposed to simplify heterogeneous programming by providing high-level abstractions that automate handling low-level details and architecture-specific optimization without significantly sacrificing performance. They employ directive-based (OpenMP, OpenACC), C++-based (OpenCL, SYCL, OneAPI), and skeleton-based (SkelCL, SkePU) approaches.

However, the existing tools suffer from some limitations, which include a lack of compiler support for nested parallelism, performance portability, automatic heterogeneous workload distribution, user-friendly thread placement and processor affinity that are essential to the portable performance of hybrid programs.

In this paper, we proposed OpenH, a novel programming model and library API for developing portable parallel programs on heterogeneous hybrid servers composed of a multicore CPU and one or more different types of accelerators.

OpenH integrates Pthreads, OpenMP, and OpenACC seamlessly to develop hybrid parallel programs. An OpenH hybrid parallel program starts as a single *main* thread, creating a group of Pthreads called *hosting* Pthreads. A hosting Pthread then leads the execution of a software component of the program, either an OpenMP multithreaded component running on the CPU cores or an OpenACC (or OpenMP) component running on one of the accelerators of the server. The OpenH library provides API functions that allow programmers to get the configuration of the executing environment and bind the hosting Pthreads of the program to the CPU cores of the hybrid server to get the best performance.

OpenH differs from OpenCL in its design philosophy. *OpenCL* provides a generic unified interface to programmers to program the software components in the heterogeneous hybrid application for multicore CPUs and accelerators. While the unified framework allows OpenCL to provide maximum code portability for its applications, it does not hold dominant mainstream support since it competes with mainstream solutions widely used in each software component category. For example, OpenCL competes with

OpenMP, the most popular programming tool for developing software components for multicore CPUs, with CUDA and OpenACC for accelerator components for Nvidia GPUs.

OpenH, however, reuses the mainstream solutions and glues them seamlessly to facilitate the development of hybrid parallel programs. It provides a minimalistic layer of API functions on top of the mainstream solutions to provide both code and performance portability.

We illustrated the OpenH programming model and library API using two hybrid parallel applications based on matrix multiplication and 2D fast Fourier transform for the most general case of a hybrid hyperthreaded server comprising p computing devices.

We demonstrated the practical performance and energy consumption of OpenH for the hybrid parallel matrix multiplication application on a server comprising an Intel Icelake multicore CPU and two Nvidia A40 GPUs. Specifically, we compared the performance and dynamic energy consumption of the OpenH matrix multiplication application against the matrix multiplication application based on OpenMP, employing only the multicore CPU and matrix multiplication application based on OpenACC and employing only the two Nvidia A40 GPUs. The OpenH matrix multiplication application outperforms the other applications in terms of performance. Furthermore, the OpenH application, when optimized for dynamic energy, performs the best and consumes the least dynamic energy by employing the most energy-efficient processor (multicore CPU in this case).

Our OpenH library implementation is available at the URL [37].

REFERENCES

[1] (2023). *Top500 Supercomputers*. [Online]. Available: https://www.top500.org/lists/green500/2023/06/

[2] (2023). *Green500 Supercomputers*. [Online]. Available: https://www.top500.org/lists/green500/2023/06/

[3] *Nvidia Compute Unified Device Architecture (CUDA) Toolkit*. Accessed: Sep. 11, 2023. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[4] *AMD ROCm Open Software Platform*, Adv. Micro Devices, Inc. (AMD), Santa Clara, CA, USA, 2024.

[5] (Jul. 2013). *OpenMP Application Program Interface Version 4.0*. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

[6] *The OpenACC API Specification for Parallel Programming*. Accessed: Sep. 11, 2023. [Online]. Available: https://www.openacc.org/

[7] *OpenCL—Open Standard for Parallel Programming of Heterogeneous Systems*. Accessed: Sep. 11, 2023. [Online]. Available: https://www.khronos.org/opencl/

[8] *Heterogeneous Device Programming Using Sycl*. Accessed: Sep. 11, 2023. [Online]. Available: https://www.khronos.org/sycl/

[9] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, pp. 3202–3216, Dec. 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[10] D. S. Medina, A. St-Cyr, and T. Warburton, "OCCA: A unified approach to multi-threading languages," 2014, *arXiv:1403.0968*.

[11] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "RAJA: Portable performance for large-scale scientific applications," in *Proc. IEEE/ACM Int. Workshop Perform., Portability Productiv. HPC (PHPC)*, Nov. 2019, pp. 71–81.

[12] M. Steuwer and S. Gorlatch, "Skelcl: Enhancing opencl for high-level programming of multi-gpu systems," in *Proc. 12th Int. Conf. Parallel Comput. Technol.*, vol. 7979. Berlin, Germany: Springer, 2013, pp. 258–272.

[13] A. Ernstsson, L. Li, and C. Kessler, "SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems," *Int. J. Parallel Program.*, vol. 46, no. 1, pp. 62–80, Feb. 2018.

[14] R. Xu, S. Chandrasekaran, and B. Chapman, "Exploring programming multi-GPUs using OpenMP and OpenACC-based hybrid model," in *Proc. IEEE Int. Symp. Parallel Distrib. Process., Workshops Phd Forum*, May 2013, pp. 1169–1176.

[15] T. Komoda, S. Miwa, H. Nakamura, and N. Maruyama, "Integrating multi-GPU execution in an OpenACC compiler," in *Proc. 42nd Int. Conf. Parallel Process.*, Oct. 2013, pp. 260–269.

[16] Y. Yan, P.-H. Lin, C. Liao, B. R. de Supinski, and D. J. Quinlan, "Supporting multiple accelerators in high-level programming models," in *Proc. 6th Int. Workshop Program. Models Appl. Multicores Manycores*. New York, NY, USA: Association for Computing Machinery, 2015, pp. 170–180, doi: 10.1145/2712386.2712405.

[17] Y. Yan, J. Liu, K. W. Cameron, and M. Umar, "HOMP: Automated distribution of parallel loops and data in highly parallel accelerator-based systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2017, pp. 788–798.

[18] H. D. Cho, O. Kwon, and S. P. Midkiff, "HDArray: Parallel array interface for distributed heterogeneous devices," in *Languages and Compilers for Parallel Computing*, M. Hall and H. Sundar, Eds. Cham, Switzerland: Springer, 2019, pp. 176–184.

[19] R. Torres, R. Ferrer, and X. Teruel, "A novel set of directives for multi-device programming with OpenMP," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2022, pp. 401–410.

[20] V. Kale, W. Lu, A. Curtis, A. M. Malik, B. Chapman, and O. Hernandez, "Toward supporting multi-GPU targets via taskloop and user-defined schedules," in *OpenMP: Portable Multi-Level Parallelism on Modern Systems*, K. Milfeld, B. R. de Supinski, L. Koesterke, and J. Klinkenberg, Eds. Cham, Switzerland: Springer, 2020, pp. 295–309.

[21] *OpenMP Compilers & Tools*. Accessed: Sep. 11, 2023. [Online]. Available: https://www.openmp.org/resources/openmp-compilers-tools/

[22] *NVIDIA HPC Fortran, C and C++ Compilers With OpenACC*. Accessed: Sep. 11, 2023. [Online]. Available: https://developer.nvidia.com/hpc-compilers

[23] *GCC, the GNU Compiler Collection—GNU Project*. Accessed: Sep. 11, 2023. [Online]. Available: https://gcc.gnu.org/

[24] *OpenACC—GCC Wiki*. Accessed: Sep. 11, 2023. [Online]. Available: https://gcc.gnu.org/wiki/OpenACC

[25] M. Fahad, A. Shahid, R. R. Manumachu, and A. Lastovetsky, "A comparative study of methods for measurement of energy of computing," *Energies*, vol. 12, no. 11, p. 2204, Jun. 2019. [Online]. Available: https://www.mdpi.com/1996-1073/12/11/2204

[26] H. Khaleghzadeh, R. R. Manumachu, and A. Lastovetsky, "A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous HPC platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 10, pp. 2176–2190, Oct. 2018.

[27] H. Khaleghzadeh, R. R. Manumachu, and A. Lastovetsky, "Efficient exact algorithms for continuous bi-objective performance-energy optimization of applications with linear energy and monotonically increasing performance profiles on heterogeneous high performance computing platforms," *Concurrency Comput., Pract. Exper.*, vol. 35, no. 20, p. e7285, Sep. 2023.

[28] H. Khaleghzadeh, M. Fahad, A. Shahid, R. R. Manumachu, and A. Lastovetsky, "Bi-objective optimization of data-parallel applications on heterogeneous HPC platforms for performance and energy through workload distribution," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 543–560, Mar. 2021.

[29] H. Khaleghzadeh, M. Fahad, R. Reddy Manumachu, and A. Lastovetsky, "A novel data partitioning algorithm for dynamic energy optimization on heterogeneous high-performance computing platforms," *Concurrency Comput., Pract. Exper.*, vol. 32, no. 21, p. e5928, Nov. 2020.

[30] A. Lastovetsky and R. R. Manumachu, "Energy-efficient parallel computing: Challenges to scaling," *Information*, vol. 14, no. 4, p. 248, Apr. 2023.

[31] J. Diaz, C. Muñoz-Caro, and A. Niño, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 8, pp. 1369–1386, Aug. 2012.

[32] J. Fang, C. Huang, T. Tang, and Z. Wang, "Parallel programming models for heterogeneous many-cores: A comprehensive survey," *CCF Trans. High Perform. Comput.*, vol. 2, no. 4, pp. 382–400, Dec. 2020.

[33] K. Matsumura, M. Sato, T. Boku, A. Podobas, and S. Matsuoka, "MACC: An OpenACC transpiler for automatic multi-GPU use," in *Supercomputing Frontiers*, R. Yokota and W. Wu, Eds. Cham, Switzerland: Springer, 2018, pp. 109–127.

[34] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on heterogeneous multicore and multi-GPU systems using functional performance models of data-parallel applications," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2012, pp. 191–199.

[35] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on multicore and multi-GPU platforms using functional performance models," *IEEE Trans. Comput.*, vol. 64, no. 9, pp. 2506–2518, Sep. 2015.

[36] E. Vitali, D. Gadioli, G. Palermo, A. Beccari, C. Cavazzoni, and C. Silvano, "Exploiting OpenMP and OpenACC to accelerate a geometric approach to molecular docking in heterogeneous HPC nodes," *J. Supercomput.*, vol. 75, no. 7, pp. 3374–3396, Jul. 2019.

[37] S. Farrelly, R. R. Manumachu, and A. Lastovetsky. (2023). *OpenH: A Tool for Programming Portable Parallel Applications on Heterogeneous Hybrid Servers*. [Online]. Available: https://csgitlab.ucd.ie/manumachu/openh.git

SIMON FARRELLY is currently pursuing the B.Sc. degree (Hons.) with the School of Computer Science, University College Dublin.

RAVI REDDY MANUMACHU (Member, IEEE) received the B.Tech. degree from IIT, Madras, in 1997, and the Ph.D. degree from the School of Computer Science, University College Dublin (UCD), in 2005. He is currently an Assistant Professor with the School of Computer Science, UCD. His main research interests include high performance heterogeneous computing and energy-efficient computing.

ALEXEY LASTOVETSKY (Member, IEEE) received the Ph.D. degree from the Moscow Aviation Institute, in 1986, and the Doctor of Science degree from the Russian Academy of Sciences, in 1997. He is currently an Associate Professor with the School of Computer Science, University College Dublin (UCD). At UCD, he is also the Founding Director of the Heterogeneous Computing Laboratory. He authored the monographs *Parallel Computing on Heterogeneous Networks* (Wiley, 2003) and *High Performance Heterogeneous Computing* (Wiley, 2009). His main research interests include high performance heterogeneous computing and energy-efficient computing.