Gecko: Hierarchical Distributed View of Heterogeneous Shared Memory Architectures

Millad Ghane
Department of Computer Science
University of Houston
TX, USA
mghane2@uh.edu

Sunita Chandrasekaran
Department of Computer and Information Sciences
University of Delaware
DE, USA
schandra@udel.edu

Margaret S. Cheung
Physics Department
University of Houston
Center for Theoretical Biological Physics, Rice University
TX, USA
mscheung@central.uh.edu

Abstract

The November 2018 TOP500 report shows that 86 systems in the list are heterogeneous systems configured with accelerators and co-processors, of which 60 use NVIDIA GPUs, 21 use Intel Xeon Phi cards, one uses AMD FirePro GPUs, one uses PEZY technology, and three systems use a combination of NVIDIA GPUs and Intel Xeon Phi co-processors. From a software standpoint, managing data locality on such heterogeneous systems is as important as exploiting parallelism in order to achieve the best performance. With the advent of novel memory technologies, such as non-volatile memory (NVM) and 3D-stacked memory, there is an urgent need for effective mechanisms within programming models to create an easy-to-use interface that addresses such memory hierarchies. It is also equally crucial for applications to evolve with data locality for the expression of information. In this paper, we propose *Gecko*, a novel programming model that addresses the underlying memory hierarchy topology within computing elements in current and future platforms. *Gecko*'s directives distribute data and computation among devices of different types in a system. We develop a source-to-source transformation and a runtime library to efficiently manage devices of different types to run asynchronously. Although our current implementation of *Gecko* targets Intel Xeon CPUs and NVIDIA GPUs, it is not restricted to such architectures. We used SHOC and Rodinia benchmark suites to evaluate *Gecko*. Our experiments used a single node consisting of four NVIDIA Volta V100 GPUs. Results demonstrated scalability through the multi-GPU environment. We observed 3.3 times speedup when using multiple GPUs.

PMAM'19 , February 17, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6290-0/19/02…$15.00
https://doi.org/10.1145/3303084.3309489

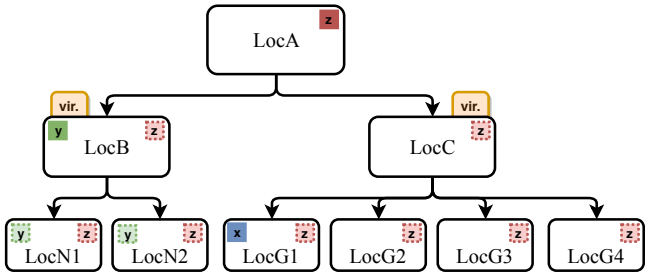

Figure 1. An overview of our proposed model, *Gecko*, with 9 locations. The solid boxes represent variables in our program. The dotted boxes show locations with access to those variables. The *"vir." tags* show virtual locations in *Gecko*.

CCS Concepts • **Computer systems organization → Heterogeneous (hybrid) systems**; **High-level language architectures**;

Keywords Hierarchy, Heterogeneous, Portable, Shared Memory, Programming Model, Abstraction.

ACM Reference Format:
Millad Ghane, Sunita Chandrasekaran, and Margaret S. Cheung. 2019. *Gecko:* Hierarchical Distributed View of Heterogeneous Shared Memory Architectures. In *The 10th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'19), February 17, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3303084.3309489

1 Introduction

Heterogeneity has been the primary source of computational performance in modern high performance systems [3, 15, 16] since the loss of conventional improvements (Dennard scaling) [6, 26]. Due to the advances in semiconductor manufacturing, GPUs [13] and MICs (Many Integrated Cores) have been widely adopted in the design of high performance computing (HPC) systems. The trend of such design prevails in the latest ORNL's supercomputer, Summit, that contains six NVIDIA Volta V100 GPUs and only two IBM POWER9 processors per node. Future exascale computing nodes are expected to embrace such heterogeneity that grow with the number of computational devices [3], which makes utilization of all available resources a challenging task.

The aforementioned heterogeneity has become complicated as we face challenges imposed by the memory wall [3]. Recent advances in memory technologies have led to dramatic changes in their hierarchy. The inclusion of novel memory technologies, such as non-volatile memory (NVM) [19] and 3D-stacked memory [18, 27], has complicated the memory hierarchy (registers, caches, DRAM). As the design factors grow, the complexity in both the hardware and the software requires a performance-friendly approach. The complexity is exacerbated when multiple devices with different hardware types are utilized to parallelize the code. It is particularly problematic when a diverse memory technology is implemented on GPUs through Processing-In-Memory (PIM) [1, 25]. Consequently, we are in dire need of a simple and robust model to efficiently express the implicit memory hierarchy along with the parallelization opportunities available in the scientific application [26].

In this paper, we present *Gecko*[1], a novel programming model and runtime system that represents the hierarchical structure of memory and computational resources in current and future systems. *Gecko* addresses multi-device heterogeneous platforms and provides opportunities for applications to add and remove memory and computational resources at the execution time.

Gecko provides *a distributed view of the heterogeneous shared memory system to the applications* with one exception: **a variable defined in a particular location is accessible by that location and all of its children, while the same variable remains private with respect to its parent**. This exception helps applications to improve their data locality by bringing data closer to where it is supposed to be processed, and prevent side-effects of data sharing (i.e., false sharing). Developed as a directive-based programming model on top of OpenACC [22], *Gecko* consists of a set of directives that helps applications to declare devices, allocate memories, and launch kernels on different devices. We build on the work of Ghane et al. [12], which was a simpler approach towards supporting heterogeneity in future systems.

Here are the contributions of this work:

- We propose *Gecko*, a novel programming model that addresses the underlying memory hierarchy topology within computing elements.
- We present a prototype of *Gecko* and integrate it with a directive-based programming model, OpenACC. This approach does not impact the directive-based model itself (Section 4).
- We present the work distribution and runtime policies that *Gecko* currently supports (Section 4).
- We evaluate *Gecko* by using the case studies from SHOC and Rodinia benchmark suites and demonstrate the impact on performance (Sections 5 and 6).

[1]Accessible from: https://github.com/milladgit/gecko

2 Methodology: the *Gecko* model

The principal constructs of the *Gecko* model are *locations*. Locations are an abstraction of storage and computational resources available in the system and are represented as a node in *Gecko*'s tree-based hierarchy model. Similar to the Parallel Memory Hierarchy (PMH) model [2], *Gecko* is a tree of memory modules with workers at the leaves. Workers will perform computations, and they attach to the memory modules. Figure 1 illustrates an example of our *Gecko* model. This model represents a regular cluster/supercomputer node with two non-uniform memory access (NUMA) multicore processors, *LocNi*, and four GPUs, *LocGi*, similar to NVIDIA's PSG cluster [21] and ORNL's Titan supercomputer [24].

The location hierarchy in *Gecko* designates how one location shares its allocated memories with another. When a memory is allocated in a location, the allocated memory is accessible by its children. They will have access to the same address as their parent. However, the allocated memory is not shared with their parent and is considered to be a private memory with respect to their parent. Figure 1 shows how hierarchy will affect memory accesses among locations. The allocated memory y is allocated in Location *LocB* and consequently, it can be accessed by *LocB*, *LocN1*, and *LocN2*. However, *LocA* has no knowledge of the variable y. With the same logic, allocated memory z can be accessed by all locations, while accesses to x are only limited to *LocG1*.

In *Gecko*, locations are categorized by one of the followings: (1) memory modules, (2) memory modules with a worker, and (3) *virtual* locations. Memory modules are annotated with their attributes (like type and size); *LocA* in Figure 1 is a memory module. If a location is a memory module with a worker attached to it, the location will be used to launch computational kernels by the runtime library; *LocNi* and *LocGi* are examples of memory modules with workers. Finally, the virtual locations, *LocB* and *LocC* in Figure 1, are neither memory modules nor computational ones. They are an abstract representation of their children in the hierarchy. Grouping a set of locations under a virtual location provides a powerful feature for the application to address the location set as a whole. Similar approaches have been observed in related work [28, 29]. Like other location categories, virtual locations can also be the target of memory allocation and kernel launch requests. Depending on the type of requests and hints from scientific developers, the runtime library acts upon the requests and performs them at the execution time.

Locations are abstractions of available resources in the system. Any location in *Gecko*, internal or leaf locations, is possibly the target of a kernel launch by application. Virtual locations, however, provide flexibility to the applications. With virtual locations, applications aptly fix the target of their kernel to that location while changing the underlying structure of the tree for the same location. As a result, the application targeted for a multicore architecture dynamically

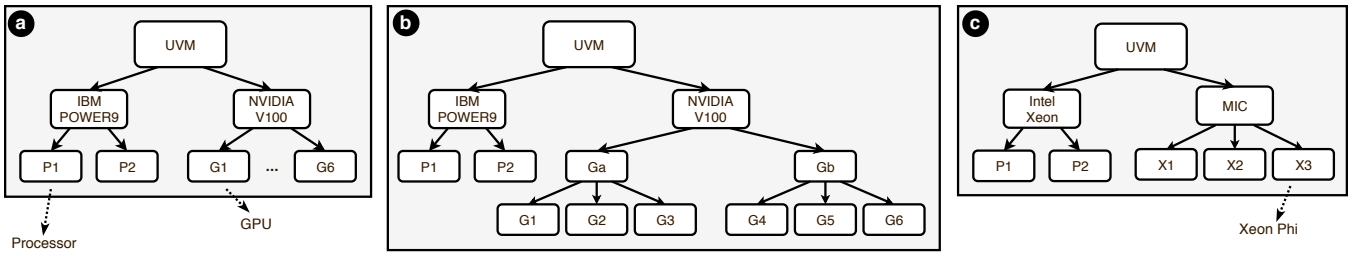

Figure 2. *Gecko*'s model representing various system. ORNL's Summit (*a* and *b*) with two IBM POWER9 processors and six NVIDIA Volta V100 GPUs – Tianhe-2 (*c*) with two Intel Xeon processors and three Intel Xeon Phi co-processors

or statically morphs into a program targeting different types of accelerators (e.g., NVIDIA GPUs, AMD GPUs, or FPGAs).

Similar to Hierarchical Place Trees (HPT) [29], *Gecko* provides facilities to represent a physical machine with different abstractions [28, 29]. The best configuration depends on the characteristics of the application, its locality requirements, and its workload balance. The developer or auto-tuning libraries can assist *Gecko* in choosing the effective abstraction. Figure 2 shows how *Gecko* represent nodes in Summit [23] and Tianhe-2 [17]. Model ⓐ is the most generic approach to represent Summit. The two IBM POWER9 processors in two different sockets form a NUMA domain, and all six NVIDIA Volta V100 GPUs are represented under a virtual location.

Detailed information on Summit reveals that the first three GPUs form a spatial locality with respect to each other while the last three ones show the same spatial locality to each other. They are connected to the main processors and each other with an NVLink [11] connection with a bandwidth of 100 GB/s (bidirectionally). As shown in ⓑ, applications are able to utilize this locality by declaring two virtual locations, `Ga` and `Gb`. Such an arrangement minimizes the interference between the two GPU sets. With this model, applications run Kernel K_a on `Ga` and Kernel K_b on `Gb` to fully utilize all resources and perform simultaneous execution of kernels while minimizing data bus interferences.

Gecko is a platform- and architecture-independent model. The hierarchy in ⓒ represents a system targeting Intel Xeon Phis (e.g., Tianhe-2) with *Gecko*. Xeon Phis are grouped together and declared under their parent location, `MIC`. In cases where an application faces a diverse set of accelerators (for instance, a node equipped with NVIDIA GPUs and another node with Intel Xeon Phis) and they are unknown to the application at compile time, *Gecko* adapts to the accelerator in the system without any code alterations. *Gecko* also is able to adapt to changes in the workload and employ more resources, in case they are needed to expedite the processing, by modifying the hierarchy.

3 Key Features of *Gecko*

This section is dedicated to key features that make *Gecko* superior to other flat models.

3.1 Dynamic Hierarchy Tree

Gecko's hierarchy tree is *dynamically* composed at the execution time. Unlike Sequoia [9] and HPT, the *Gecko*'s hierarchy is not fixed at compile time. An application defines the whole tree at the execution time and adds or removes other branches to or from the hierarchy as the application progresses. *Gecko* reacts to the changes in the workload size with this dynamic behavior by inserting more resources and removing them accordingly. This feature also enables applications to adapt themselves to the workload type. For applications that benefit from multicore architectures, like traversing a linked list, the hierarchy only utilizes multicore processors instead of accelerators.

3.2 Memory Allocation Algorithm

Uncertainty in location type makes memory allocation a challenging problem. The allocation process has to be postponed to execution time since only then the location is recognized. Algorithm 1 lists the allocation algorithm that *Gecko* uses to allocate memory. Depending on the location, as discussed before, a memory can be private or shared among a set of locations. This introduces a scoping mechanism on the variables in the system.

Algorithm 1 begins by recognizing if the location chosen is a leaf node in the tree or not. A leaf node is the most private location in the hierarchy. The memory allocated by a leaf node is only accessible by itself. Based on the type of the location, whether a processor or a GPU, the corresponding API function, `malloc` or `cudaMalloc`, is called.

On the other hand, if the location targeted is not a leaf location, *Gecko* traverses the subtree beneath the chosen location and determines whether all children locations are multicore or not (`children.areAllMC()`). If all children locations are multicore, like Location *LocB* in Figure 1, we will use a memory allocation API [8, 10] for the host (like `malloc`, `numa_alloc` [14], and so on). However, if only one of them is from a different architecture, like Location *LocG* in Figure 1, *Gecko* will allocate memory on the unified memory domain.

Unfortunately, the unified memory allocations are universally visible, and their visibility cannot be limited to a subset of devices. Consequently, the memory allocation requests in

Algorithm 1 Memory Allocation Algorithm

Input: *gTree*: *Gecko*'s hierarchical tree structure.
Input: *loc*: the target *Location*.
Output: Memory Allocation API.

```
1: function memAlloc(gTree, loc)
2:     allocFunc ← NULL                    ▷ Chosen Allocation API
3:     if gTree.isLeaf(loc) then
4:         if gTree.getType(loc) == HOST then
5:             allocFunc ← malloc
6:         else if gTree.getType(loc) == GPU then
7:             allocFunc ← cudaMalloc
8:         end if
9:     else
10:        children ← gTree.getChildren()
11:        if children.areAllMC() then
12:            allocFunc ← malloc
13:        else
14:            allocFunc ← cudaMallocManaged
15:        end if
16:    end if
17: end function
```

LocG and *LocA* of Figure 1 will eventually result in calling the same API function despite their inherent difference in our proposed model. A potential limitation mechanism by the CUDA library that would shrink the visibility scope of the unified memory allocations would address this shortcoming in *Gecko*.

In real applications, the hierarchy tree is not necessarily hard-coded in the source code allowing the application to transform and adapt to changes in the environment. Thus, *Gecko* provides interfaces to dynamically add new subtrees (and locations) within the hierarchy or remove them from the tree in an arbitrary manner. As a result, neither the location types nor the locations nor the hierarchy may be known prior to execution. Consequently, the memory allocation process is not straightforward and becomes challenging.

3.3 Minimum Code Changes

Gecko's hierarchical tree leads to minimum source code alterations. Applications are able to introduce an arbitrary number of virtual locations to the hierarchy at the execution time and reform themselves based on the availability of the resources. This provides a great opportunity for the single-code base approach. Figure 3a is another representation of the model in Figure 1: the same configuration with an extra virtual location, *LocV*. The dotted lines represent potential relationships between locations. Such relationships have not been finalized by the application yet. The new virtual location, *LocV*, acts like a handle for applications. Applications launch their parallel regions in the code on this location while knowing nothing about the hierarchy structure beneath *LocV*. At the execution time, an application is able to switch between the potential subtrees deliberately.

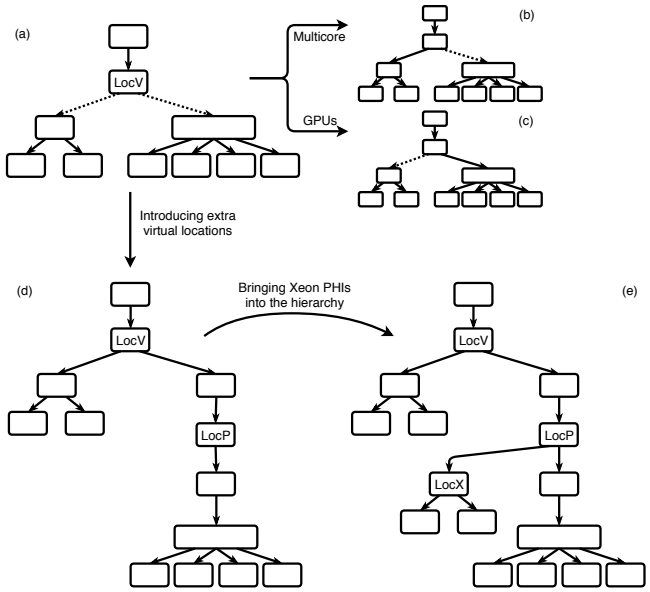

Figure 3. Polymorphic capabilities of *Gecko* leads to less source code modifications. We can change the location hierarchy at run time. Our computational target can be chosen at runtime: processors (*b*) or GPUs (*c*). *Gecko* also supports deep hierarchies in order to provide more flexibility to applications (*d*). We are able to extend the hierarchy as workload size changes (*e*).

By announcing *LocB* or *LocC* as *LocV*'s child, kernels that are launched on *LocV* will be executed on a multicore or multi-GPU architecture, respectively. This shows how *Gecko* adapts to different architectures by a simple change in the association among the locations in the hierarchy.

The structure of hierarchy can be extended arbitrarily in *Gecko*. Figure 3d shows an equivalent model for the configuration in Figure 3a. We have introduced three new virtual locations to the model on the right branch of *LocV*. Such changes to the model do not affect the workload distribution in any way since virtual locations do not possess any computational resources. The workload submitted to *LocV* is simply passed down to its children according to the execution policy (discussed in Section 4). Virtual locations are also effective in helping an application adapt to changes in the environment or workload. Suppose an application has already allocated four GPUs and wants to incorporate two new Intel Xeon Phis to the hierarchy tree due to a sudden increase in workload. The application defines a virtual location, *LocX*, and declares the Xeon Phis as its children. Then, by declaring *LocX* as the child of *LocP*, *Gecko* is able to incorporate the Xeon Phis into the workload distribution. Hereafter, the computational workloads that were previously distributed on four GPUs under *LocP* will be distributed among the four GPUs and the two newly added Xeon Phis. Figure 3e shows the new model with two Xeon Phis included in the hierarchy. Later,

one can remove the *LocX* location from the tree and return to Figure 3d.

Gecko offers a *location coverage* feature that helps extend the adaptation capabilities of virtual locations. Location coverage makes a virtual location represent all resources with the same location type. In many cases, the number of locations of a specific type are unknown until the execution time. Although the application is not aware of the number of available resources of such type, it is looking for all available ones. The *location coverage* feature brings relief to developers and makes the code more portable and robust to new environments.

4 Design and Implementation

This section discusses the prototype of *Gecko*. We developed a Python script that takes a Gecko-annotated source code as an input and creates an output source that fully conforms to the C++ standard and the OpenACC specifications. Since our model is developed as a language feature, it can also be extended to the other languages, like C and Fortran. Figure 4a shows the compilation framework that is used to compile a *Gecko*-annotated source code. After transforming code to OpenACC, we will set the compiler's flag to generate code for both multicore and GPU. During the execution time, Gecko will choose the device (multicore or GPUs) accordingly.

4.1 Directive-based Extensions

Gecko provides facilities for memory operations and kernel execution. With *Gecko*'s directives, applications are able to declare locations, perform memory operations (allocation, free), run kernels on different locations, and wait on kernels to finish. Directives provide a level of flexibility that library-based approaches do not necessarily provide. Directives also requires users to most of the times add fewer additional lines to the code thus not increasing the Lines of Code (LOC) by a large number.

Figure 4b displays a Gecko-annotated source code that implements the model in Figure 4d and runs a simple kernel on it. Each numbered part in the code shows how the *Gecko*'s directive and its clauses are utilized. The following paragraphs discuss each of the numbered parts in the code in details.

Part ❶ declares the type of locations that our application will target, which will be used within the location definition clauses later. These clauses are annotated with `loctype` and specify the resources required by our application. Every location type is named uniquely so that it can be easily accessed throughout the application's lifetime. Their names are user-defined arbitrary names. For instance, the first line declares the `uProc` location type, which is an Intel Skylake 64-bit architecture with 16 cores and 4 MB L2 cache. Such details provide more insight about the resources available to *Gecko*, leading to better decision-making strategies at the execution time. We have reserved the "`virtual`" name to represent the *virtual* locations in *Gecko*.

Part ❷ shows how to define the locations with the `location` clause. Each location is defined by a unique name and a location type. Their location types should be from the list of the previously declared types by `loctype`. The first location definition corresponds to the `LocA` with `SysMem` as its location type. This leads to declaring `LocA` as our main memory (DRAM). `LocB`, `LocN`, and `LocG` are *virtual* locations. Locations `LocN1` and `LocN2` are defined as host processors for a system with two separate processors. And, *all* GPUs attached to the current node are named as `LocGs[i]`. In cases that we are not aware of available resources from the same location type, we can ask *Gecko* to allocate all of them with `all` keyword as used in the last line of Part ❷.

Part ❸ defines the hierarchical relationship among locations in Figure 4d. Every `hierarchy` clause accepts a `parent` and `children` keyword to represent the relationship among locations. The relationship between the parent and its children is established with a '+' sign, or it is broken with a '-' sign. The first line declares that `LocA` is the parent of `LocB`. The second line shows how `LocN1` and `LocN2` have `LocN` as their parent. In case of uncertainty regarding available locations, one can use the *all* keyword, similar to defining location, as shown in the third line. Finally, the last line establishes the relationship between `LocG` and `LocB`, which results in defining `LocG` as the only child of `LocB`. `LocN` and its subtree structure are reserved for future involvement in load-balancing strategies. Such a design, defining `LocB` as the virtual location, allows us to keep the source code intact and it lets the hierarchy beneath `LocB` to be a complex one.

Statements in Parts ❶ ❷ ❸ can be declared within an external configuration file despite the hard-coded approach in our example code. This brings a degree of freedom and flexibility to *Gecko*, similar to the Sequoia model. However, unlike Sequoia, *Gecko*'s configuration file represent the available resources and not the mapping strategy of tasks to resources. The application does not need any recompilation, and any changes in the configuration file will affect the load distribution in the application. Figure 4c shows a configuration file for *Gecko* that replicates the same configuration in the sample code. Corresponding parts are numbered accordingly for the configuration file. One can ask *Gecko* to load the configuration from a file at execution time with the **`#pragma gecko config file`** statement.

Part ❹ shows how the hierarchy could be adjusted at the execution time. If targeting GPUs do not benefit the application in terms of speedup, we can drop the GPU subtree from the hierarchy and attach the local processors (CPUs) to it, as shown in the code within the `if`-statement. `LocG` is no more `LocB`'s child as it is removed and `LocN` becomes the new child of `LocB`.

Until now, we have only described the system and the resources that our application are targeting. Lines in Part ❺

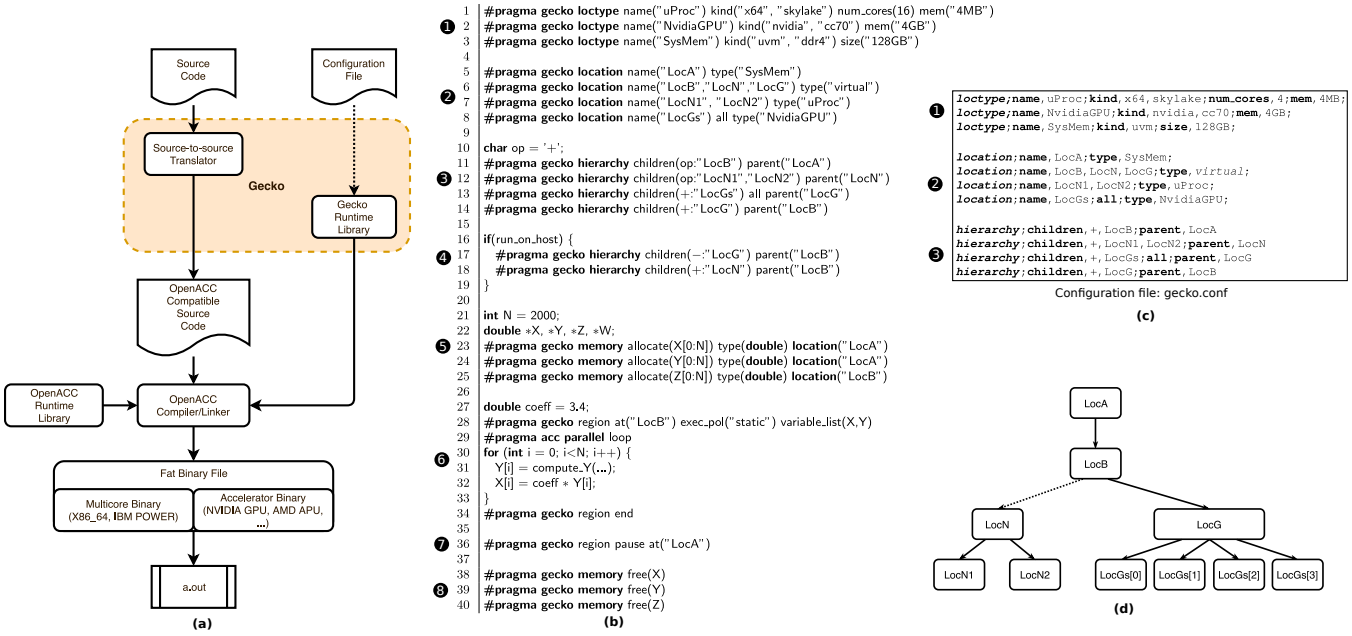

Figure 4. a) Diagram of *Gecko*'s runtime and source-to-source translator. b) A sample source code annotated with the *Gecko*'s directives. c) A sample of configuration file that resembles the first three sections of the source code (*a*). d) An overview of defined hierarchy in the source code (*b*) and configuration file (*c*).

show how the memory is allocated in *Gecko*. For every memory allocation request, application specifies the data type, size, and location. As an example, lines in this part allocate memories with N elements in locations `LocA` and `LocB`. Every allocation request follows the memory allocation algorithm, as described in Algorithm 1.

Launching a computational kernel in *Gecko* is realized by guarding the OpenACC parallel regions. With the `at` keyword, the application specifies the destination location that the parallel region will be launched on. Based on the execution policy, *Gecko* splits the loop and distributes the load among locations within the hierarchy. With `exec_pol`, application chooses the distribution policy (as discussed in Section 4.2) to distribute the iteration space among computational resources. The chosen policy governs the strategy to split the loop into slices and then assign each loop slice to a computational resource under the chosen location. Similar to the `present` keyword in OpenACC, the application should determine the list of used variables inside the region with the `variable_list` keyword. Part ❻ shows an example of how to use *Gecko* to launch a kernel.

Synchronization in *Gecko* is supported with a `pause` keyword as shown in the only line of Part ❼. To provide finer granularity, the application should specify the location on which *Gecko* should wait. This line will put a barrier inside the application until all the locations under the chosen location, `LocA` in this case, are finished with their assigned task. Finally, previously allocated memories are freed with the `free` keyword on `memory` clause as shown in Part ❽.

Gecko follows similar algorithm as the memory allocation algorithm to find the proper API call to free already allocated memory space.

4.2 Distribution Policies

Gecko distributes workload based on the inputs from the application. Workloads with different characteristics demand different distribution approaches. Within locations, *Gecko* follows the OpenACC's decision to assign number of `gangs` and `vectors`. Currently, *Gecko* supports five distribution policies: `static`, `flatten`, `percentage`, `range`, and `any`[2]. The `static` policy traverses the tree in a top-down fashion and splits iterations evenly among children, however, `flatten` splits iterations by the number of leaf locations. The `percentage` policy splits the iteration space based on the percentages that are given as input to the policy. Similar to `percentage`, `range` splits iterations based on the arbitrary input numbers by the application. Finally, the `any` policy assigns all the iterations to a single idle location that is chosen randomly at the execution time.

5 Experimental Setup

Our experimental setup is the NVIDIA Professional Services Group (PSG) cluster [21]. PSG is a dual socket 16-core Intel Haswell E5-2698 v3 at 2.30GHz with 256 GB of RAM. Four NVIDIA Volta V100 GPUs are connected to this node through PCI-Express bus. Each GPU has 16GB of GDDR5 memory.

[2]Please refer to https://github.com/milladgit/gecko for a detailed explanation and an example for each policy.

Table 1. List of benchmarks ported to *Gecko* - A: Number of kernels in the code. B: Total kernel launches. SP: Single Precision - DP: Double Precision - int: Integer - Mixed: DP+int

Application	Source	Input	Data Type	A	B
vector add	-	200,000,000 elements	DP	1	20
stencil	SHOC	2048 x 2048 matrix	DP	2	40
bfs	Rodinia	One million-edge graph	Mixed	5	39
cfd	Rodinia	missile.domn.0.2M	Mixed	5	9
gaussian	Rodinia	2048 x 2048 matrix	SP	3	6141
hotspot	Rodinia	1024 data points	DP	2	20
lavaMD	Rodinia	10x10x10 boxes	Mixed	1	1
lud	Rodinia	2048 data points	SP	2	4095
nn	Rodinia	42764 elements	SP	1	1
nw	Rodinia	2048 x 2048 data points	int	4	4095
particle filter	Rodinia	1024 x 1024 x 40 particles	Mixed	9	391
pathfinder	Rodinia	width: 500,000	int	1	99
srad	Rodinia	2048 x 2048 matrix	Mixed	7	12

We used CUDA Toolkit 9.2 and PGI 18.4 (community edition) to compile the OpenACC and CUDA codes, respectively.

Please note that we tried multiple avenues to run experiments on a system that connects GPUs via NVLink. Getting access to the Amazon and Google cloud instances that uses V100 and NVLink was not straightforward, let alone the costs involved. We were able to use Summit at ORNL only very briefly as the system is undergoing testing and maintenance. Preliminary Summit results were inconclusive, thus we could not draw a summary.

We successfully port benchmarks from the Rodinia suite and SHOC to *Gecko* by annotating their source codes with *Gecko*'s proposed directives[3]. Table 1 shows a list of benchmarks used in this paper. The table also shows their input, data type, the total number of kernels, and the total kernel launches at the execution time. The annotation process is as follows: (1) Application asks *Gecko* to load the configuration file; (2) Every `malloc`'ed memory is replaced with a `memory allocate` clause in the code; (3) All OpenACC parallel regions are guarded with a `region` clause; (4) All OpenACC's `update, copy, copyin,` and `copyout` clauses in the code are removed; (5) A `pause` clause is placed at arbitrary locations in the code to ensure the consistency of algorithm; and finally, (6) All *free* functions for memory deallocations are replaced with `memory free` clauses. These are the necessary modifications required for any code to use *Gecko*.

We create a configuration file for Figure 4d for a node in PSG. Location `LocN`, however, is defined to have only a single child, instead of two. In all the benchmarks, all computational regions were set to be executed on `LocB` by default.

[3]We were unable to compile backprop, hearwall, kmeans, leukocyte, myocyte, streamcluster of Rodinia suite using OpenACC 2.6 and PGI 18.4 despite many attempts to resolve their issues. Hence we skipped them and did not port them to *Gecko*.

This gives us the flexibility to change the final locations without any changes made to the source code. By changing the configuration file, we specify where our code has to be executed (on the host or GPUs or both). We chose the *static* execution policy for all the regions of all benchmarks by default. In our experiments, we did not customize the execution location and policy for any specific region.

6 Results

We assess the performance of *Gecko* and evaluate its impact on multiple GPUs by measuring the speedup achieved and the rate of page faults. In the process, we also change the workload distribution between a host processor and a set of GPUs.

6.1 Speedup

Figure 5 shows the speedup results for the benchmarks. We measure the execution time of computation for each benchmark. We did not measure the memory allocation and its release. The *Gecko* results use one to four GPUs. *Gecko* also uses UVM to allocate memory for different level of hierarchy. The OpenACC results use one GPU by default, with Unified Virtual Memory (UVM) (`acc-managed` in the figure) and without UVM (`acc` in the figure). The speedup results are with respect to the `acc-managed` version of the code.

Currently the compilers supporting OpenACC do not provide an implementation that will support the automatic distribution of the workload to multiple devices nor is there an explicit clause within OpenACC that can be used for workload distribution. As a result, the OpenACC speedup results use only a single GPU. The results of *Gecko* for 1-GPU compared with that of the OpenACC versions (`acc-managed`, `acc`) reveal promising performance improvements. One of the key factors that contribute to such an improvement is the asynchronous execution of kernels, which is enabled by default in *Gecko*. The asynchronous execution is not enabled by default in OpenACC and programmers are required to explicitly request for such behavior.

Although *Gecko* is imposing performance overhead due to the extra technicalities to support multiple devices, the main source of performance loss is due to the UVM technology. Figure 6 shows the total page faults occurred on all GPU devices. In all cases, except for *vector add* and *cfd*, utilization of more GPU devices by *Gecko* has led to an exponential increase in total page faults on the devices.

Excessive page faults are due to two reasons: (1) CUDA performs the memory allocation on the current active device (usually device 0). Consequently, any accesses to the allocated memories from other devices are serviced via the communication medium, in our case, the PCI-E bus. However, utilizing PCI-E to fulfill remote memory accesses slows down the execution time of the kernel that leads to severe performance loss. To address this issue, NVIDIA introduced

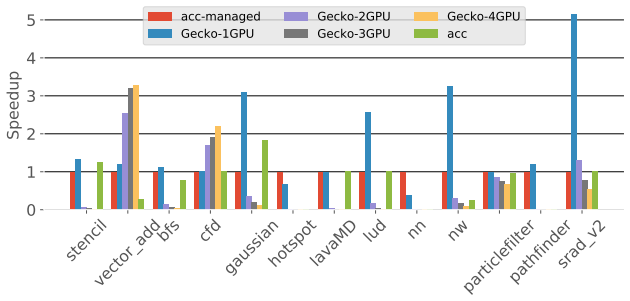

Figure 5. Speedup of benchmark applications with *Gecko*. `acc-managed` is OpenACC with UVM and `acc` is OpenACC without UVM.

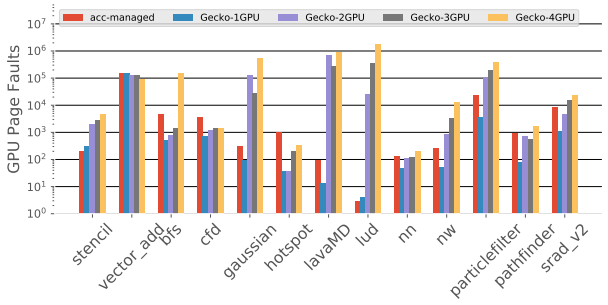

Figure 6. Total GPU page faults of benchmark applications with *Gecko*. `acc-managed` is OpenACC with UVM and `acc` is OpenACC without UVM.

NVLink [11] and NVSwitch [20] technologies that reduce access times to the remote data among GPU devices thus addressing the disadvantages of PCI-E. We expect the performance of *Gecko* to significantly improve if the aforementioned technologies are employed instead of PCI-E as the main communication medium; (2) Memory access patterns also play a significant role in obtaining the best performance. *Gecko* distributes the loop iterations among multiple devices with the idea of distributing workload. However, the memory accesses for some applications, like *hotspot*, is very similar to a stencil code – *false sharing effects*. Thus, they suffer from multi-device utilization. Therefore, it is most likely that a GPU is accessing memory belonging to another GPU. With the current allocation mechanism for UVM by CUDA, the chance of inter-GPU accesses are high.

The *cfd* and *vector_add* benchmarks benefit from *Gecko* when compared to the other ones. Further investigation of their source codes reveals that they have very little remote accesses and they access the global memory in a coalesced pattern: global memory loads and stores are packed into a few transactions to minimize the device memory accesses. The coalescing accesses minimize the turnaround time in retrieving data from the global memory.

6.2 Heterogeneous Execution

Gecko targets multiple different platforms with zero code modification. In this subsection, we will discuss targeting multiple GPU devices while the host is also participating in the workload distribution. By adding a line of code to the configuration file and declaring `LocN` as a child of `LocB`, *Gecko* recognizes the host as an execution resource.

Figure 7 shows the speedup results (wall-clock time) for the heterogeneous execution. With the help of the `percentage` execution policy, *Gecko* splits the iteration space between the host and the GPU devices. We have changed the host's iteration share from 0% to 100% and then split the rest of iterations among the GPU devices with the *static* execution policy. The X-axis in Figure 7 shows the host's share in execution. For instance, 20 on X-axis determines that host executes 20% iterations of a loop while the rest (80%) is partitioned among GPUs equally. The Y-axis shows the speedup achieved with respect to the 100% case (only on the host). Results reveal how benchmarks like *bfs*, *hostspot*, *lavaMD*, *pathfinder*, and *srad_v2* benefit from utilizing only a single GPU. In such cases, applications can use the `any` execution policy of *Gecko* to target only one single GPU. As we engage the host CPU in workload distribution, speedup drops, except for *bfs*.

Some benchmarks like *nn* and *particlefilter* are platform-neutral: utilizing either host or any number of GPUs to perform their computations will not affect their performance in any ways. The *lud* and *nw* benchmarks, on the other hand, are sensitive to the platform chosen. Heterogeneity kills the performance for such workloads. As long as all iterations are performed with a specific platform (either host or GPU devices), they do not lose performance. In such cases, the host participation in the execution adversely affects the speedup. Moreover, those benchmarks are not multi-GPU friendly too. As we increase the number of GPUs, the performance of *lud* is lost and *nw* shows no performance improvement.

The *gaussian* benchmark shows how an application benefits from assigning a big chunk of its iteration space to the GPUs. As we increase the share of the host CPU, the speedup decreases to its minimum value. Moreover, similar to *lud* and *nw*, *gaussian* is suffering from multi-GPU utilization too. When we target only GPUs (CPU share is 0), the speedup drops as the total number of GPUs utilized increases.

The *stencil* application suffers from both heterogeneity and multi-GPU execution. As we increase CPU's share in workload distribution, the performance drops gradually. Utilizing more GPUs does not improve performance. The main factor that contributes to the performance degradation in multi-device utilization is the excessive remote accesses. Stencil codes are not locality-friendly codes: computing an element of a matrix depends on the computation of neighbor points, which are not necessary close in their memory layout. As a result, a subset of accesses will cross distribution

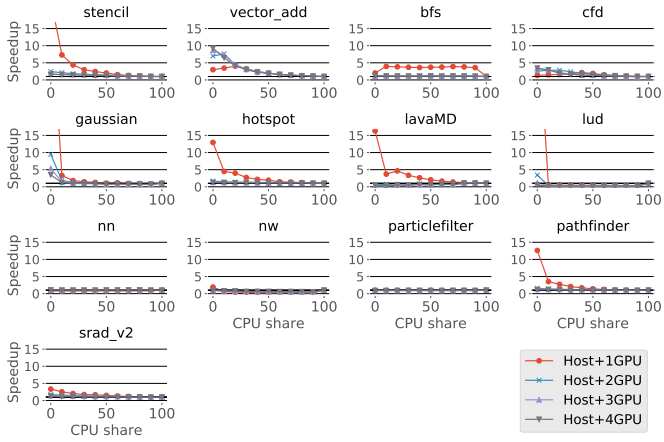

Figure 7. Simultaneous execution of benchmark applications on a single core of Intel Xeon (host) and four NVidia GPUs. The X-axis determines the percentage of iterations executed on the host (for all kernels) and the rest is distributed among GPUs. The Y-axis shows the achieved speedup with respect to 100% execution on the host.

boundaries, which, in turn, leads to remote access to other devices. Since memory consistency is guaranteed at page-size units, every remote access result in the transfer of a page between devices. Thus, the peak performance of *stencil* codes is realized when only one GPU is utilized for the whole dataset. Applications like *hotspot*, *lavaMD*, and *pathfinder* follow the same pattern as *stencil* does. Investigating their source code reveals that their memory accesses are similar to the accesses by *stencil*.

The *cfd* benchmark shows interesting results. As we decrease host's share and offload more computations to GPUs, the performance increases and single-GPU configuration performs better with comparison to the others. However, as we decrease the host's share to 40%, 1-GPU configuration stops improving. Decreasing host's share and utilizing more GPU devices lead to performance improvements. Offloading all iterations to GPUs, in the *cfd*'s case, results in better performance in comparison to other configurations. Similarly, the performance of *vector add* also improves as the host's share decreases. As the host's share decreases from 100% to 20%, speedup gradually increases up to 4.1x. *vector add* reaches its peak performance (9.26x) when we utilize all GPU devices.

7 Related Work

Shared memory systems have been the target of HPC applications for the past decade. Chores [7] proposed a programming model to enable applications to run on uniform memory access (UMA) shared-memory multiprocessors. OpenMP has also been a great advocate for parallel programming models for shared-memory architecture in a homogeneous platform, and since 4.0, they support heterogeneous systems as well. Similarly, OpenACC has provided support for heterogeneous systems and rapidly gained wide momentum. However, the above-mentioned approaches support just a single accelerator, while the prevalent trend is adding multiple accelerators to a single node. *VirtCL* [30] discusses an OpenCL programming model utilizing multiple homogeneous GPUs. It replicates an array on host and other devices, and ensures the consistency by locking the whole object (arrays) on any devices that are using it. However, locking objects prevents applications from declaring finer granularity of parallelism. *Gecko* allows finer granularity while relying on the consistency control provided by hardware.

Targeting multi-platforms have been addressed in the past. Seqouia [9] is a cluster-wide approach that is based on the Parallel Memory Hierarchy (PMH) model [2]. It represents a hierarchy among available memory spaces in the system, and workload is distributed through task definition. Similar to *Gecko*, computations in Seqouia occur within the leaf nodes, and the hierarchy is be defined through a configuration file. However, *Gecko* does not finalize the hierarchy, and applications are able to modify their structure according to their requirements. Hierarchical Place Trees (HPT) [29] model bears similarity to Seqouia and *Gecko* in the exploitation of the "location" concept. However, HPT lacks the dynamic features of *Gecko*, such as dynamic memory allocation and dynamic hierarchy. *Gecko* provides static and dynamic declaration of locations and their relationship, and, with help of the memory allocation algorithm introduced in Section 2, applications allocate memories dynamically based on the chosen location at run time. Many modern languages have also been introduced (e.g., Chapel [4] and X10 [5]) that provide facilities to describe data distribution and parallelism through an abstract model for both data and computation.

8 Conclusion

In this paper, we designed and developed a novel hierarchical portable abstraction raised at the language level in order to target heterogeneous shared memory architectures commonly found in modern platforms. Following are some of the unique features of our novel model *Gecko*: (1) The model allows the scientific developers to choose locations arbitrarily and switch between locations depending on application requirements with minimum code alteration. (2) Once the location is chosen, the decision gets relegated to the runtime that will assist with using appropriate computational resources. The runtime also acts as a smart tool dynamically choosing between the resources available for the execution of the code. (3) The model is highly user-friendly thus minimizing the amount of code to be changed whenever the architecture and the program requirements vary.

Our experiments with *Gecko* demonstrate that the model is well suited for a multi-GPU environment as it delivers

a portable and scalable solution primarily for benchmark where remote memory accesses between devices are minimum. In the near future, we will explore support for *Gecko* to enable the rich functionalities of Processing-In-Memory (PIM) architectures. Moreover, *Gecko* will support NVM and persistent memories that will be available in modern architectures.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1531814 and Department of Energy under Grant No. DE-SC0016501.

References

[1] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. 2015. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 336–348.

[2] B. Alpern, L. Carter, and J. Ferrante. 1993. Modeling parallel computers as memory hierarchies. In *Proceedings of Workshop on Programming Models for Massively Parallel Computers*. 116–123.

[3] J. A. Ang, R. F. Barrett, R. E. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S. D. Hammond, K. S. Hemmert, S. M. Kelly, H. Le, V. J. Leung, D. R. Resnick, A. F. Rodrigues, J. Shalf, D. Stark, D. Unat, and N. J. Wright. 2014. Abstract Machine Models and Proxy Architectures for Exascale Computing. In *2014 Hardware-Software Co-Design for High Performance Computing*. 25–32.

[4] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.* 21, 3 (Aug. 2007), 291–312.

[5] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 519–538.

[6] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (Oct 1974), 256–268.

[7] Derek L. Eager and John Jahorjan. 1993. Chores: Enhanced Run-time Support for Shared-memory Parallel Computing. *ACM Trans. Comput. Syst.* 11, 1 (Feb. 1993), 1–32.

[8] Diego Elias, Rivalino Matias, Marcia Fernandes, and Lucio Borges. 2014. Experimental and Theoretical Analyses of Memory Allocation Algorithms. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC '14)*. ACM, New York, NY, USA, 1545–1546.

[9] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*. ACM, New York, NY, USA, Article 83.

[10] T. B. Ferreira, R. Matias, A. Macedo, and L. B. Araujo. 2011. An Experimental Study on Memory Allocators in Multicore and Multithreaded Applications. In *2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*. 92–98.

[11] D. Foley and J. Danskin. 2017. Ultra-Performance Pascal GPU and NVLink Interconnect. *IEEE Micro* 37, 2 (Mar 2017), 7–17.

[12] M. Ghane, S. Chandrasekaran, R. Searles, M.S. Cheung, and O. Hernandez. 2018. Path forward for softwarization to tackle evolving hardware. In *Proceedings of SPIE - The International Society for Optical Engineering*, Vol. 10652.

[13] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. 2011. GPUs and the Future of Parallel Computing. *IEEE Micro* 31, 5 (Sept 2011), 7–17.

[14] Christoph Lameter. 2013. NUMA (Non-Uniform Memory Access): An Overview. *Queue* 11, 7, Article 40 (July 2013), 12 pages.

[15] Alexey Lastovetsky. 2013. Heterogeneity in parallel and distributed computing. *J. Parallel and Distrib. Comput.* 73, 12 (2013), 1523 – 1524.

[16] Alexey L Lastovetsky and Jack Dongarra. 2009. *High performance heterogeneous computing*. Vol. 78. John Wiley & Sons.

[17] Xiangke Liao, Liquan Xiao, Canqun Yang, and Yutong Lu. 2014. MilkyWay-2 supercomputer: system and application. *Frontiers of Computer Science* 8, 3 (01 Jun 2014), 345–356.

[18] G. H. Loh. 2008. 3D-Stacked Memory Architectures for Multi-core Processors. In *2008 International Symposium on Computer Architecture*. 453–464.

[19] A. K. Mishra, X. Dong, G. Sun, Y. Xie, N. Vijaykrishnan, and C. R. Das. 2011. Architecting on-chip interconnects for stacked 3D STT-RAM caches in CMPs. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 69–80.

[20] NVidia NVSwitch Whitepaper. 2018. http://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf. (2018). Accessed: 2018-08-08.

[21] NVIDIA PSG. 2017. http://psgcluster.nvidia.com/trac. (2017). Accessed: 2017-12-03.

[22] OpenACC Language Committee. 2017. OpenACC Application Programming Interface, Version 2.6. https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf. (November 2017).

[23] ORNL's Summit. 2018. https://www.olcf.ornl.gov/for-users/system-user-guides/summit/. (2018). Accessed: 2018-08-08.

[24] ORNL's Titan. 2018. https://www.olcf.ornl.gov/for-users/system-user-guides/titan/. (2018). Accessed: 2018-08-08.

[25] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das. 2016. Scheduling techniques for GPU architectures with processing-in-memory capabilities. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 31–44.

[26] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. PericÃąs. 2017. Trends in Data Locality Abstractions for HPC Systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (Oct 2017), 3007–3020.

[27] D. H. Woo, N. H. Seong, D. L. Lewis, and H. S. Lee. 2010. An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 1–12.

[28] Yonghong Yan, Ron Brightwell, and Xian-He Sun. 2017. Principles of Memory-Centric Programming for High Performance Computing. In *Proceedings of the Workshop on Memory Centric Programming for HPC (MCHPC'17)*. ACM, New York, NY, USA, 2–6.

[29] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. 2010. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *Languages and Compilers for Parallel Computing*, Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 172–187.

[30] Yi-Ping You, Hen-Jung Wu, Yeh-Ning Tsai, and Yen-Ting Chao. 2015. VirtCL: A Framework for OpenCL Device Abstraction and Management. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, New York, NY, USA, 161–172.