HOMP: Automated Distribution of Parallel Loops and Data in Highly Parallel Accelerator-Based Systems

Yonghong Yan and Jiawen Liu
Department of Computer Science and Engineering
Oakland University
Email: {yan, jliu}@oakland.edu

Kirk W. Cameron and Mariam Umar
Department of Computer Science
Virginia Polytechnic Institute and State University
Email: cameron@cs.vt.edu, mariam.umar@vt.edu

***Abstract*—Heterogeneous computing systems, e.g., those with accelerators than the host CPUs, offer the accelerated performance for a variety of workloads. However, most parallel programming models require platform dependent, time-consuming hand-tuning efforts for collectively using all the resources in a system to achieve efficient results. In this work, we explore the use of OpenMP parallel language extensions to empower users with the ability to design applications that automatically and simultaneously leverage CPUs and accelerators to further optimize use of available resources. We believe such automation will be key to ensuring codes adapt to increases in the number and diversity of accelerator resources for future computing systems. The proposed system combines language extensions to OpenMP, load-balancing algorithms and heuristics, and a runtime system for loop distribution across heterogeneous processing elements. We demonstrate the effectiveness of our automated approach to program on systems with multiple CPUs, GPUs, and MICs.**

***Keywords*-accelerator architecture; OpenMP; parallel loops; performance model; runtime system; data and computation distribution; alignment; load balance**

I. Introduction

Coprocessors or accelerators, e.g. NVIDIA GPUs and Intel Many Integrated Cores (MICs), incorporate specialized processing capabilities to handle particular tasks and to augment performance of conventional processors. Applications are commonly developed to offload a computational loop onto an accelerator while utilizing host processors for helper tasks such as data movement. Ideally, multiple accelerators and even the host processors could be utilized fully on demand for applications that can benefit from their combined computational capabilities. Unfortunately, typical implementations separate the memory space of these devices.

Moreover, programming constructs designed for processors and accelerators primarily support isolated computation across the devices. Most existing programming models support offloading a computation task to a single device a time. Co-scheduling of tasks on different devices is possible. It, however, needs significant manual efforts of programming for decomposing computation and input data, and for synchronization and merging the output data [32], [29], [13], [31]. This renders state-of-the-art node-level programming models, e.g. OpenMP [22], OpenACC [1], CUDA and OpenCL [16], unproductive for programming across multiple devices.

In this paper, we present HOMP (Hybrid OpenMP)[1] programming interfaces for automating the distribution of computation and data of parallel loops across multiple CPUs, GPUs and MICs within heterogeneous computing nodes. The contributions of this work are highlighted as follows.

1) We develop minimal language extensions to OpenMP for distributing data and computation of parallel loops onto multiple computation devices of same or different types. Those extensions allow users to specify the alignment of data and computation that uses the data to improve programmability. These extensions are compatible with the OpenMP standard, thus facilitating the migration of existing applications to use multiple devices collectively.

2) When distributing work between all devices within a computing node, one challenge is to achieve load balance among devices. We develop seven different algorithms for enabling partitioning of parallel loops at runtime to balance the use of computationally different resources. We also implement an approach of using cutoff ratio to automatically select devices for a parallel loop for the optimal performance.

3) We develop a runtime prototype to address challenges of hybrid execution within heterogeneous systems, including data and computation alignment and binding, mechanisms of loop scheduling and analytical modeling, runtime techniques for multi-target execution, and solutions for unified memory management of discrete and shared address space.

Performance analysis was performed using scientific kernels on a machine with multiple CPUs, GPUs and Intel MICs to study the effectiveness of loop distribution algorithms, and the performance of the HOMP implementation. The results indicate that by using a few HOMP directives, users can effectively use multiple architecturally different devices through hybrid execution of a parallel loop across those devices.

This paper is organized as follows: Section II introduces the background and motivation. Section III presents our language extensions to OpenMP. In Section IV, the loop distribution approaches and algorithms are illustrated. Section V describes our prototype implementation. We present our evaluation re-

[1]Available from https://github.com/passlab/homp

sults in Section VI. Section VII covers the related work and Section VIII concludes the paper.

II. Motivation

In an accelerator-based heterogeneous system, an accelerator, such as GPUs, operates in a different memory domain from the host processor. The use of accelerators employs an offloading execution model, an approach to shipping data and computation to another device for computation. The model is realized in programming languages through dedicated interfaces. For example, the OpenMP target construct allows users to annotate regions of code and data to be offloaded to an accelerator. In Figure 1, we include a simple AXPY example (vector addition) written in OpenMP.

```
void axpy_omp(REAL* x, REAL* y, int n, REAL a) {
  #pragma omp target device (0) map(tofrom: y[0:n]) \
        map(to: x[0:n],a,n)
  #pragma omp parallel for shared(x, y, n, a)
  for (int i = 0; i < n; ++i)
    y[i] += a * x[i];
}
void axpy_omp_mdev(REAL* x, REAL* y, int n, REAL a) {
  int ndev = omp_get_num_devices();
  #pragma omp parallel num_threads(ndev)
  { int devid = omp_get_thread_num();
    int start, size, remnant;
    remnant = n % ndev; size = n / ndev;
    if (devid < remnant) {
      size++; start = size*devid;
    } else start = size*devid+remnant;
    #pragma omp target device (devid) \
        map(tofrom: y[start:size]) \
        map(to: x[start:size],a,size)
    #pragma omp parallel for shared(x, y, size, a)
        for (int i = 0; i < size; ++i)
          y[i] += a * x[i];
  }
}
```

Fig. 1. AXPY OpenMP example for accelerators: The *axpy_omp* function shows how to offload the whole computation and two arrays to a single accelerator. The *axpy_omp_mdev* function demonstrates the use of OpenMP parallel and target constructs for offloading the same loop and data onto multiple accelerators by evenly partitioning the iteration and two arrays.

The offloading in OpenMP, which is similar to other models such as OpenACC, CUDA and OpenCL, etc, is performed in an all-or-none fashion for a single accelerator. Offloading onto more than one accelerators can be achieved by using the techniques illustrated in the *axpy_omp_mdev* function. In [32], the authors also demonstrated the use of OpenMP+OpenACC to achieve this. However, such an approach is a manual process that requires partitioning of both data and loop iterations, and making sure only necessary data are copied to the each accelerator associated with the assigned loop iterations. Moreover, such hand-tuning solutions do not adapt across multiple and different accelerators. To improve efficiency and performance, it is additionally necessary to automate this process to the greatest extent possible and to steer load balancing with consideration of the computational abilities of each device. Our work automates such process of distributing and aligning computation and data across host and accelerators using language extensions, loop distribution algorithms and runtime support.

III. Language Extensions for Distributing and Binding Computation and Data

In principle, using multiple accelerators for a single loop involves decomposing data and work, so each portion of data and work are offloaded to one device and the whole execution exploits multiple devices. Our design follows this principle. Figure 2 highlights the design using the AXPY example and these extensions are described as follows.

```
 1 /* align computation with data using ALIGN(x)*/
 2 void axpy_homp_v1(REAL* x, REAL* y, int n, REAL a) {
 3   #pragma omp parallel target device (*) \
 4         map(tofrom: y[0:n] partition([BLOCK])) \
 5         map(to: x[0:n] partition([BLOCK]),a,n)
 6   #pragma omp parallel for distribute \
 7         dist_schedule(target:[ALIGN(x)])
 8   for (int i = 0; i < n; ++i)
 9     y[i] += a * x[i];
10 }
11
12 /* align data with computation using ALIGN*/
13 void axpy_homp_v2(REAL* x, REAL* y, int n, REAL a) {
14   #pragma omp parallel target device (*) \
15         map(tofrom: y[0:n] partition([ALIGN(loop)])) \
16         map(to: x[0:n] partition([ALIGN(loop)]),a,n)
17   #pragma omp parallel for distribute \
18         dist_schedule(target:[AUTO])
19 loop:  for (int i = 0; i < n; ++i)
20     y[i] += a * x[i];
21 }
```

Fig. 2. AXPY using HOMP extensions. axpy_homp_v1: aligning computation with data; axpy_homp_v2: aligning data with computation.

1) Extension to the device *clause of the* target *directive:* The current OpenMP 4.5 syntax for device clause is device(scalar-integer-expression), which allows for only one device as a target for offloading. We extend it for supporting specifying multiple devices as offloading targets with the form device(device_specifier[[,device_specifier],...]). The device_specifier should be formed as initial_devid[[:nums][:dev_type_filter]], which lists the device IDs starting from initial_devid consecutively for nums number of devices that are of dev_type_filter type. The nums parameter, whose default value is 1 if not provided, should be either an integer expression or wildcard character *, which denotes all devices from the initial_devid. The following usages are legal device targets: device(0:*) for all devices; device(0, 2, 3, 5) for a list of devices; device(0:2, 4:2) which is the list of 0,1,4,5; and device(0:*:HOMP_DEVICE_NVGPU) which includes all the NVIDIA GPU devices.

2) Extension to the distribute *directive for distributing loop iterations among multiple devices:* The distribute directive in current OpenMP standard is used for distributing a loop among multiple teams of threads within a single device. We extend its semantics for allowing to specify the distribution of the loop iteration among multiple target devices as well. The usage of the directive is shown in line 6 and 17 of Figure 2. We also extend the dist_schedule clause of the distribute directive for specifying distribution policy between devices and between teams of each device by using either the target or teams directive name modifier.

The usage is shown in line 7 and 18 of Figure 2. Valid kinds for the dist_schedule(target:) clause are listed in Table I. The AUTO policy for loop distribution leaves to the runtime to determine the partition of a loop. Using this policy, a loop could be distributed among multiple devices in a way to achieve optimal load balance. For example, the sizes of loop chunks for each device, determined at runtime, could be proportional to their computation capabilities.

3) Extension to the map *clause to include an optional* partition *parameter for each mapped variable to support distribution of data among multiple devices:* Line 4, 5, 15, and 16 in Figure 2 show the usage of the partition parameter. Observing the fact that the range of an array in one dimension can be used for specifying the iteration range of a loop, we can use the same policies for loop and data distribution, which are listed in Table I. For distributing a multiple dimensional array, the extensions allow for specifying different policies in each dimension by using the syntax similar to multi-dimensional array declaration. Similarly, for distributing the iteration spaces of nested loops, users can specify policies for each loop.

The ALIGN policy provides an approach to bind an array subregion and a chunk of a loop iteration space so data can be automatically copied to the device that the loop chunk is assigned to. There are two ways to use this policy, 1) decompose data and then align the loop iteration with the data allocated for each device, and 2) partition the loop iteration space among multiple devices and then decompose data aligning with the computation. The *axpy_homp_v1* and *axpy_homp_v2* procedures in Figure 2 illustrate the usage of these two ways.

TABLE I
CURRENT DISTRIBUTION POLICIES

FULL	The full range of this dimension is used for distribution. This is the default policy if no policy is specified.
BLOCK	Divides the indices of a dimension evenly into contiguous blocks.
ALIGN(*dist*, *ratio*)	Aligns the distribution with the *dist* distribution provided as the clause arguments with ratio. Ratio default is 1, i.e. the two distributions are the same.
AUTO	Distributes the loop iterations with goal to achieve load balance. This policy only applies to loop distribution.

4) Introducing the parallel target *composite construct for specifying that a code region is to be executed on the target devices in parallel:* Our first extension for allowing multiple devices to be specified as offloading target does not require parallel offloading, thus offloading to the target devices could be serialized. The parallel target[2], used in line 3 and 14 in Figure 2, explicitly requires that data distribution, loop distribution and offloading computation should be performed in parallel by multiple host threads, thus providing an option to concurrently offload computation of a large scientific loop onto multiple devices.

A. Comparing with Related Work

A more complicated example, the Jacobi iterative kernel, is shown in Figure 3 using those extensions. In general, similar approaches of distributing data have been used in programming models for distributed systems. HPF [17] has the ALIGN directive to indicate elements (subarrays) of multiple arrays to be co-located in the same processors, which however, did not provide mechanisms to align computation with data. The Unified Parallel C(UPC) [8]'s upc_forall worksharing construct added the affinity field to the standard C for loop for specifying the correlation of loop iterations with UPC threads, an approach of distributing loop iterations among threads. UPC also provide interfaces to define array distribution using such policies as blocking or cyclic. The bindings between array subregions and loop partitions are formed implicitly through their links to the thread affinity. X10 [6] and Chapel [5] allow for array distribution among abstractions of memory segments (places in X10 and locale in Chapel). Launching an asynchronous task with an array subregion as the target location means to execute the task in a place where the subregion resides. They however, does not provide policy for specifying loop distribution.

OpenMP worksharing (parallel for) and distribute directives are designed to distribution parallel loops among threads of a team and teams of a league, respectively [22], both within the same memory space. OpenMP does not support data distribution or loop distribution among multiple devices.

Comparing with these efforts, our approach supports the distribution of both data and computation, as well as the alignment between them. It gives users more control of managing data and computation for heterogeneous systems. We realize these features through extensions to the OpenMP distribute and map clauses. The approach of viewing a loop iteration space and array dimension both as a region upon which a distribution can be applied offers an intuitive solution for binding data and computation, thus improving the productivity of using multiple accelerators.

The design of these interfaces and their semantics leverages the most recent OpenMP standard (4.5) with attempt to be intuitive to use, thus providing an easy migration path for existing OpenMP program to use multiple accelerators concurrently. Though we use OpenMP as baseline for extension, these interfaces, with modification, can be used with other programming APIs such as OpenACC. The principles are also applicable to other models, including runtime library, C++ template, meta-programming or domain specific languages.

B. The Challenges of Implementing HOMP Extensions

The designed extensions enable hybrid execution of parallel loops among multiple devices, There are however three challenges to implement those extensions: 1) to achieve load

[2]The target parallel combined construct of the current OpenMP standard is a shortcut for specifying a parallel construct immediately nested inside the target construct, which is for offloading a parallel region on one device.

```
#pragma omp parallel target data device(*) \
  map(to:n, m, omega, ax, ay, b,\
  f[0:n][0:m] partition([ALIGN(loop1)],FULL)) \
  map(tofrom:u[0:n][0:m] \
    partition([ALIGN(loop1)],FULL))\
  map(alloc:uold[0:n][0:m] \
        partition([ALIGN(loop1)],FULL) halo(1,))
while ((k<=mits)&&(error>tol)) {
#pragma omp parallel for target device(*) collapse(2)\
        distribute dist_schedule(target:[ALIGN(loop1)])
  for(i=0;i<n;i++)
    for(j=0;j<m;j++)
      uold[i][j] = u[i][j];

#pragma omp halo_exchange (uold)

#pragma omp parallel for target device(*)\
  reduction(+:error) \
  distribute dist_schedule(target:[AUTO])
loop1: for (i=0;i<n;i++) {
         if (i==0||i==n-1) continue;
         for (j=1;j<(m-1);j++) {
           resid = (ax*(uold[i-1][j] + uold[i+1][j])\
             + ay*(uold[i][j-1] + uold[i][j+1])\
             + b * uold[i][j] - f[i][j])/b;
           u[i][j] = uold[i][j] - omega * resid;
           error = error + resid*resid ;
         }
       }
  // the rest code omitted  ...
}
```

Fig. 3. Jacobi kernel for showing alignment of array and loop distributions as well as the use of AUTO policy

balance when distributing loop iterations across multiple computationally different devices, such as CPU, GPU and MICs, 2) to automatically schedule loop distribution and data movement (copy or share) so only the necessary data will be copied to the accelerators for the computation assigned to each device, and 3) to automatically select appropriate target devices for the optimal performance. We present our solutions to these challenges in the following two sections.

IV. Approaches to Loop Distribution for Achieving Load Balance

Illustrated in Section III, the distribute(target:AUTO) clause indicates to distribute the loop iteration among multiple devices (host and accelerators) so the computation could be load-balanced. In this section, we discuss three approaches we developed to support load-balanced hybrid execution.

A. Chunk Scheduling

Conventionally, loop scheduling on shared memory systems is the process of distributing iteration of a parallel loop to multiple threads [15]. The same approach can be applied to distribute loop among multiple accelerators and CPUs.

1) Static Chunking (BLOCK): It is beneficial to divide the work evenly among multiple devices of the same when the work performed by each iteration are the same. This approach is known as static chunk scheduling. Provided that each device computes at the same rate, all the devices should complete at the same time, thus achieving load-balance.

2) Dynamic Chunking (SCHED_DYNAMIC): Static chunking may not achieve good load balance when the work performed by each iteration varies. In dynamic chunking algorithm, after completion of its chunk, a device tries to acquire another chunk from the same loop. So in general, faster devices will likely perform more works, thus to achieve load-balanced distribution. The selection of the chunk size is critical for the load balance and it is a decision for tradeoffs between load-balance and chunking scheduling overhead.

3) Guided Chunking (SCHED_GUIDED): Guided chunking works similar to dynamic chunking, except that each device gets successively smaller sizes of chunks. In this approach, program execution starts with large chunk sizes and then chunks reduce in sizes as the computation close to finish, thus reducing the total amount of chunks and still maintaining good balance when it closes to finish.

B. Distributing Loop using Analytical Models

In this approach, analytical models are constructed to predict the throughput of each device for a loop, and then use the prediction to partition the loop iteration among multiple devices as well as the distribution of data associated with the loop chunks assigned to the devices. While using a full-fledged performance model such as those developed in [9], [12] and [3] will provide accurate prediction of the execution time, it will also incur large overhead to the application execution when the model is used at the runtime. Thus our models are specifically designed for handling computation-intensive parallel loops and are significantly simplified for the needs of loop distribution.

1) Analytical Model Considering only Computation Capability (MODEL_1_AUTO): For a given device i, the execution time T taken to compute a loop with N number of iteration can be represented as follows:

$$T = g_i(N) \tag{1}$$

For a given amount of time, the number of iteration that can be completed, i.e. throughput, is the reverse function of g_i, denoted as $f_i = {g_i}^{-1}$. Using N_i as the throughput for T time units, we have equation (2).

$$N_i = f_i(T), 0 \leq i < M \tag{2}$$

To distribute a loop with N iteration on to M number of devices, the objective of the model is to create M number of chunks of the loop, each computed by one device, and it takes the same amount of T_0 time to complete, theoretically. Thus we have equation (3).

$$N = \sum_{0 \leq i < M} N_i = \sum_{0 \leq i < M} f_i(T_0) \tag{3}$$

For data parallel loops in which each loop iteration contains the same amount of work (e.g. dense linear algebra), we can combine the Equation 2 and 3 to arrive a model that solves a linear system with M+1 variables, i.e. N_0, N_1, ..., $N_M - 1$, and T_0. In this model, all the devices compute possibly different sizes of loop chunks according to their computation capabilities and complete at the same time, T_0, which is the loop completion time.

2) Analytical Model Considering Both Computation and Data Movement Cost (MODEL_2_AUTO): To apply the basic analytical model to accelerators, the execution time of T

TABLE II
COMPARISONS OF LOOP DISTRIBUTION ALGORITHMS

Approaches	Algorithms	Notations used in Evaluation	# Stages	Overhead	Load Balancing	Descriptions	Related Work
Chunk Scheduling	Static Chunking	BLOCK	1	Low	Poor to good	Even distributions of iterations	[26], [27], [20]
	Dynamic Chunking	SCHED_DYNAMIC,2%	Multiple	High	Good	Each device receives chunks of same size	
	Guided Chunking	SCHED_GUIDED,20%	Multiple	High	Good	Each device receives chunk of different sizes	
Analytical Modeling	Compute-only Modeling	**MODEL_1_AUTO,-1,15%**	1	Low	Medium	Only considers computation in modeling	[25]
	Compute/Data Modeling	**MODEL_2_AUTO,-1,15%**	1	Low	Medium to good	Considers both computation and data movement	
Sample Profiling	Constant Sampling	**SCHED_PROFILE_AUTO,10%,15%**	2	Medium	Medium to good	Constant sample size for profiling	[14], [21], [28]
	Model-based Sampling	**MODEL_PROFILE_AUTO,10%,15%**	2	Medium	Medium to good	Uses models to select sample sizes for profiling	

Note: SCHED_PROFILE_AUTO,10%,15%: 10% is the chunk size (-1 if not used) and 15% is CUTOFF ratio. CUTOFF ratio is only applicable to the last four algorithms.

in Equation 1 for each device consists of both the cost of computation and data movement.

$$T = DataT_{dev} + ExeT_{dev} = g_i(N)$$

The parameters needed to solve the system in Equation 3, which could be linear or non-linear, include *DataT*$_{dev}$ and *ExeT*$_{dev}$ for each device. Although the model needs two parameters (*DataT*$_{dev}$ and *ExeT*$_{dev}$) for each device for distributing a parallel loop, it is actually the speedup of the particular loop on one device over another, e.g. CPUs vs GPUs, that determines the ratio of distributions of loop iteration.

$$\frac{DataT_{dev} + ExeT_{dev}}{ExeT_{host}} = \frac{DataT_{dev}}{ExeT_{host}} + \frac{ExeT_{dev}}{ExeT_{host}} \quad (4)$$

TABLE III
NOTATIONS USED IN MODELING

Notation	What it represents
ExeT$_{host}$	Execution time taken by host
ExeT$_{dev}$	Execution time taken by a device
DataT$_{dev}$	Data transfer cost, to and from a device
Flops	# of FLOPS of offloaded kernel on a device
Perf$_{host}$	Sustaining peak performance of the host
Perf$_{dev}$	Sustaining peak performance of a device
Size$_{data}$	Size of data moved from and to a device
Size$_{data}$	Size of data moved from and to a device
MemComp	Memory load/stores to computation ratio
DataComp	Data transfer to computation ratio
Flop$_{ssHost}$	Flops per second performance of host
Flop$_{ssDev}$	Flops per second performance of a device

For each of the parameters in the above speedup formula, the following assumptions and heuristics are applied:

- For the $DataT_{dev}$ parameter, the time to move data to and from a device, we use the Hockney's model, also known as "α-β model" [11]. It is a linear model of data sizes and the latency and bandwidth of memory systems.
- For $ExeT_{dev}$, each loop iteration has the approximately the same amount of work. For data parallel loops that contain no branching, this assumption is valid. Also due to the mechanisms of branch scheduling in GPU and MIC's SIMD architecture, which execute all the branches even there is divergence, the assumption holds as well.
- The $ExeT_{host|dev}$ parameter is computed using FLOPs/($Perf_{host|dev}$*MemComp), by assuming similar memory behaviors for the loop iteration across devices. For example, applications on GPUs that have highly coalesced memory access achieve similar effects of hiding memory access latency to how CPUs with multi-level cache memory hierarchy works. For each device, the **MemComp** parameter is the ratio of the amount of memory load/stores to the amount of computation.

Under these assumptions, the speedup formula is written as:

$$\frac{Size_{data} * MemComp}{FLOPs} * \frac{Perf_{host}}{Bandwidth} + \frac{Perf_{host}}{Perf_{dev}} \quad (5)$$

In this model, the $\frac{Size_{data}}{FLOPs}$, represents the kernel characteristics, i.e. the ratio of the amount of data to be moved to the amount of computation, denoted as **DataComp**. The use of **MemComp** and **DataComp** is similar to the use of a single parameter in roofline performance model [30]. Thus the first factor is reduced to $\frac{MemComp}{DataComp}$, which could be further reduced to $\frac{MemoryAccess}{DataTransfer}$. This is the ratio of the amount of memory access to the amount of data to be transferred. The second factor, $\frac{Perf_{host}}{Bandwidth}$, is the machine characteristics, i.e. the ratio of host CPU performance to the memory bus performance. The third ratio factor, $\frac{Perf_{host}}{Perf_{dev}}$, is the relative performance between CPU and GPU device, again a machine characteristics.

For a machine, the last two machine factors are constants, each of which is obtained through microbenchmark profiling in our experiment. The parameters in first ratio factor is collected through compiler analysis or direct user input. As an approximate model for loop distribution, the model does not take into consideration the differences of the performance impact to CPU and GPU from factors such as control-flow divergence, irregular memory access patterns (i.e., memory divergence), and register file pressure.

C. Loop Distribution based on Sample Profiling

In this approach, the system first computes a small amount of loop iterations on CPU and accelerators (GPUs and MICs) to determinate the throughput of each device for the loop (stage 1), and then distributes the remaining iterations according to the rate (stage 2). The selection of chunk sizes impacts the distribution of the remaining iterations. Large chunk sizes will provide more accurate profiling, but may also incur load-balance issues in stage 1. Smaller chunk sizes will reduce the profiling overhead and also incur less imbalance in stage 1, it however may not provide accurate profiling result for guiding the loop distribution in stage 2. By combing chunking scheduling and analytical modeling, we developed the following two profiling-guided approaches.

1) Constant Sample Sizes (SCHED_PROFILE_AUTO): In this approach, each device receives the same amount of loop iterations and compute in stage 1. After all devices finish the computation, profiling information will be broadcasted to each devices for computing the portions of iterations assigned to each device of the remaining iterations.

2) Sample Size Determined by Analytical Models (MODEL_PROFILE_AUTO): In this algorithm, we first distribute a small portion of the iterations using analytical model in stage 1. All the device compute and profile the execution. After all devices finish, profiling information will be broadcasted for computing their own portions from the remaining iterations.

D. Heuristics for Selecting an Algorithm

In Table II, the seven algorithms are summarized. The number of stages impact the runtime overhead since the more stages of loop distributions, e.g. in dynamic and guided chunking, the more runtime overhead would incurred. These overhead include cost from both runtime scheduling and data movement since more stages need more memory movement transactions. The selection of chunk sizes also play an important role for the effectiveness of chunk-based algorithms.

Thus it is important to provide heuristics for the system to automatically select an algorithm for an offloading kernel. To make the runtime lightweight, we use computational intensity based on the roofline model [30] to capture the computation and data movement behavior of an application to decide an algorithm to use. The heuristics are derived from experimental study as shown in Section VI.

E. Using CUTOFF Ratio for Selecting Offloading Devices

We observed that when offloading a parallel loop onto devices whose computational capability are significantly different, slower devices may contribute negatively to the overall performance. It is often that fewer faster devices deliver better performance than the combination of those faster devices and some slower devices since the additional overhead incurred by involving those slower devices are much higher than the contributions made by those devices. We address this by introducing a CUTOFF ratio parameter for the algorithms in modeling and profiling approaches. When distributing loop iterations, the runtime will not offload computation to a device whose the predicted contribution is less than the CUTOFF. In our experiment, CUTOFF ratio is selected as the average contribution by one device when considering all the devices are the same.

V. HOMP IMPLEMENTATION

The implementation includes both the compiler and runtime for the designed interfaces. The compiler generates multi-target kernel code and transforms the usage of HOMP syntax to runtime calls. The runtime system implements functionalities such as offloading kernel scheduling, loop distribution, data distribution, data mapping and movement, the seven loop distribution algorithms, system profiling and device managements. The library is designed to be portable and cross platform adaptations with different kind of architectures and memory systems. Loop scheduling framework is implemented modularly such that new scheduling algorithms can be easily added or tweaked for improved performance results.

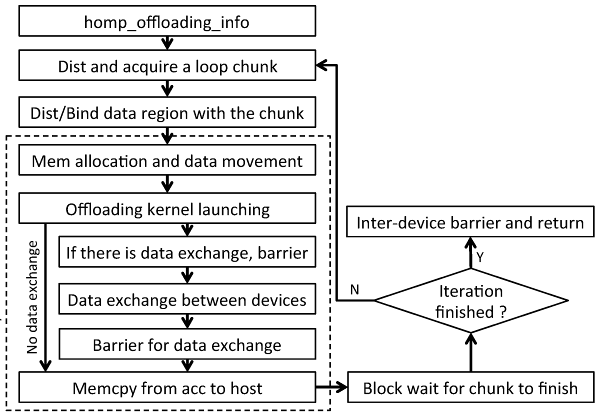
homp_offloading_info
Dist and acquire a loop chunk
Dist/Bind data region with the chunk
Mem allocation and data movement
Offloading kernel launching
If there is data exchange, barrier
Data exchange between devices
Barrier for data exchange
Memcpy from acc to host
Async call with stream
No data exchange
Inter-device barrier and return
Y
N
Iteration finished ?
Block wait for chunk to finish

Fig. 4. The Offloading Loop by A Device's Proxy Thread

When being initialized, the HOMP runtime reads from a given machine description file the specification of host CPU and accelerators. An accelerator can be specified to have either shared memory (if available) or discrete memory types, allowing runtime to choose either copying data or simply sharing data for data mapping. Each device (host, GPU or MIC) has a host side pthread as a proxy accepting offloading request. Such a request is represented as an homp_offloading_info object that contains information for data source pointers, dimension information of an array, data distribution policies, data mapping directions, offloading loop distribution policies, etc. A proxy thread performs operations such as array and loop distribution, memory allocation, data movement, launching the computation kernel, and other book-keeping tasks. The proxy pthread participates in the computation if it represents a host. For an accelerator, the proxy thread invokes asynchronous calls to interact with accelerators to perform data movement and launching tasks, maximizing the utilization of all devices. Figure 4 shows the offloading procedure by a proxy thread.

A. Multi-Target Code Generations

To allow maximum runtime adaptation when offloading a single code region to multiple and different targets, each code region is outlined into a separate function and transformed into multiple versions, one for each type of targets. For NVIDIA GPU, the ROSE compiler [19] is used for generating CUDA codes. For host, OpenMP worksharing loops are created from the original loop, and similarly for Intel MIC, Intel offloading directives are inserted into OpenMP code. The multi-target source codes are built into a fat binary.

B. LOOP Distribution

The proxy threads perform loop distribution and scheduling in parallel. Each thread calculates the subregion range of the loop iteration according to the global information stored in the homp_offloading_info object. For chunk scheduling algorithms, each proxy thread calculates the next chunk size and then picks a chunk from the remaining iterations using a compare-and-swap operation. For profiling-based algorithms, each proxy thread computes its first chunk and then broadcasts to all other proxies its own loop throughput calculated using the profiled timing. Each proxy thread then calculates the ratio of distribution for the remaining iterations using the throughput of all devices. For model-based algorithms, the proxy thread predict the execution time (T_0) first. They then estimate the costs of data movement of non-aligned array and the cost of transferring aligned array and the computation itself. These information are then aggregated together and be broadcasted to all the device threads. Each device thread then computes the number of iterations (N_i) and synchronizes with each other to make sure the whole range are properly distributed among the participating devices.

C. Data Mapping and Array Distribution

The HOMP runtime performs data mapping for both host CPUs and accelerators. CPU cores share host memory, and accelerators, e.g. NVIDIA GPU, each has its own device memory separated from host. NVDIA GPUs also supports unified memory between host and device, which can be considered as shared memory between host and accelerators. When mapping a data region from host memory to device memory, data are “shared” between host multiple CPU cores and/or GPUs that have unified memory enabled. The mapped data are “copied” between discrete memory spaces. If the data mapping semantics of the user program allow, the HOMP runtime makes mapping decisions (shared or copied) according to the memory types (shared or discrete) of the devices, thus optimizing and reducing unnecessary data movement.

For array references and loop iteration range by each device, the compiler transformations guarantee array references to its original array index spaces are properly translated to references to the array subregion that is mapped to each device. We use internal book-keeping variables and their corresponding runtime functions to keep track of the mapping.

Unified memory access between host and GPUs provides sharing memory semantics between host and device, hence no explicit data movement operations in programming is required. However, physical memories between CPUs and GPU are still separate, and data has to be transferred on demand through the PCIe bus, incurring much higher latency than accessing data from memory in accelerators. In our implementation, if not explicitly specified in the user program, we do not use this feature because of the observed poor performances of using unified memory as compared with explicit data movement (maximum of 10 and 18 times slowdown in our BLAS examples).

D. Computation and Data Alignment

When the ALIGN policy is used between loop iteration space and an array region, the runtime makes copies of the ranges of the alignees as the aligners’ ranges. For example, in the *axpy_homp_v1* function of Figure 2, for each target device, the iteration range of the loop (aligner) is a copy of the the array subregion of x or y since the distribution of array x or y is determined when the loop is encountered. For alignment in which multiple distributions form an inter-dependent alignment relationships, the runtime re-links those distribution so each aligner points to the root alignee’s distribution.

VI. Evaluation

We evaluate HOMP on a machine with two CPUs (Xeon E5-2699 Haswell), four NVIDIA K40 GPUs in two K80 dual-GPU cards, and two Intel Xeon Phi SC7120P for both performance and strong-scaling.

Representative scientific kernels are selected to evaluate the load-balance algorithms based on the range of the computation-data and computation-memory intensity of these kernels, and the patterns of computation. This allows the evaluation to cover a large range of data parallel algorithms for scientific applications being considered for offloading. Table IV summarizes these kernels. The *MemComp* and *DataComp* ratios represent the intensity each kernel impresses onto the memory and bus system with regards to the computation. The greater of the ratio, the more intensive it is.

TABLE IV
Benchmark Characteristics

	MemComp	DataComp	Characteristics
AXPY (N)	1.5	1.5	Data-intensive
Matrix Vector (NxN)	1+0.5/N	0.5+1/N	Compute-data balanced
Matrix Multiplication (NxN)	1.5/N	1.5/N	Compute-intensive
Stencil (NxN, 13 points)	0.5	1/13	Compute-data Balanced and neighborhood communication
Sum (N)	1	1	Data-intensive and reduction
Block Matching (NxN)	0.5	0.06	Compute-intensive and neighborhood communication

A. Evaluation on 4 GPUs

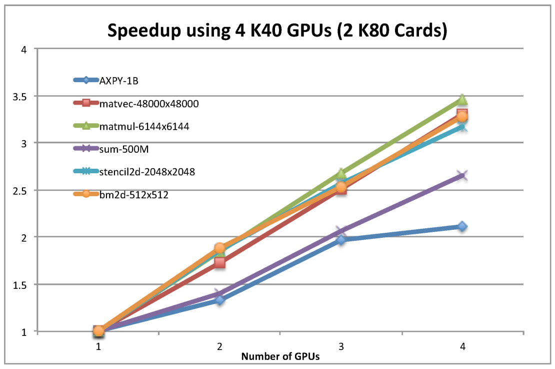

Fig. 7. Speedup using 2 K80 GPUs (Total 4 K40 GPUs)

We first experiment HOMP on four technically identical K40 GPUs to evaluate the scheduling algorithms for load balancing. The performance results are shown in Figure 5.

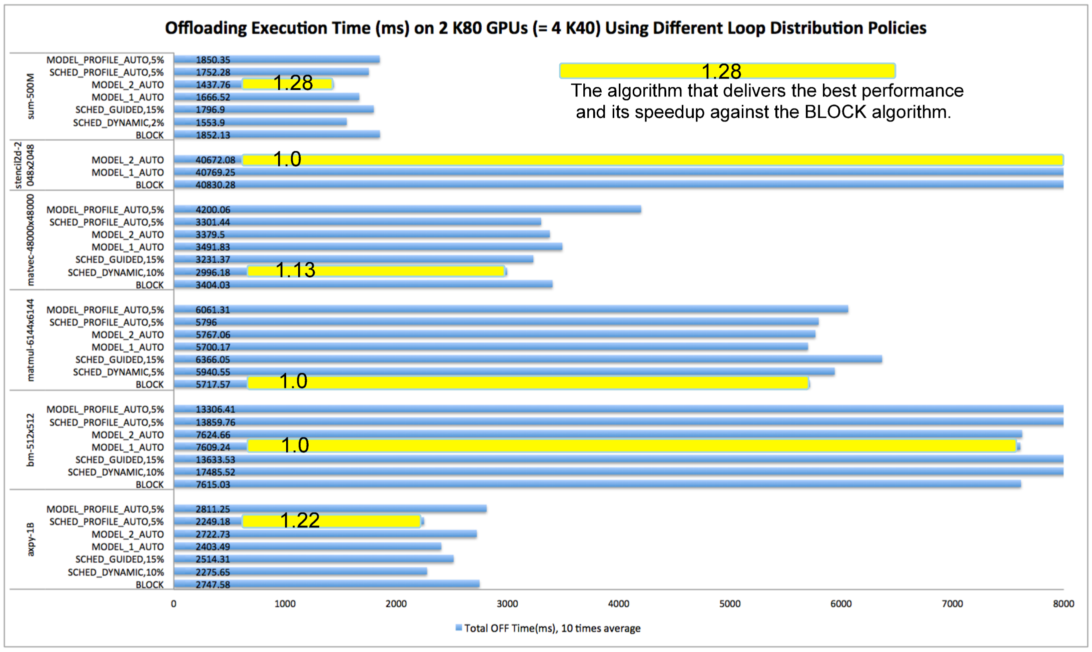

Fig. 5. Offloading Execution Time (ms) on 2 K80 GPUs (Total 4 K40 GPUs) Using Different Loop Distribution Policies

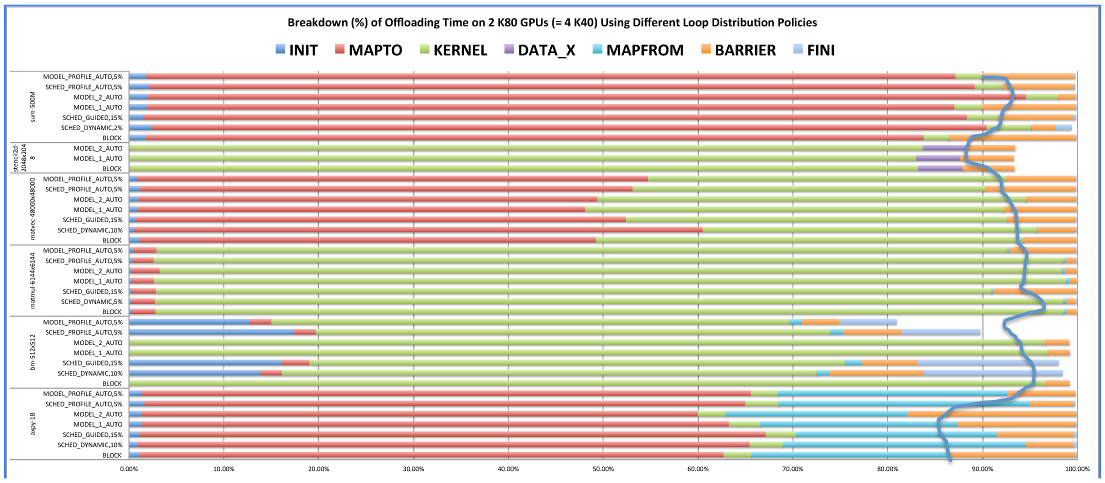

Fig. 6. Accumulated Breakdown (%) of Offloading Time on 2 K80 GPUs (= 4 K40) Using Different Loop Distribution Policies. The curve line crossing the figure shows the percentage of the incurred load imbalance which is below 5% in average.

Figure 6 shows the breakdown execution time in percentage among different operations of offloading and Figure 7 shows the speedup. Theoretically, BLOCK distribution, e.g. evenly distributing a loop among the same four accelerators, would yield good load balance, thus best performance. The results in Figure 5, however, indicate slightly differently. Computational-intensive kernels, i.e. mm, stencil and bm, deliver the best performance under the BLOCK policy. For the other three kernels(axpy, mv, sum), which are medium to highly data-intensive, the SCHED_DYNAMIC algorithm delivers better performance than using the BLOCK policy since it achieves overlapping of data movement and computation when scheduling multiple chunks to the same device. The accumulated breakdown percentage of the total execution time shown in Figure 6 indicate that most of the algorithms are able to schedule the loop with less than 5% overhead per device in average as the cost of barrier synchronizations. The results in Figure 7 show strong-scaling when using the four GPUs.

B. Evaluations on 2 CPUs + 2 MICs

We then experiment HOMP using 2 CPUs and 2 MICs for true hybrid and heterogeneous offloading. The results for the execution time are shown in Figure 8. For each kernel, CPU execution is handled using OpenMP, so no real data movement happens, and execution on MICs is handled using Intel MIC offloading mode. Practically, we would use peak performance as guideline to distribute loop iterations on CPU

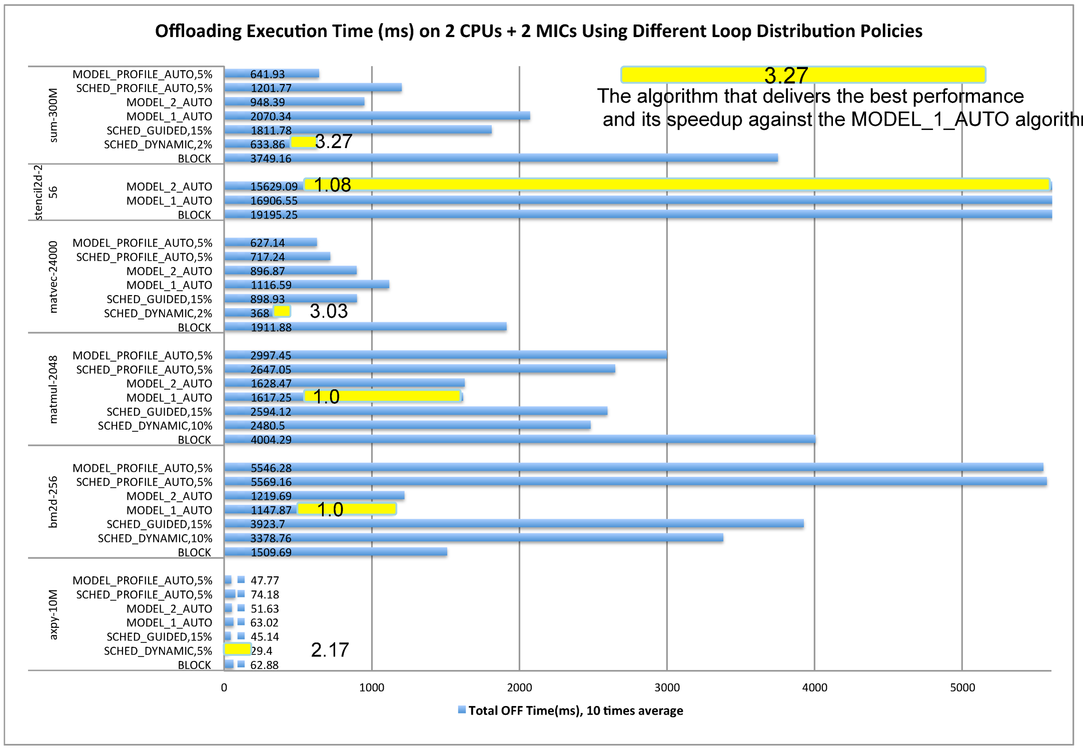

Fig. 8. Offloading Execution Time (ms) on 2 CPUs and 2 MICs Using Different Loop Distribution Policies

and MICs, which is our MODEL_1_AUTO algorithm. The results demonstrate the effectiveness of such an approach in computation-intensive kernels (mm, bm, and stencil). For other kernels, SCHED_DYNAMIC demonstrates to be an effective option. The breakdown percentage results, which are not included in this version of the paper, show average barrier overheads around 2% to 8% of the total execution time of each device, demonstrating the agility of the algorithms when scheduling loop iterations across heterogeneous devices.

TABLE V
SPEEDUP USING CUTOFF

Benchmarks	Devices after CUTOFF	CUTOFF Speedup
axpy-10M	2 CPU + 4 GPUs	1.35
bm2d-256	2 CPU + 4 GPUs	1.01
matul-6144	4 GPUs	2.68
matvec-48k	**4 GPUs**	0.56
stencil2d-256	4 GPUs	3.43
sum-300M	2 CPUs + 4 GPUs	2.09

C. Evaluations on 2 CPUs + 4 GPUs + 2 MICs

In Figure 9, we show the performance results using all the computational resources on the machine. We report the execution time of these kernels using different scheduling policy and the minimum execution time when 15% CUTOFF ratio is applied. The CUTOFF ratio is selected based on the average contribution by each of the device (100/7, considering 2 CPUs as one host device). In general, the results show that when computational resources vary significantly in performance, SCHED_DYNAMIC yields decent performance for most kernels. In Table V, it shows the speedup range from 0.5 - 3.x when 15% CUTOFF ratio is applied. It demonstrates the effectiveness of the algorithms to automatically select appropriate devices for the computation for optimal performance.

D. Evaluation Summary

The evaluation demonstrates the effectiveness of the load-balancing algorithms (average around 5% load imbalance) and also the heuristics of selecting loop distribution algorithms, which are as follows: 1. For computation intensive kernels, BLOCK should be the first choice for the same devices and MODEL_AUTO_1 will be the choice if devices are different because of the simplicity of the two algorithms. 2. For kernels with balanced data and computation, SCHED_DYNAMIC seems to work better for achieving overlapping of data movement and computation. 3. For data intensive kernels, MODEL_AUTO_2 is used since the mode takes into consideration of the data movement.

The evaluation also demonstrates the effectiveness of using CUTOFF heuristics for selecting device targets for offloading. If the contribution by one device is below the average when considering all the devices are the same, it could be removed from offloading targets.

VII. RELATED WORK

Our previous work [33] extended OpenMP target constructs with dist_data and dist_iteration clauses for distributing data

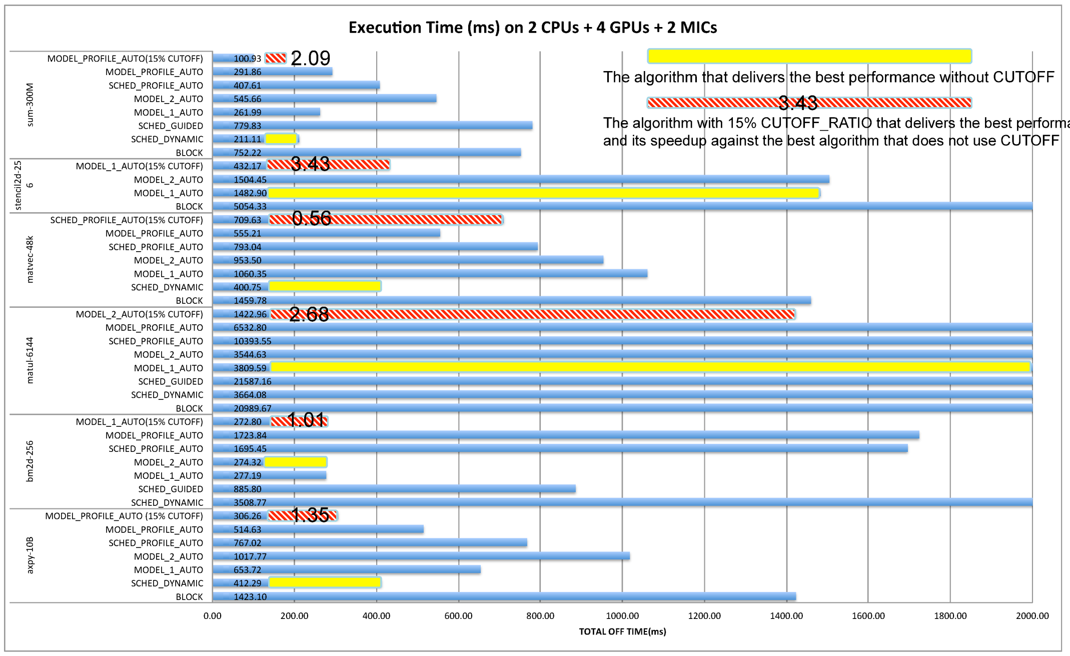

Fig. 9. Offloading Execution Time (ms) on 2 CPUs, 2 K80 GPUs and 2 MICs Using Different Loop Distribution Policies and Using CUTOFF_RATIO(%15)

and loops onto multiple accelerators, not including host. Only BLOCK distribution policy was introduced. The work presented in this paper provides support for data and computation alignment, AUTO loop distribution policy, load balancing algorithms and heuristics, and for automatically selecting devices for computations.

Ravi et. al, [26], [27] have developed work distribution and scheduling mechanism between CPU and GPU for iterative stencil type of applications that involve generalized reductions. It uses dynamic chunk scheduling algorithm (equal chunk size). Rashid et. al, in [14] used profiling-based algorithm for loop distributions in Concord C++ framework. Phothilimthana et.al, extends PetaBricks [23] with empirical auto-tuning and workstealing runtime for selecting optimal mapping of programs to devices to achieve load-balancing. It relies on static compiler analysis for data and computation alignments. Similarly, Lee et.al, in SKMD [18], Cabezas et.al, in [4] and Ramashekar et.al in [24] also use compiler analysis to obtain data boundary associated with computations assigned to a device. While this approach is appealing for users, it would be difficult for compiler to accurately derive optimal boundary when the application becomes complicated. Our approach introduces minimal language extension for users to explicitly specify data and computation distribution and alignment, which allows for more control to optimize data movement with computations.

Luk et al., in [21] use historical execution to project the execution time of a given problem sizes. Grewe et al. in [10] use machine learning approach to partition tasks statically, which could be categorized as model-based. These work were demonstrated using CPU and a single GPU only.

Ravi, Nishkam et al., in [25] presented an approach to split a parallel loop between CPU and an Intel MIC in a semi-automatic way using user-inserted cost models. Their approach relies on compiler analysis to identify boundary of data access, which may be challenging for more complicated applications.

Sundaresan et. al in [29] demonstrated a hand-tuning approach to distribute Jacobi kernels on hybrid CPU/GPU systems. While manual optimization may deliver better performance than the automated approaches including ours, this application-specific effort limits the usage of the approach and tools for other applications.

Scogland et.al. in [28] developed CoreTSAR runtime library for distributing loops by extending OpenACC with interfaces such as pcopy(mat[true:10:1][false:10]), hetero(TRUE), etc. It uses sample profiling algorithm for loop distribution.

Related work for runtime systems focusing on task scheduling address the load balance challenges through variants of workstealing, for example in [20], [7], and [2]. In comparison, our work extends OpenMP with minimum extensions to allow for easy migration. We implement most of the available algorithms in one system and design the modeling and CUTOFF ratio approaches to work more effectively with system with diverse computation devices.

VIII. Conclusion and Future Work

In this paper, we explore automating the process of simultaneous hybrid execution of parallel loops using all computing devices across memory domains in a heterogeneous system. Our work focuses on systems with multiple accelerators and a host CPU to test a prototype implementation of a runtime optimization infrastructure that exploits advanced parallel language extensions and loop distribution algorithms. We demonstrate quantitatively that significant performance improvements can be achieved by dividing and load balancing both computation and data automatically using methods that are inherently portable across architectures.

In the increasingly heterogeneous parallel systems, the importance of tapping all of the computational capabilities across memory domains while optimizing performance and maintaining code portability will grow substantially. With our prototype, we have demonstrated that software can improve our ability to leverage opportunities across memory domains to ensure performance continues to grow with the total number of computational engines available. Future enhancements include improving prediction models for more advanced computational engines, and more platform and code diversity experiments.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under Grant No. 1409946 and 1551182, and 1422712.

REFERENCES

[1] OpenACC. http://www.openacc-standard.org/.

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[3] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An adaptive performance modeling tool for gpu architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[4] Javier Cabezas, Lluís Vilanova, Isaac Gelado, Thomas B. Jablin, Nacho Navarro, and Wen-mei Hwu. Automatic execution of single-gpu computations across multiple gpus. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, 2014.

[5] Bradford L. Chamberlain, Steven J. Deitz, David Iten, and Sung-Eun Choi. User-defined distributions and layouts in chapel: Philosophy and framework. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, pages 12–12, Berkeley, CA, USA, 2010.

[6] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[7] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: An execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, pages 197–200. ACM, 2008.

[8] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003.

[9] D. Grewe, Zheng Wang, and M.F.P. O'Boyle. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *IEEE/ACM Code Generation and Optimization (CGO) Symposium*, 2013.

[10] Dominik Grewe and Michael F. P. O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *Proceedings of the 20th International Conference on Compiler Construction*.

[11] Roger W. Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel Computing*, 20(3):389 – 398, 1994.

[12] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009.

[13] Yulu Jia, Piotr Luszczek, and Jack Dongarra. Multi-gpu implementation of {LU} factorization. In *Proceedings of the International Conference on Computational Science, {ICCS} 2012*, pages 106 – 115, 2012.

[14] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshav Pingali. Adaptive heterogeneous scheduling for integrated gpus. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, 2014.

[15] David J. Liljab Kelvin K. Yuea. Parallel loop scheduling for high performance computers. In L. Grandinetti J.J. Dongarra, G.R. Joubert and J. Kowalik, editors, *High Performance ComputingTechnology, Methods and Applications*, volume 10 of *Advances in Parallel Computing*, pages 243 – 264. North-Holland, 1995.

[16] Khronos OpenCL Working Group. The OpenCL Specification - Version 1.0. Technical report, The Khronos Group, 2009.

[17] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA, USA, 1994.

[18] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. Skmd: Single kernel on multiple devices for transparent cpu-gpu collaboration. *ACM Trans. Comput. Syst.*, 33(3):9:1–9:27, August 2015.

[19] Chunhua Liao, Yonghong Yan, Bronis R de Supinski, Daniel J Quinlan, and Barbara Chapman. Early Experiences with the OpenMP Accelerator Model. In *OpenMP in the Era of Low Power Devices and Accelerators (IWOMP'13)*, pages 84–98. Springer, 2013.

[20] João V. F. Lima, Thierry Gautier, Nicolas Maillard, and Vincent Danjean. Exploiting concurrent GPU operations for efficient work stealing on multi-gpus. In *IEEE 24th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2012*.

[21] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 45–55, New York, NY, USA, 2009. ACM.

[22] OpenMP Architecture Review Board. The OpenMP API Specification for Parallel Programming. http://www.openmp.org/.

[23] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. Portable performance on heterogeneous architectures. In *The 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[24] Thejas Ramashekar and Uday Bondhugula. Automatic data allocation and buffer management for multi-gpu machines. *ACM Trans. Archit. Code Optim.*, 10(4):60:1–60:26, December 2013.

[25] Nishkam Ravi, Yi Yang, Tao Bao, and Srimat Chakradhar. Semi-automatic restructuring of offloadable tasks for many-core accelerators. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, 2013.

[26] Vignesh T. Ravi and Gagan Agrawal. A dynamic scheduling framework for emerging heterogeneous systems. In *18th International Conference on High Performance Computing, HiPC 2011, Bengaluru, India, December 18-21, 2011*, pages 1–10, 2011.

[27] Vignesh T. Ravi, Wenjing Ma, David Chiu, and Gagan Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 137–146, New York, NY, USA, 2010. ACM.

[28] Thomas R. Scogland, Wu-Chun Feng, Barry Rountree, and Bronis R. Supinski. Coretsar: Adaptive worksharing for heterogeneous systems. In *Proceedings of the 29th International Conference on Supercomputing - Volume 8488*, ISC 2014, pages 172–186, 2014.

[29] Sundaresan Venkatasubramanian and Richard W. Vuduc. Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 244–255, New York, NY, USA, 2009. ACM.

[30] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[31] Noah Wolfe, Tianyu Liu, Christopher Carothers, and Xie George Xu. Heterogeneous concurrent execution of monte carlo photon transport on cpu, gpu and mic. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '14, pages 49–52, Piscataway, NJ, USA, 2014. IEEE Press.

[32] Rengan Xu, Sunita Chandrasekaran, and Barbara Chapman. Exploring programming multi-gpus using openmp and openacc-based hybrid model. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pages 1169–1176. IEEE Computer Society, 2013.

[33] Yonghong Yan, Pei-Hung Lin, Chunhua Liao, Bronis R. de Supinski, and Daniel J. Quinlan. Supporting multiple accelerators in high-level programming models. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, pages 170–180. ACM, 2015.