Abstraction of Computation and Data Motion in High-Performance Computing Systems

by

Millad Ghane

A dissertation submitted to the Department of Computer Science,

College of Natural Sciences and Mathematics

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in Computer Science

Chair of Committee: Margaret S. Cheung, University of Houston

Committee Member: Larry (Weidong) Shi, University of Houston

Committee Member: Edgar Gabriel, University of Houston

Committee Member: Sunita Chandrasekaran, University of Delaware

University of Houston

December 2019

"What you seek is seeking you."

– Rumi

"The mountain begins with the first rocks,

And the man with the first pain."

– Shamlou

To my mom, dad, and sister,
Farideh, Asadollah, and Midya.

I genuinely miss you all,
You are my whole world.

This whole dissertation is devoted to you.

Acknowledgment

I am indebted to many great people during the past 5 years at UH. I want to take this opportunity and acknowledge their valuable contributions to my life and this dissertation. First and foremost, I want to express my gratitude to Professor Margaret S. Cheung as my advisor, and Professor Sunita Chandrasekaran as my co-advisor for all their help, consideration, inspiration, and encouragement during my journey. I was fortunate to have them as my academic advisors during my studies.

I also want to thank the members of my PhD committee, Dr. Larry (Weidong) Shi and Dr. Edgar Gabriel at UH for their support and insights. And, furthermore, I want to thank Dr. Chris (CJ) Newburn and Chris Lamb for the summer internship opportunity at NVIDIA. It was a very unique opportunity for me to spend a summer with them and expand my knowledge in my research area.

During my PhD, I got to know many amazing people who they became good friends of mine. Here, I try to remember as many of their names as possible.

At UH and in Houston, Hessam Mohammadmoradi (Moradi), Milad Heydariaan, Yaser Karbaschi, Farnaz Ziaee, Foroozan Akhavan, Hosein Neeli, Amir Hossein Souri, Reza Fathi, Mohammadmehdi Ezzatabadipour, Hasan Salehi Najafabadi, Masoud Poshtiban, Siddhartha Jana, Shengrong Yin, Nour Smaoui, Nacer Khalil, Fabio Zegarra, Caleb Daugherty, Andrei Gasic, Yossi Eliaz, Pengzhi Zhang, Jacob Ezerski, James Liman, Jacob Tinnin, Carlos Bueno, and Jules Nde.

At NVIDIA and PGI, Chris (CJ) Newburn, Chris Lamb, Anshuman Goswami, Sreeram Potluri, Michael Wolfe, Mathew (Mat) Colgrove, Duncan Poole, Pat Brooks, James Beyer, Guray Ozen, Julia Levites, Jeff Larkin, Robert Searles, and Jinxin (Brian) Yang.

Finally, and most importantly, my deepest appreciation goes to my beloved family back in Iran, especially my mom Farideh, my dad Asadollah, my sister Midya, and all my aunties and uncles (especially my close aunties, Zhila and Valieh). It has been a very long journey, both in time and space for me and you. I would not have been able to do this without your unconditional love and support from such a distance. This dissertation is devoted to all of you because of all the sacrifices that you have made for me during the 33 years of my life (and especially the last 5 years) that I had on this planet. Thank you!

Abstract

Supercomputers are at the forefront of science and technology and play a crucial role in the advancement of contemporary scientific and industrial domains. Such advancement is due to the rapid developments of the underlying hardware of supercomputers, which in turn, have led to complicated hardware designs. Unlike decades ago when supercomputers were homogeneous in their design, their current developments have been widely heterogeneous to lower their energy and power consumption.

As hardware architectures of supercomputers become complex, so do the software applications that target them. In recent years, scientists have been utilizing directive-based programming models, such as OpenMP and OpenACC, to mitigate the complexity of developing software. These programming models enable scientists to parallelize their code with minimum code interventions from their developers. However, targeting heterogeneous systems effectively is still a challenge despite having productive programming environments.

In this dissertation, we will introduce a directive-based programming model and a hierarchical model to improve the usability and portability of several scientific applications and prepare them for the exascale era. For the first model, our `pointerchain` directive replaces a chain of pointers with its corresponding effective address inside a parallel region of code. `Pointerchain` enables developers to efficiently perform deep copying of the data structures in heterogeneous platforms. Based on our analysis, `pointerchain` has led to 39% and 38% reductions in the amount of generated code and the total executed instructions, respectively.

Secondly, our hierarchical model, *Gecko*, abstracts the underlying memory hierarchy of the exascale platforms. This abstraction paves the way for developing scientific applications on supercomputers. To prove its feasibility, we developed an implementation of Gecko as a directive-based programming model. Moreover, to evaluate its effectiveness, we ported real scientific benchmark applications — ranging from linear algebra to fluid dynamics — to Gecko. Furthermore, we also demonstrated how Gecko helps developers with code portability and ease-of-use in real scenarios. Gecko achieved a 3.3 speedup on a four-GPU system with respect to one single GPU while having only a single source-code base.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 What is High-performance Computing (HPC)?

High-performance computing (HPC) plays an imperative role in the advancement of engineering, science, research, security, and industry. It is at the forefront of discovery in science and commercial innovations over the past years. HPC, in general, refers to the utilization of high-end supercomputers and computing machines to solve complex problems through data analysis and simulations. A supercomputer is basically a collection of high-end computers (referred to as *compute nodes*), which are connected through an efficient high-speed network infrastructure. Each compute node comprises multiple processing cores with their own local memory space. Modern generations of supercomputers have hundreds of thousands of such compute nodes to satisfy the ever-growing performance demand of the HPC applications. Currently, we are reaching the exascale era of HPC as the modern generation HPC systems are performing one quintillion (10^{18}) floating-point operations per second. With such capability, HPC systems are experiencing major changes to overcome the constraints imposed by the energy, power [39], and locality of the exascale systems [6, 106].

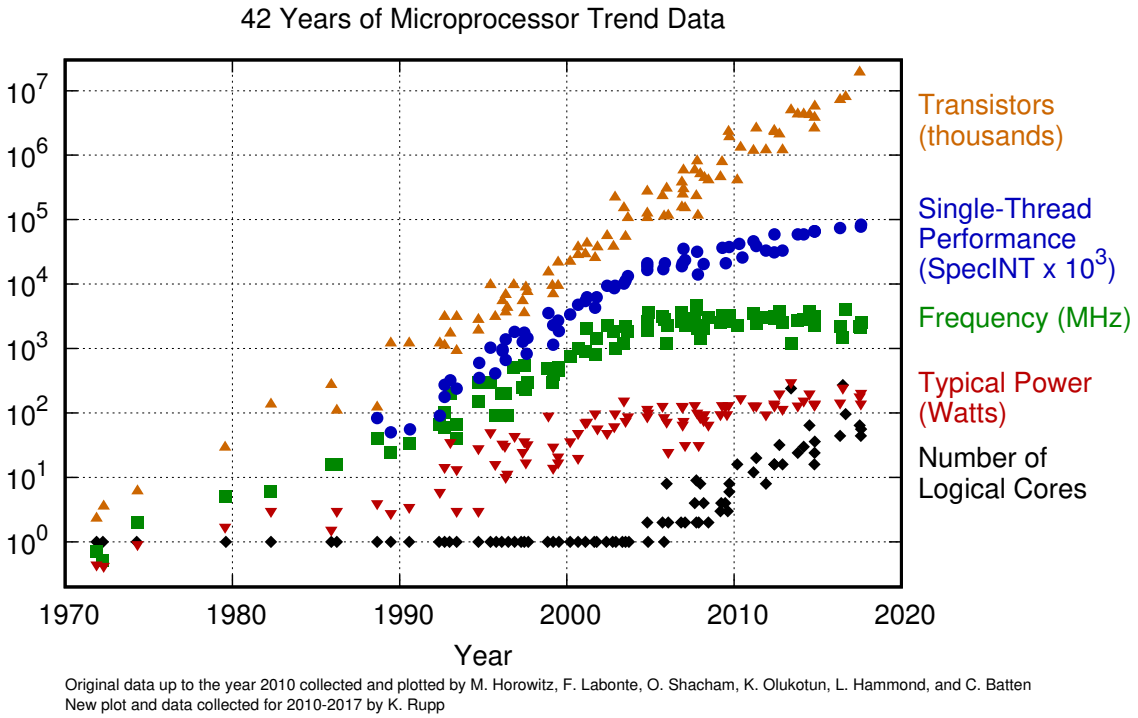


Figure 1.1: The trend of 42 years of microprocessor design. Courtesy of Karl Rupp [96].

1.2 Heterogeneity Is the Answer

Since 1971, Moore's Law [98] has been the main source of performance improvement in microprocessor design. Based on the observation of Gordon Moore, the number of transistors on a chip is essentially doubling every two years, which has implicitly led to doubling performance. However, with the emergence of the *power wall* [6, 39], the landscape of multiprocessor design has changed dramatically. Since the early 2000s, the working frequency of transistors has been essentially flat, and this has also led to a constant performance rate (as is shown by the blue circles in Figure 1.1). As elegantly put by Sutter: "The free lunch is over [101].[1]" This phenomenon is also referred to as the law of Dennard scaling [24][2]. Consequently, computer architects are increasing the number of logical cores on the device

[1]This statement refers to the fact that software no longer runs faster with each successive generation of microprocessors. Developers have to modify their software to utilize their hardware to the fullest extent.

[2]In short, Dennard scaling refers to the fact that increasing the working frequency of transistors is not a feasible approach anymore, and we have to turn to new methods (e.g., multicore design and heterogeneous computing).

in order to utilize the potential performance of all available transistors on the chip. That being the case, we begin to observe the advent of the multicore era, as evidenced by the rising rate of the number of logical cores in Figure 1.1.

Dennard scaling [24] has instigated the HPC community's adaptation of heterogeneous architectures in the design of supercomputers and high-performance clusters. As a result, heterogeneity has been the primary source of computational performance in modern high performance systems [7, 25, 59] since the decline of conventional improvements [24, 104].

1.2.1 Node-level Heterogeneity

Heterogeneity is not realized through a single approach. A broad range of decisions on the architectural level leads to heterogeneity. On the node level, integration of accelerators with conventional processors has been the primary technique in improving the performance of current supercomputers. Accelerators like Graphics Processing Units (GPUs) [53] and Many Integrated Cores (MICs) have been widely adopted in the design of modern HPC systems. Such systems possess one or two conventional processors with a couple of GPUs and MICs. As an example, the ORNL's latest supercomputer, Summit [84], possesses six NVIDIA Volta V100 GPUs and only two IBM POWER9 processors per node. Similarly, the Tianhe-2 supercomputer [61], located in the National Supercomputer Center in Guangzhou, China, is equipped with three Intel Xeon Phi MIC accelerators while having only two Intel Ivybridge Xeon processors per node.

1.2.2 Chip-level Heterogeneity

On the chip level, we achieve heterogeneity with diversity in the chip design. Any diversity in frequency, voltage, core area, application[3], and instruction set (ISA) leads to a heterogeneous

[3]Application-specific cores are those cores that are dedicated to accelerate specific class of problems. Examples of such cores are Digital Signal Processor (DSP), encryption accelerators, and speech accelerators.

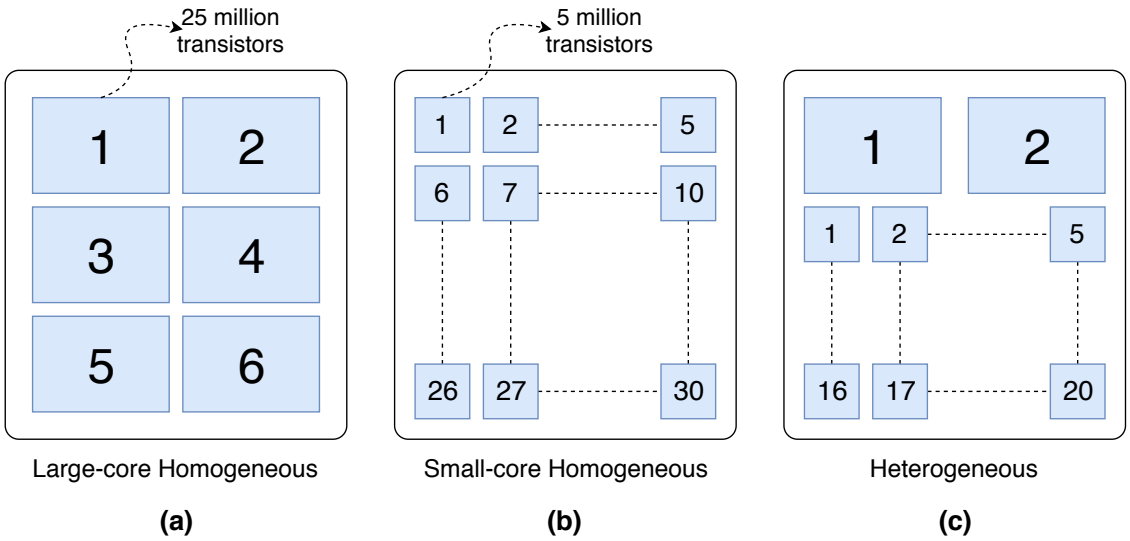


Figure 1.2: Three scenarios to integrate 150-million transistors into different cores in a heterogeneous architecture. Adopted from Borkar and Chien [13].

chip design. Figure 1.2 shows three integration scenarios when we budget our transistors differently. As illustrated in Figure 1.2a, we can have six large (fat) cores on a single chip, where each core has 25 million transistors for a total of 150 million transistors. This is the most common approach in the design of processors in the multicore era. If we change our budget strategy and allocate only five million transistors to each small (thin) core, as illustrated in Figure 1.2b, we will end up having a homogeneous architecture with 30 cores. However, by mixing these two approaches, we will end up having a heterogeneous architecture. The imposed heterogeneity is due to the diversity in the core areas on the chip, as illustrated in Figure 1.2c. Furthermore, any variations to the working frequency of the on-chip cores results in a heterogeneous design too. For instance, Figure 1.3 depicts a scenario where a few large cores are utilized for single-thread performance while many small cores, under different working frequencies, are used to perform energy-efficiently.

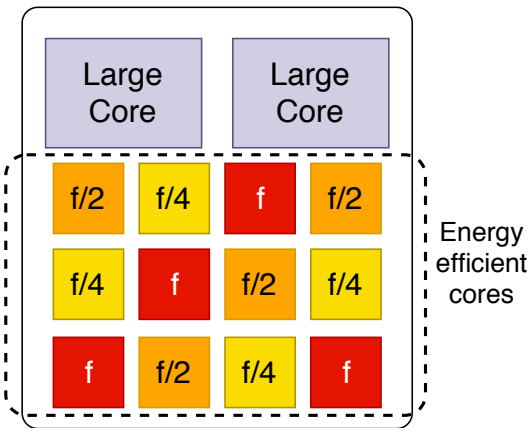


Figure 1.3: A heterogeneous processor with three different frequency domains. Each frequency domain has four cores. This design will lead to an energy-efficient processor. Adopted from Borkar and Chien [13].

1.2.3 Heterogeneity of On-chip Memories

The complexity arises when we use the transistor budget to expand the on-chip storage capability of the system. The computational and memory components in platforms like Summit [84] are installed on separate chips and connected to each other through fast interconnect networks. Accessing data from another component is feasible only through such interconnects. However, data transfer via the interconnects is very slow, and it should be avoided. As a result, computer architects are integrating processors and memory modules on a single chip. Such integration has been feasible with advances in semiconductor manufacturing.

This radical change in architecture design leads to better data locality for on-chip processors, which minimizes data transfer between on- and off-chip components. Modern on-chip memories have manifested themselves in the form of scratchpad memories [10], 3D Stacked memories [64, 110], and Flash-based memories [58]. The integration of these components on the chip generates another level of complexity, which inevitably will happen in upcoming architectures and chips in the near future.

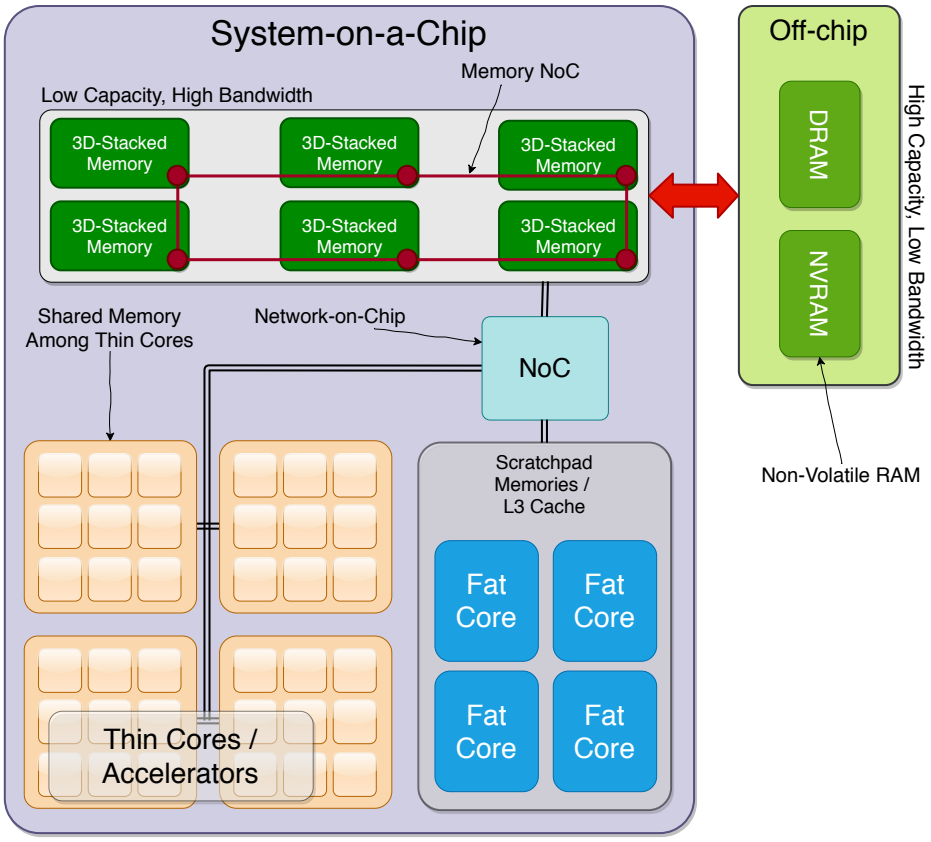


Figure 1.4: An Abstract Machine Model (AMM) for future architectures. This figure represents the increasing level of complexity that high performance computing systems will face in near future (as one can also observe from Figure 2.1). Adopted from Unat et al. [104].

1.3 Abstraction Models

Figure 1.4 provides a detailed abstract model of a future architecture [104], which we will refer to it as the Abstract Machine Model (AMM) throughout this work. An exascale node in the future will most likely follow this configuration. In AMM, a node is composed of a set of *thin cores* (e.g., GPUs and MICs) and *fat cores* (e.g., general purpose processors like x86 [48] and Arm [49, 43] processors). Both of them (thin and fat cores) are embedded inside a single chip to minimize the data transfer time between all the components. Fat cores in the figure are characterized by their sophisticated branch prediction units, deep pipelines, instruction-level parallelism, and other architectural features that optimize the serial execution to its full extent. Fat cores are similar to the conventional processors in current systems.

Thin cores, on the other hand, have less complex design, consume less energy, need less physical die space, and have higher throughput. Such cores are designed to boost the performance of data parallelism algorithms [7]. Both fat and thin cores communicate with each other and transfer data via a customized network on the chip, known as Network-on-Chip (NoC) as shown in Figure 1.4 [11, 34]. This model is providing better spatial locality to the cores in comparison to the contemporary approaches where cores and memories reside on different chips.

On-chip memories are utilized to provide data in high bandwidth with low latency to the on-chip cores. This design provides high spatial locality to cores. Consequently, we are in dire need of a simple and robust model to efficiently express such implicit memory hierarchy along with the parallelization opportunities available in current advances in hardware design for scientific applications [104].

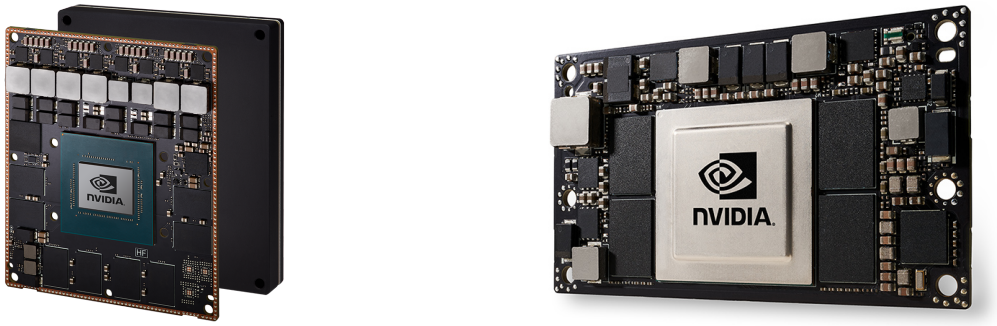
As mentioned above, compute nodes are evolving and becoming increasingly complex while offering several folds of parallelism; hence, compute nodes demand sophisticated programming approaches to exploit every parallelism opportunity exhibited by their underlying hardware [73, 104]. As compute nodes become complex on the hardware level, so does developing software applications for them. Hence, to leverage available hardware resources on all connected devices, we need low-level programming frameworks, such as CUDA and OpenCL, for developers. However, merely exploiting low-level or proprietary languages such as CUDA or OpenCL in the long run will be impractical. A low-level framework comes with *maintenance and debugging challenges, the inability to move the code to any other hardware, and the steep learning curve that demands an in-depth knowledge of memory, hardware, and software stack*. We need to examine trade-offs while applying optimization techniques that maximize utilization of accelerator devices (e.g., GPUs and MICs) so that the additional programming burden would still lead to noticeable improvement. Moreover, as researchers have confirmed in their studies [73, 102, 104, 13, 69], there is an urgent need for portability

of software architecture in order to support recent developments in HPC. Easily portable software tools would allow the scientific community to spend more time on science and less time on programming challenges. This serves as a strong motivation for our work.

High-level directive-based programming models [83, 82, 60, 95] have been a promising solution to map scientific applications to hardware with rich and evolving features. Such models typically require minimal to no code changes, thus allowing scientists to focus more on science and less on the programming itself. Contrary to the legacy approaches[4], which are inherently infeasible with the currently rapid advancement in hardware development, directive-based programming models provide unprecedented opportunities and freedom to developers of scientific applications in utilizing a system to its full extent.

As clearly stated before, heterogeneous systems are the current state-of-the-art computing platform that support scientific simulations beyond a petascale computing era. Such a demand suggests supplying nodes with accelerators that feature high levels of parallelism such as on-chip or off-chip GPUs and MICs. In the Top500 list of high-computing resources in December 2017 [103], eight out of the top ten supercomputers are equipped with varying types of accelerators. This number nearly triples when it is compared to the same list in December 2010 where only three out of the top ten architectures adopted GPUs. This progress shows how heterogeneity has become the de facto standard in designing a modern HPC system. A next-generation exascale system by the year 2023 [73] will require the effective integration of different accelerators [66, 4] into computational nodes that expect high levels of parallelism.

[4]The legacy approaches utilize a runtime library for every architecture and platform that we target in our application.



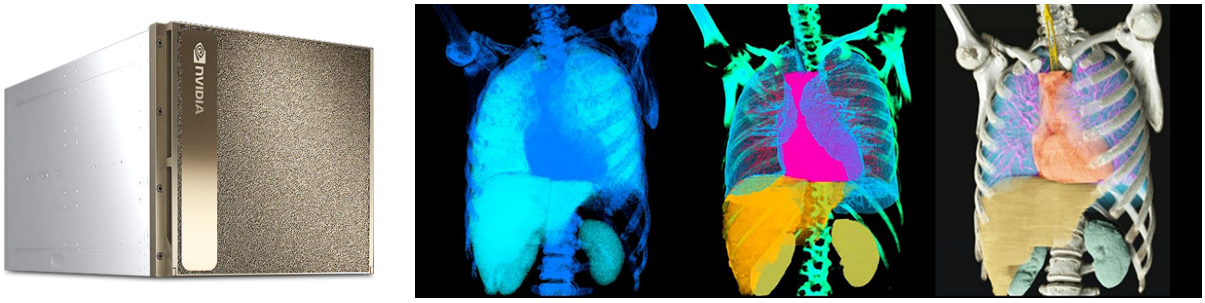
(a) Nvidia Jetson AGX Xavier **(b)** Nvidia Jetson TX2

Figure 1.5: The latest embedded platforms by NVidia. © Courtesy of NVidia.

1.4 Why Does HPC Matter?

The advances in HPC has also led to the development of various commercial technologies, which potentially impact every day of our lives. Due to the success of the NVidia GPUs in their experiences in HPC, they have brought the promised computational power of GPUs to industry with recent products in the autonomous (self-driving) vehicles and healthcare domains. As the need to deploy the self-driving technology to bring a safer experience among drivers grows, they need high-performance computing processors to perform their split-second decisions in controlling cars on the road. To address this issue, NVidia has introduced its embedded platform, Jetson [79, 76]. This embedded platform has expanded its application to other areas in industry as well, such as robotics [113, 76], Artificial Intelligence of Things (AIoT) [62], manufacturing [67, 76], and construction [9]. Figure 1.5 shows two of the latest Nvidia flagship products, the Jetson AGX Xavier and Jetson TX2 platforms.

NVidia has disrupted the healthcare industry as well with its latest hardware and software stack. NVidia Clara [77] is a computational platform that empowers physicians and doctors to improve the diagnostic accuracy and the quality of their medical imagings to enhance patient outcomes and reduce the cost of their care. This has been enabled with the latest DGX-2 products by NVidia. DGX-2 is the first two-PetaFLOPs system that engages



(a) Nvidia DGX-2 [78] **(b)** A sample output of Nvidia Clara Imaging [77]

Figure 1.6: NVidia's DGX-2 and a snapshot of the output of the Clara platforms. © Courtesy of NVidia.

16 GPUs (fully interconnected). Figure 1.6 shows a DGX-2 system and an output of an image from NVidia Clara.

1.5 Organization of this Dissertation

In this work, we present two directives that make the current source codes portable for future exascale systems with minimum code modifications. The first directive, `pointerchain`, enables developers to deal with the chain of pointers in the source code. When source code possesses nested data structures, the parallelization process becomes very complicated with the current approaches when targeting modern accelerators. By maintaining the effective address of the last pointer in the chain, `pointerchain` provides better flexibility to the developers. Chapter 3 discusses the `pointerchain` directive, and we discuss a case study in molecular dynamics (MD) domain based on `pointerchain` in Chapter 5 as well.

The second directive is a realization of the hierarchical model that we introduce to address the heterogeneity in current HPC systems. Our model, Gecko, is a hierarchical distributed view of shared memory architectures. Gecko enables developers to target heterogeneous

systems with minimum code modification, similar to `pointerchain`. With Gecko, an application is able to target different architectures without any changes in the source code and, more importantly, in the executable binary code. Chapter 6 discusses different capabilities of Gecko in more details.

Portability of the current source codes for future hardware generations has always been a concern of the scientific community. Supporting new architectures and platforms with the same source code is the ultimate goal of scientists. To demonstrate how scientists can harness Gecko's potential to their benefit, we chose the Rodinia benchmark suite [16] and ported it to Gecko [36]. The suite contains a number of benchmarks that possess different workload characteristics. They have real applications in scientific domains. Each benchmark represents different scientific domains that range from solving a system of linear algebras to traversing and parsing graphs in graph-based algorithms. Chapter 6 discusses such benchmarks and our efforts in porting them to Gecko.

Finally, this dissertation includes two appendices. Appendix A describes Gecko's directives in detail. For each clause in Gecko, the syntax of the clause and an example for that clause is provided. Appendix B describes the funding sources that generously supported this work. It also includes the address of the online repository for all developed source codes during this work.

Chapter 2

Background

Previously published content:

1. M. Ghane, S. Chandrasekaran, and M. S. Cheung. pointerchain: Tracing pointers to their roots A case study in molecular dynamics simulations. Parallel Computing, 2019. [37]

2. M. Ghane, S. Chandrasekaran, and M. S. Cheung. Gecko: Hierarchical Distributed View of Heterogeneous Shared Memory Architectures. In Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM '19, New York, NY, USA, 2019. ACM. [36]

Heterogeneous computing systems comprise multiple and separate levels of memory spaces; thus, they require a developer to explicitly issue data transfers from one memory space to another with software application programming interfaces (APIs) [104]. In a system composed of a host processor and an accelerator (referred to as device in this dissertation), the host processor cannot directly access the data on the device and vice versa. For such systems, the data are copied back and forth between the host and the device with an explicit request from the host[1]. This issue has become particularly severe for supercomputers as the

[1]In this scenario, accelerators are passive elements, and the host is the active element that controls the whole system (including the accelerators).

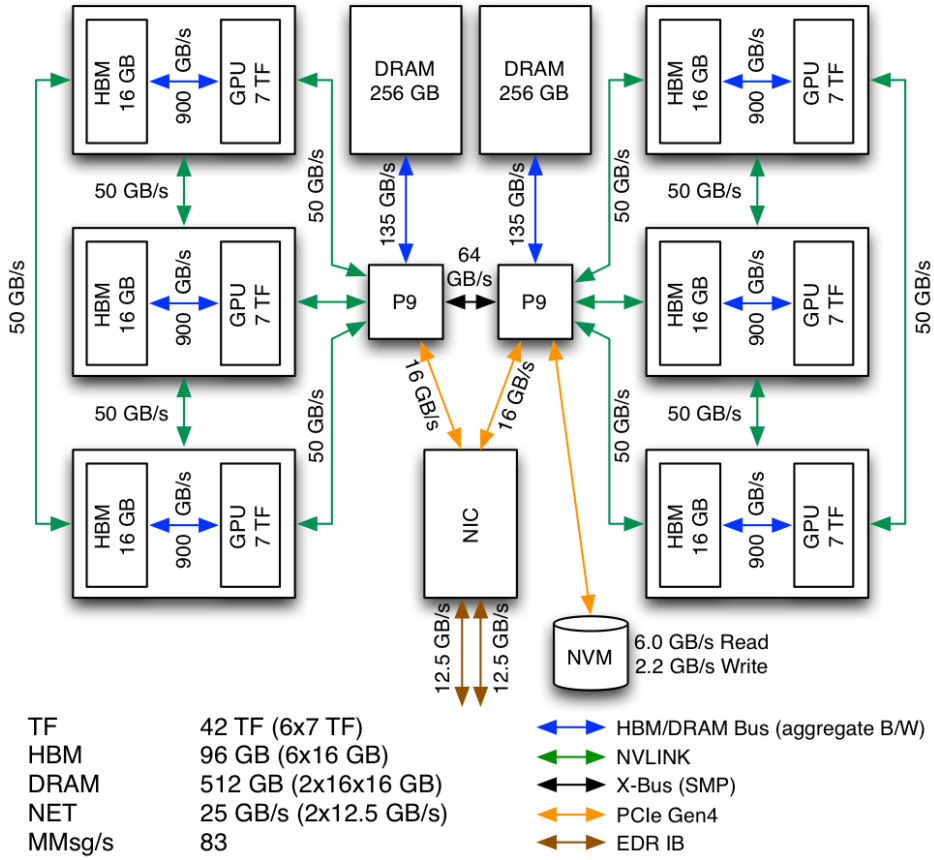


Figure 2.1: The architecture of a single node in Summit [84]. © Courtesy of Oak Ridge National Laboratory (ORNL). Each node possesses two processors (IBM Power9 shown as P9), two DRAM modules as the main memory (256 GB each), an NVM memory module (to save the temporary state of the application), a network interface card (NIC), and six NVIDIA Volta GPUs. Note how local storage (NVM) becomes a bottleneck to the system performance and how NIC provides better bandwidth in comparison to NVM.

number of devices connected to one node increases. For example, the Titan supercomputer from ORNL [85] has only one NVIDIA K20 GPU[2] per node, while this number for the latest supercomputer, Summit from ORNL [84], is six NVIDIA Volta GPUs [84]. Figure 2.1 demonstrates the complexity of the Summit architecture for only one node. Each node possesses two processors (IBM Power9 shown as P9), two DRAM[3] modules as the main memory (each one has the capacity of 256 GB), an NVM memory module (to save the temporary results of the application), a network interface card (NIC) to communicate with other nodes

[2]Graphics Processing Unit
[3]Dynamic Random-access Memory

through the high performance network, and six NVIDIA Volta GPUs. Each GPU has an HBM memory module on the card with a 16 GB global memory. Three GPUs form a clique amongst themselves and are connected through NVLink connections. They also form a locality domain with one of the P9 processors. Supercomputers with different device families will continue to exacerbate this issue [104].

Developing software for the heterogeneous systems poses a challenge to the community of scientific applications. First, we will discuss how nested data structures in scientific applications are an issue for developers to handle, especially as developers target heterogeneous architectures. Due to their separate memory spaces, handling nested data structures is cumbersome work and leads to severe performance loss in heterogeneous architectures. Second, we will discuss the challenges that the scientific applications are facing in targeting heterogeneous architectures. In addition, we will argue how portability has become a first-class concern for future generations of the scientific applications.

2.1 Challenges of Nested Data Structures

As a scientific framework becomes sophisticated, so do its data structures. A data structure typically includes pointers (or dynamic arrays) that point to *primitive data types* or to other user-defined data types [12, 18]. As a result, transfer of the data structure from the host to the other devices mandates the transfer of not only the main data structure but also its nested data structures, a process known as *deep copy* [81, 18]. The tracking of pointers that represent the main data structure on the host from its counterpart on the device further complicates the maintenance of the data structure. Although this complicated process of deep copy avoids a major change in the source codes, it imposes unnecessary data transfers. In some cases, a *selective deep copy* is sufficient when only a subset of the fields of the device's data structure is of interest [81]; however, even though the data motion decreases

proportionately, the burden to maintain data consistency among the host and other devices still exists.

This dissertation addresses the shortcomings of data transfer between the host and a device by extracting the effective address of the final pointer of a chain of pointers [18]. Utilizing the effective address leads to a reduction in the generated assembly code by replacing the pointer chain with a single pointer as well. This single pointer suffices for the correct execution of the kernel on both the host and the device with no code modification. As a result, our method improves the performance of our parallel regions by reducing the generated assembly code and omitting an unnecessary deep copy of the data structures between the host and the device. We have developed the `pointerchain` directive to provide these helpful features to developers in data transfer, which eliminates the need for a complete implementation of deep copy in a compiler and runtime library and modifying the source codes. Chapter 3 provides detailed discussion of our proposed directive.

We have demonstrated the merit of `pointerchain` by improving the portability of a molecular dynamics (MD) proxy application (known as CoMD [21]) on a heterogeneous computing system. MD is an essential tool for investigating the dynamics and properties of small molecules at the nano-scale. It simulates the physical movements of atoms and molecules with a Hamiltonian of N-body interactions. Over the past three decades, we have witnessed the evolution of MD simulations as a computational microscope that has provided a unique framework for understanding the molecular underpinning of cellular biology [91], which applies to a large number of real-world examples [42, 114, 100, 32, 115, 29, 99]. Currently, major MD packages, such as AMBER [90], LAMMPS [93], GROMACS [63], and NAMD [92], use low-level approaches, like CUDA [75] and OpenCL [54], to utilize GPUs to their benefits for both code execution and data transfer. They are not, however, equipped for the dire challenge in next-generation exascale computing in which the demand of parallelism [86] is achieved by the integration of a wide variety of accelerators, such as

GPUs [53] and MICs [66, 4], into the high-performance computational nodes.

We have chosen the OpenACC programming model [82] as the target programming model in implementing the `pointerchain` directive. Ratified in 2011, OpenACC is a standard parallel programming model designed to simplify the development process of scientific applications for heterogeneous architectures [108, 45]. The success of our approach has far-reaching impact on modernizing legacy MD codes, readying them for the exascale computing. Chapter 5 provides detailed discussion on MD simulation frameworks and steps we took to parallelize an MD proxy code with OpenACC and `pointerchain`.

2.2 Challenges of Accessing Memory in Heterogeneous Systems

The aforementioned heterogeneity in current and future generation of exascale supercomputers has become complicated as we face the challenges imposed by the "memory wall" [7]. Essentially, memory wall refers to the phenomenon of how the memory subsystem has become a huge bottleneck for the performance in the current systems. To mitigate the effects of the memory wall, the memory subsystem has undergone many changes, and the technologies in designing the memory subsystem have advanced substantially.

Recent advances in memory technologies have led to dramatic changes in their hierarchy. Deep memory hierarchy is no longer defined as multiple cache levels (L1, L2, and L3), DRAM, and external memory devices. The inclusion of novel memory technologies, such as non-volatile memory (NVM) [70] and 3D-stacked memory [64, 110] has complicated the hierarchy too. Various technologies like High Bandwidth Memory (HBM) [50], Hybrid Memory Cube (HMC) [88], and Phase-change Memory (PCM) [109, 94] address different aspects of the challenges imposed by the memory wall (like trade-offs between bandwidth and capacity

or on-chip versus off-chip memories). As the design factors grow, the complexity in both the hardware and the software components compels researchers to develop a performance-, developer-friendly approach. The above-mentioned complexity is exacerbated when multiple devices with different hardware types are utilized to parallelize the code. It is particularly problematic when they are equipped with diverse memory technologies, e.g., the Processing-In-Memory (PIM) technology on multicore and GPU architectures [2, 87]. Recent advances in die-stacking (3D) technology has put PIM in the spotlight as one of the promising methods to tackle the memory wall [65, 74]. It tries to bring the memory subsystem closer to the compute units so that the data transfer between them is performed at higher bandwidth, lower latency, and lower energy consumption in comparison to the other memory technologies in the system (which resides outside of the processor, unlike PIM units). The potentials of the PIM concept has disrupted both homogeneous and heterogeneous domains [74, 87].

To that end, a simple yet robust model helps applications to utilize the hardware to their benefit [28, 112]. Applications can put their frequently used data close to their computational cores in hope of maximizing locality, while the data that are used less frequently are placed as far as possible with respect to the compute unit (for instance, in the main memory or the secondary-storage devices like hard disks or solid-state drives). That being the case, we will present *Gecko*[4] [36], a novel programming model and runtime library system that represents the hierarchical structure of memory and computational resources in current and future HPC systems. *Gecko* addresses multi-device heterogeneous platforms and provides a portable software platform to applications to add and remove memory and computational resources at the compilation and execution time.

Gecko provides a distributed shared memory (DSM) view of available memory hierarchy in a hierarchical tree model to the applications. *Gecko*, however, has a specific rule: **a variable defined in a particular location is accessible by that location and all**

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

of its children, while the same variable remains *inaccessible* with respect to its parent. This rule enables applications to improve their data locality by bringing the data closer to where it is supposed to be processed, and eventually prevent the side-effects of data sharing (also known as *false sharing* [40]). Developed as a directive-based programming model on top of OpenACC [82] and OpenMP [83], *Gecko* consists of a set of directives that help applications to declare devices, allocate memories, and launch kernels on different devices. Chapter 6 provides a detailed discussion on *Gecko* and how to utilize it.

Chapter 3

Chain of Pointers and Deep Copy

Previously published content:

1. Ghane, Millad, and Chandrasekaran, Sunita, and Cheung, Margaret S., "pointerchain: Tracing pointers to their roots - A case study in molecular dynamics simulations." *Parallel Comput.*, 2019. [37]
2. Ghane, Millad, and Chandrasekaran, Sunita, and Cheung Margaret S., "Assessing Performance Implications of Deep Copy Operations via Microbenchmarking." *arXiv preprint*, 2019. [35]

In this chapter, we will present a directive, called `pointerchain`, as an extension to the OpenACC programming model. The proposed directive will help developers to transfer complex data structures between connected devices and the host with minimum code intervention.

3.1 The Programmatic Feature Gap

Modern HPC platforms possess two separate memory spaces: the *host* memory space and the *device* memory space. A memory allocation in one of them does not guarantee an allocation

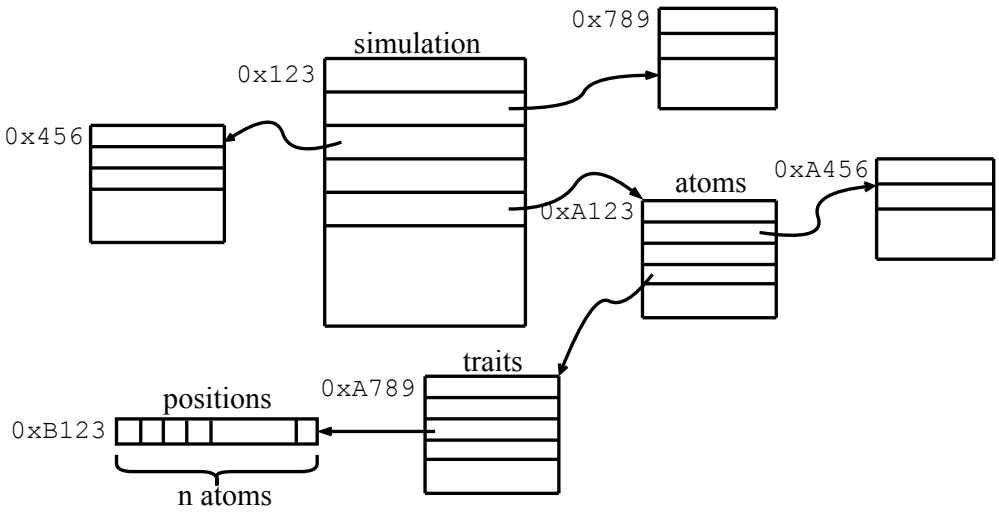


Figure 3.1: An example of a pointer chain. An illustration of a data structure and its children. In order to reach the `position` array, one must go through a chain of pointers to extract the effective address.

in the other. Such an approach demands a complete replication of any data structure in both spaces in order to guarantee data consistency. However, data structures get complicated as they retain complex states of the application.

Figure 3.1 shows a typical case in the design of a data structure for scientific applications. The arrows represent pointers. The number next to each structure shows the physical address of an object in the main memory. Here, the main data structure is the `simulation` structure. Each object of this structure has pointers embedded to other structures, like the `atoms` structure. The `atoms` structure also has a pointer to another `traits` structure, and so on. As a result, in order to access the elements of the `positions` array from the `simulation` object we have to dereference the following chain of pointers: `simulation->atoms->traits->positions`. Every arrow from this chain goes through a dereference process to extract the effective address of the final pointer. Throughout this thesis, this chain of of accesses to reach the final pointer (in this case, `positions`) is called a *pointer chain*. Since every pointer chain eventually resolves to a memory address, we propose the extraction of the effective address and replace it with the chain in the parallel sections of the code.

Currently, there are two primary approaches to address pointer chains. The first approach is the deep copy that requires excessive data transfer between the host and the device as previously mentioned in the Introduction chapter. The second approach is the utilization of Unified Virtual Memory (UVM) on Nvidia devices. UVM provides a single coherent memory image to all processors (CPUs[1] and GPUs) in the system, which is accessible through a common address space [57]. It eliminates the necessity of explicit data movement by applications. Although it is an effortless approach for developers, it has several drawbacks: 1) It is only supported by Nvidia devices and not by others (Xeon Phis, AMD GPUs, FPGAs, etc.); 2) It is not a *performance-friendly* approach due to its arbitrary memory transfers that potentially happen at any time. The consistency protocol in UVM depends on the underlying device driver that traces memory page-faults on both host and device memories. Whenever a page fault occurs on the device, the CUDA driver fetches the most up-to-date version of the page from the main memory and provides it to the GPU. Similar steps are taken when a page-fault happens on the host.

Although deep copy and UVM address the data consistency, they impose different performance overheads on the application. In many cases, we are looking for a somewhat intermediate approach; while we are not interested in making a whole object and all of its nested children objects accessible on the device (like UVM), we want to transfer only a subset of the data structures to the device without imposing the deep copy's overhead on the performance of our application. Our proposed approach, `pointerchain`, is meant to be a minimal approach that borrows the beneficial traits of the above-mentioned approaches. `pointerchain` is a directive-based approach that provides selective accesses to data fields of a structure while having a less error-prone implementation.

[1] Central Processing Units

3.2 Proposed Directive: `pointerchain`

As a compiler reaches a pointer chain in the source code, it generates a set of machine instructions to dereference the pointer and correctly extract the effective address of the chain for both the host and the device. However, dereferencing each intermediate pointer in the chain is the equivalent of a memory load operation, which is a high cost operation. As the pointer chain lengthens with a growing number of intermediate pointers, the program performs excessive memory load operations to extract the effective address. This extraction process impedes performance, especially when the process happens within a loop (for instance a `for` loop). In order to alleviate the implications of the extraction process, we propose to perform the extraction process before the computation region begins, and then reuse the extracted address within the region afterwards.

The example in Figure 3.1 demonstrates the idea of extracting process from a pointer chain. In this configuration, the pointer chain of `simulation->atoms->traits->positions` is replaced with its corresponding effective address (in this case `0xB123`). This pointer, then, is used for data transfer operations to and from the accelerator and also the computational regions. It bypasses the transmission of redundant structures (in this case, `simulation`, `atoms`, and `traits`) to the accelerator that, in any case, will remain intact on the accelerator. The code executed on the device will modify none of these objects. Moreover, it keeps the accelerator busy performing "useful" work rather than spending time on extracting effective addresses.

Utilizing the effective addresses as a replacement to a pointer chain, however, demands code modifications in both the data transfer clauses and the kernel codes. To address these concerns, we propose `pointerchain`, a directive that minimally changes the source code to announce the pointer chains and to specify the regions that include the pointer chains. The justification for having an `end` keyword in `pointerchain` is that our implementation does

not rely on a sophisticated compiler (as will be discussed in Section 3.3) to recognize the beginning and the end of complex statements (e.g., for-loops and compound block statements). Our motivation behind utilizing a script rather than a compiler was to minimize the prototyping process and implement our proof-of-concept approach by avoiding the steep learning curve of the compiler design. The steps mentioned in Section 3.3 can also be supported with a modern compiler.

3.2.1 Expanded Version

In its simple form, the `pointerchain` directive accepts two constructs: `declare` and `region`. Developers use `declare` construct to announce the pointer chains in their code. The syntax in C/C++ is as follows:

`#pragma pointerchain declare(`*variable [,variable]...*`)`

where *variable* is defined below:

`variable := name{type[:`*qualifier*`]}`

where

- `name`: the pointer chain
- `type`: the data type of the effective address
- `qualifier`: an optional parameter that is either `restrictconst` or `restrict`. They will cause the underlying variable to be decorated with `__restrict const` and `__restrict` in C/C++, respectively. These qualifiers provide hints to the compiler to optimize the code with regard to the effects of pointer aliasing.

After declaring the pointer chains in our code, we have to determine the code region that we target to perform the transformation. The following lines describe how to use `begin` and `end` clauses with the `region` construct. The pointer chains that have been declared before in the current scope are the subject of transformation in subsequent regions.

```
#pragma pointerchain region begin
<...computation or data movement...>
#pragma pointerchain region end
```

Our proposed directive, `pointerchain`, is a language- and programming-model-agnostic directive. Although, in this paper, for implementation purposes, `pointerchain` is targeting C/C++ and OpenACC [82] programming models, one can utilize it for the Fortran language or target the OpenMP [83] programming model as well.

3.2.2 Condensed Version

Our two proposed clauses (`declare` and `region`) provide developers with the flexibility of reusing multiple variables in multiple regions. However, there exists a condensed version of `pointerchain` that performs the declaration and replacement process at the same time. The condensed version of `pointerchain` replaces the declared pointer chain with its effective address in the scope of the targeted region. It is placed in the `region` clauses. An example of a simplified version, enclosing a computation or data movement region, is shown below:

```
#pragma pointerchain region begin declare(variable [,variable]...)
<...computation or data movement...>
#pragma pointerchain region end
```

The condensed version is a favorable choice in comparison to the `declare/region` pair when our regions have few chains and we do not plan to reuse them in other parts of the code

in future. It leads to a clean, high quality code. Furthermore, utilizing the pair combination helps with the code readability, reduces the complexity of code, and expedites the porting process to OpenACC and OpenMP programming models. Potentially, the current modern compilers will be able to incorporate the condensed version of `pointerchain` with the OpenACC or OpenMP directives directly. The following example shows how the condensed version could be incorporated into the OpenACC programming model.

```
#pragma acc parallel pointerchain(variable [,variable]...)
<...computations...>
```

3.2.3 Sample Code

Listing 3.1 shows an example on how to use `pointerchain` in a source code. Lines 1-16 show the data structures for configuration in Figure 3.1, including the main object variable (`simulation`). Our computational kernel, Lines 26-33, initializes the position of every atom in 3D space in the system. These lines represent a normal, formal `for-loop` that has been parallelized by the OpenACC programming model. First, we declared our pointer chain (Line 19), then utilized the `region` clause to transfer the data to our target device (Lines 21-23), and finally, utilized the `region` clause to parallelize the `for` loop (Lines 26-33). Without `pointerchain`, parallelizing the `for`-loop requires to transfer every member of the chain to the device separately while retaining their relationship during the transfer. This will adversely impact the performance while making its implementation also challenging.

`Pointerchain` is capable of dealing with both pointers and scalar variables. Unlike pointers, dealing with the scalar variables requires more attention. The following example lays out the challenge in dealing with scalar variables. Suppose we want to change the number of atoms in the `atoms` structure (`simulation->atoms->N`). The `declare` clause

Listing 3.1: An example on how to use `pointerchain` directive for data transfer and kernel execution.

```
1  typedef struct {
2    ...
3    // position, momenta, and force in 3D space
4    double *positions[3];
5  } Traits;
6  typedef struct {
7    ...
8    // position, momenta, and force in 3D space
9    Traits *traits;
10 } Atoms;
11 typedef struct {
12   ...
13   // atom data (positions, momenta, ...)
14   Atoms* atoms;
15 } Simulation;
16 Simulation *simulation;
17
18 // Declaring the targeted pointer chain
19 #pragma pointerchain declare(simulation->atoms->traits->positions{
       double*})
20
21 #pragma pointerchain region begin
22 #pragma acc data enter copyin(simulation->atoms->traits->positions[0:N
       ])
23 #pragma pointerchain region end
24
25 // pointerchain region
26 #pragma pointerchain region begin
27 #pragma acc parallel loop
28 for(int i=0;i<nAtoms;i++) {
29   simulation->atoms->traits->positions[i][0] = ...;
30   simulation->atoms->traits->positions[i][1] = ...;
31   simulation->atoms->traits->positions[i][2] = ...;
32 }
33 #pragma pointerchain region end
```

extracts the value stored in this variable and records it in a temporary variable for the future references in the upcoming regions. However, when the region is done, the temporary variable has the most up-to-date value and while its corresponding chain is unaware of such update. Therefore, `pointerchain` updates the corresponding pointer chain with the updated temporary variable.

3.3 Implementation Strategy

To simplify the prototyping process, we have developed a Python script that performs a *source-to-source* transformation of the source codes annotated with the `pointerchain` directives. Our transformation script searches for all source files in the current folder and finds those annotated with the `pointerchain` directives. Then, they are transformed to their equivalent code.

Here is the overview of the transformation process. Upon encountering a `declare` clause, for each variable, a *local variable* with the specified `type` is created and initialized to the effective address of our targeted pointer chain (variable `name`). If `qualifier`s are set for a chain, they will also be appended. Any occurrences of pointer chains in between `region begin` and `region end` clauses are replaced with their counterpart local pointers announced before by `declare` clauses in the same functional unit.

Scalar variables (i.e., `simulation->atoms->N`) are treated differently in `pointerchain`. It starts by defining a local temporary variable to store the latest value of the scalar variable. Then, all occurrences of the scalar pointer chain within the region are replaced with the local variable. Finally, after exiting the region, the scalar pointer chain variable is updated with the latest value in the local variable.

Introducing new local pointers to the code has some unwelcome implications on the usage

of the stack memory. They are translated into a memory space on the call stack of the calling function. `Pointerchain` has alleviated this burden by reusing the local variables that were extracted from the chain instead of reusing the chains over and over again. This is especially beneficial when our application targets GPU devices. Section 4 discusses the implications imposed by `pointerchain` by several aspects including code generation, performance, and stack memory layout. For a detailed discussion, please refer to Section 4.

3.4 C++ Pointers

A number of well-known simulation frameworks (e.g., LAMMPS [93]) are developed in the C++ language. Such frameworks, in some cases, heavily utilize the getter/setter functions to deal with the pointers in their codes. An example of the getter/setter functions is shown in Listing 3.2.

Listing 3.2: An example of setter/getter functions

```
1  class Simulation {
2  private:
3      // ...
4      AtomStruct *atoms;
5
6  public:
7      // ...
8
9      AtomStruct *getAtoms() {
10         return atoms;
11     }
12
13     void setAtoms(AtomStruct *as) {
14         atoms = as;
15     }
16 };
```

In such cases, similar to the previous cases in C, `pointerchain` is effective since it is dealing with the memory addresses. As long as the chain has an effective address, `pointerchain` operates as expected. After extracting the effective address and reserving it in a

temporary variable, such temporary variable is a good candidate to replace the chain in our kernels. For such cases, we can easily declare our chains, and then, use it accordingly within the regions. Listing 3.3 shows an example of how to use `pointerchain` when we utilize C++ approach. This example is based on the Listing 3.1.

Listing 3.3: An example using `pointerchain` with setter/getter functions.

```
1  // Declaring the targeted pointer chain
2  #pragma pointerchain declare(getSimulation()->getAtoms()->getTraits()->
       getPositions(){double*})
3
4  #pragma pointerchain region begin
5  #pragma acc data enter copyin(getSimulation()->getAtoms()->getTraits()
       ->getPositions()[0:N])
6  #pragma pointerchain region end
7
8  // pointerchain region
9  #pragma pointerchain region begin
10 #pragma acc parallel loop
11 for(int i=0;i<nAtoms;i++) {
12   getSimulation()->getAtoms()->getTraits()->getPositions()[i][0] = ...;
13   getSimulation()->getAtoms()->getTraits()->getPositions()[i][1] = ...;
14   getSimulation()->getAtoms()->getTraits()->getPositions()[i][2] = ...;
15 }
16 #pragma pointerchain region end
```

Moreover, C++ also provides *reference variables (or references in short)*. References are *an alias (in other words, an alternative)* to another variable in our application. Since references will eventually reduce to an address in memory, we are able to exploit `pointerchain` with C++ references as well.

Chapter 4

Deep Copy and Microbenchmarking

Previously published content:

1. Ghane, Millad, and Chandrasekaran, Sunita, and Cheung, Margaret S., "pointerchain: Tracing pointers to their roots - A case study in molecular dynamics simulations." *Parallel Comput.*, 2019. [37]
2. Ghane, Millad, and Chandrasekaran, Sunita, and Cheung, Margaret S., "Assessing Performance Implications of Deep Copy Operations via Microbenchmarking." *arXiv preprint*, 2019. [35]

In this chapter, we will discuss what deep copy is, how it is a challenge in HPC, and what the current techniques to perform deep copy are. We will also introduce our proposed directive, `pointerchain`, as one of the techniques to perform the deep copy operation. Moreover, a set of benchmarks is introduced to evaluate the current deep copy approaches.

4.1 Semantics of Deep Copy

As discussed in Chapter 3, scientific applications utilize nested data structures in their design. Nested data structures are composed of a set of simple or complex member variables. The

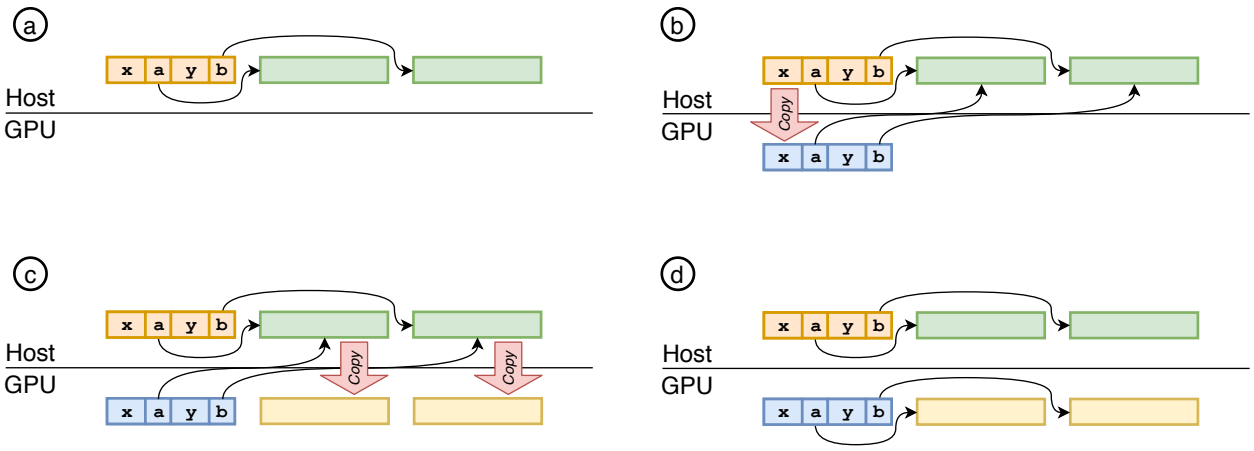


Figure 4.1: Steps to perform a deep copy operation when the targeting device is a GPU. The horizontal line separates the memory spaces between the host and the GPU. (a) initialize the data structures; (b) copy the main structure to the GPU; (c) copy other nested data structures to the device; (d) fix corresponding pointers in every data structure.

simple member variables are those members with primitive data types (e.g., `int`, `float`, `double` in C/C++). However, the complex member variables are those that are user-defined data structures themselves. The situation gets complicated as the complex member variable may itself possess another complex data structure within itself.

The common approach to utilize complex member variables in C/C++ for such cases is to define them as pointers. Since the array size is not known at the compilation time, they have to be allocated at run time. This causes their addresses in memory to be known only at execution time. This is not an ideal case for heterogeneous platforms with separate memory address spaces. Figure 4.1 illustrates the necessary steps required to perform the deep copy. After initializing (Step ***a***) and transferring (Steps ***b*** and ***c***) the structure from the host to the device, the pointers on the main structure hold illegal addresses. They still point to the same memory address on the host, which is inaccessible on the device. We have to fix this issue by reassigning the pointers to their correct corresponding addresses on the device (Step ***d*** in Figure 4.1).

Deep copy, as described in [81], can be categorized into two groups: 1) Full Deep Copy;

2) Selective (Partial) Deep Copy. A full deep copy operation copies a data structure with all of its nested data structure to the device. As a result, a replica of the whole structure is available on the device. The process discussed in Figure 4.1 demonstrates a full deep copy operation. However, a full deep copy is not always an appropriate approach and we need mechanisms to perform a partial copy operation. In those cases, not all variable members of a data structure are accessed during a kernel execution on the device. As a result, there is no need to transfer them to the device. Consider the example in Figure 4.1. If our kernel is only accessing array `x->a`, we should not copy array `x->b` to the device and keep it on the host. This will significantly improve performance of the copy operation. This is an example of a selective deep copy operation.

4.2 Methodology

This section is dedicated to discussing our methodology in benchmarking the deep copy operations for two different scenarios, *Linear* and *Dense*. Each scenario is tested with various *transfer* and *layout* schemes. In the following, we will discuss the detailed description of each scenario and scheme. All the source codes of our microbenchmark are accessible on Github[1].

4.2.1 Linear Scenario

In this case, we will design a set of experiments to study the effect of nesting depth on the performance of applications. Figure 4.2 shows the data layout for the Linear scenario. All the data structures in this scenario have similar member variables. They consist of two integer variables (`nA` and `nLnext`), a floating-point array (`A`), and a pointer to the next nested data structure (`Lnext`). The main data structure is the the data structure at level 0, which is

[1] `https://github.com/milladgit/deepcopy-benchmark`.

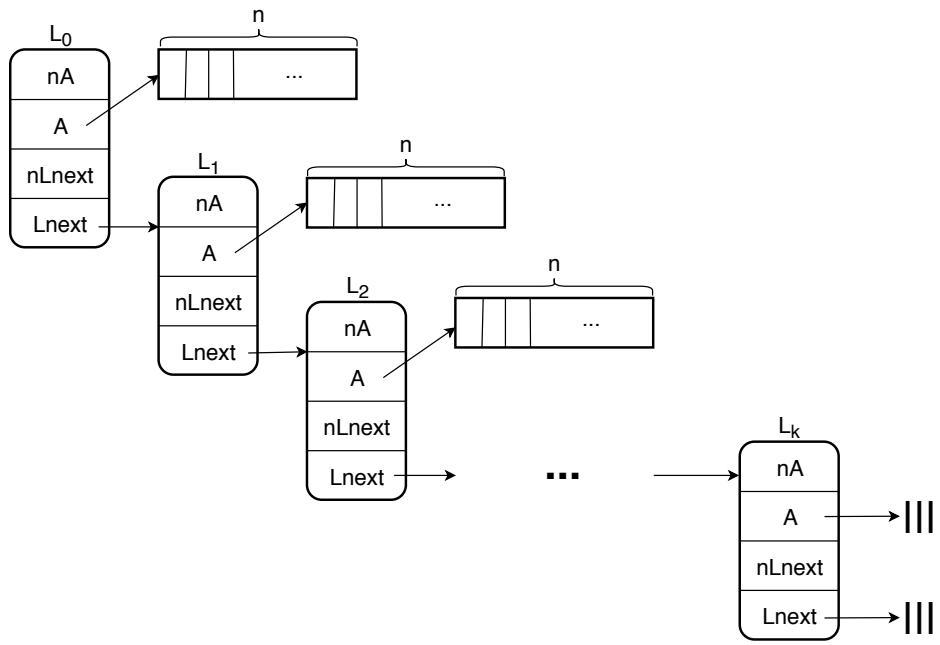


Figure 4.2: The overview of the Linear scenario as described in Section 4.2.1. Increasing k increases the depth of our nested data structures.

designated by L_0. Our design for this scenario has two parameters: k and n. The parameter k controls the depth of our data layout and the parameter n controls the length of the extra payload that we have assigned to each nested data structure.

In order to perform these experiments, we developed a Python script that accepts an integer k as an input parameter and generates a total of k C++ source files with 1 to k nested data structures, similar to the configuration in Figure 4.2. The parameter n is an input to the main program of each C++ source file.

4.2.1.1 Transfer Schemes

For Linear scenarios, we have investigated three options to transfer the data structures to the device:

1. *UVM:* Since we are targeting NVidia GPUs, we utilized Unified Virtual Memory (UVM) technology [57] for memory allocations. UVM allows developers to allocate memories

that are accessible by both host and device. The PGI compiler provides UVM allocations with `-ta=tesla:managed` flag at the compile time for every memory allocation requests (`mallocs`) by the application.

2. *Marshalling data structures:* We developed a method to enable the marshalling/demarshalling of structures at the run time of the application using `acc_attach`/ `acc_detach` API methods in OpenACC. Algorithm 1 shows the steps our implementation takes to implement the marshalling. At the beginning, developers determine how big the whole tree is (the main data structure with all of its nested data structures). Then, that amount of memory is allocated. Afterwards, any subsequent memory allocation requests from the program are responded to by returning the next available space from our allocated buffer. These steps compact all the allocated memories into a contiguous space in the memory. This approach is the ideal case for transferring a complicated data structure tree in one batch instead of multiple batches for each structure. After transferring the whole buffer to the device, we have to call `acc_attach` on each pointer on the device so that the pointers on the device point to a correct memory address. The demarshalling process is performed exactly in the reverse order of the marshalling algorithm. It is highly probable that the implementations of deep copy in different compilers follow similar marshalling approach.

3. `pointerchain:` the proposed directive as described in Section 3.

4.2.1.2 Layout Schemes

Three separate layout schemes are introduced for our Linear scenario. The layout schemes differ in whether the A arrays in Figure 4.2 are allocated or not, and whether they will be transferred to the device and utilized or not.

Algorithm 1 Marshalling algorithm

```
1: function MARSHALLIZE(struct)
2:     n ← DETERMINETOTALBYTES(struct)
3:     buff ← Allocate n bytes buffer on heap
4:     requestList ← []
5:     for memory allocation of size w do
6:         Append the allocation request to the requestList
7:         Return a pointer to w bytes from buff
8:     end for
9:     Transfer buff to the device
10:    for req in requestList do
11:        ACC_ATTACH(req)
12:    end for
13: end function
```

1. *allinit-allused*: In this scheme, all the A arrays in all levels allocate n elements and they are accessed on the GPU. Our kernel scales all elements of the A arrays with an arbitrary number. This layout scheme helps us understand the efficiency of each transfer scheme when a full deep copy is inevitable.

2. *allinit-LLused*: Similarly, we allocate n elements for all the A arrays, however only the A arrays of the last level are utilized within a kernel on the device. This scheme helps us understand how selective deep copy improves the performance when the kernels target only a subset of data structures on the device.

3. *LLinit-LLused*: In this scheme, only the A array in the last-level (L_k) allocates memory space. This scheme helps us understand which transfer scheme performs the best in a long chain of pointers. This is a dominant scheme in scientific applications like molecular dynamics simulations [37].

4.2.1.3 Data Size

The amount of data generated by our tree of data structures for each layout scheme, as shown in Figure 4.2, is as following. For the *allinit-allused* and *allinit-LLused* cases, the size

of our configuration, as a function of n and k, is

$$\begin{aligned} DataSize(k,n) &= \sum_{i=1}^{k}(24+8n) \\ &= 24k + 8nk \end{aligned} \tag{4.1}$$

where 24 is the size of the L_i structures and 8 is the size of an element in A in bytes (for double-precision floating-point numbers).

For the *LLinit-LLused* case, the data size can be computed as follows

$$\begin{aligned} DataSize(k,n) &= \sum_{i=1}^{k} 24+8n \\ &= 24k + 8n \end{aligned} \tag{4.2}$$

4.2.2 Dense Scenario

In the dense scenario, the intermediate pointers are an array of objects instead of a single object. Figure 4.3 illustrates the dense scenario. This configuration provides a dense tree of data, which the size of the data will grow exponentially with small changes in both parameters in our design. The parameter q describes number of elements in the intermediate arrays L_i, and the parameter n determines the number of elements in the A arrays.

4.2.2.1 Transfer Scheme

In comparison to the Linear scenario, transferring the data structure tree represented in Figure 4.3 is more complicated. For the marshalling and `pointerchain` approaches, extra work is required to make the intermediate pointers legal on the device so that they could be dereferenced correctly. In cases similar to Dense, utilizing the `pointerchain` directive to

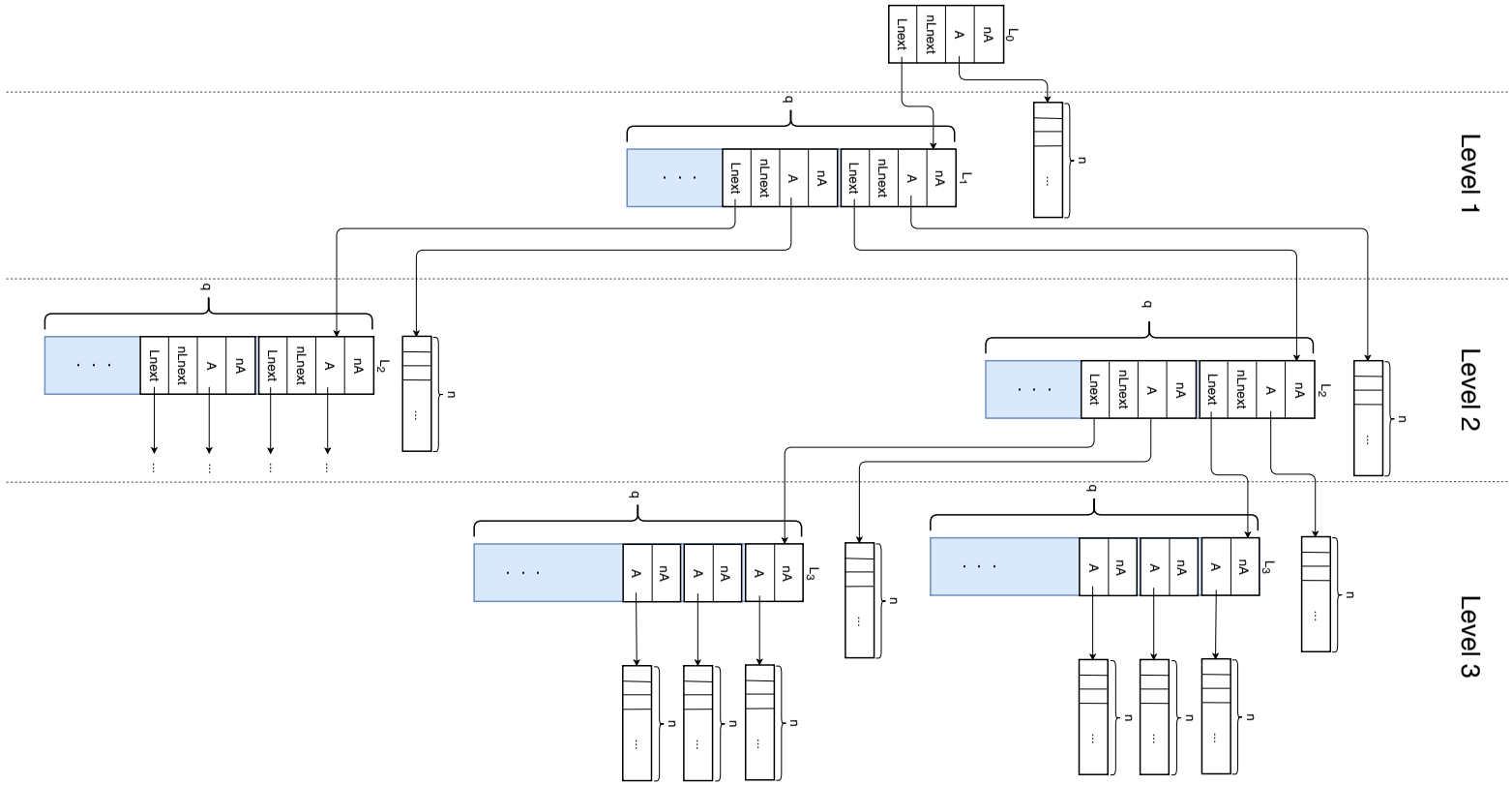


Figure 4.3: The overview of the Dense scenario as described in Section 4.2.2. Increasing q increases the data size exponentially. Unlike the Linear scenario, the depth is fixed to three levels. The dots in the figure show the recursive nature of the data structure.

perform a *full deep copy* operation is not a viable option due to the increasing number of intermediate pointers, which grows exponentially in this case.

Similar to the Linear scenario, the data transfer to the device is performed with UVM, marshalling, and `pointerchain`. Each scheme is described in detail in Section 4.2.1.1.

4.2.2.2 Layout Scheme

In the Dense scenario, an arbitrary index of each intermediate array L_i (in our case, the last element of the array) is chosen, and then, its associated A array is transferred to the device to perform the computational kernel. As an example, consider the configuration in Figure 4.3. For such nested data structure, the kernel that parallelizes the code will look like Listing 4.1, where `q` is the number of elements in the intermediate arrays L_i, and `a0` is the main structure at the first level.

Listing 4.1: The scaling kernel used in our Dense scenario, where q is the number of elements in the intermediate arrays L_i.

```
1 for(int i=0;i<N;i++)
2     a0->Lnext[q-1].Lnext[q-1].Lnext[q-1].A[i] *= scale;
```

4.2.2.3 Data Size

The amount of data generated by the data structure tree in the Dense scenario, as shown in Figure 4.3, is very sensitive to the input parameters, q and n. Small changes in these parameters lead to significant increases in the data size. Equation 4.3 shows the amount of data generated in bytes for our configuration in recursive form:

$$DataSize(q, n, D) = 24 + 8n + q \times DataSize(q, n, D-1)$$
$$DataSize(q, n, 0) = 12 + 8n \tag{4.3}$$

where 24 is the size of L_i structures, 8 is the size of each element in array A, q is the length of the intermediate arrays, and D is the depth of our nested data structure. $DataSize(q, n, 0)$ refers to the size of our last-level data structures (the L3 structures in Figure 4.3). For our experiments in this paper, the maximum value of D is set to 3. Please note that the last-level data structure is half of the original structure in size.

4.3 Experimental Setup

Located at the University of Houston, Sabine [97] clusters host HPE compute nodes. Each system is equipped with two Intel Xeon E5-2680v4 CPUs, with 28 logical cores, and 256 GB of host RAM. Sabine has both NVidia P100 and V100 GPU architectures. The P100 systems have 16 GB global memory with 4 MB L2 caches. The V100 GPUs also have 16 GB global memory while their L2 caches are 6 MB. Our software environment, for both system, includes the PGI compiler 18.4.

For the Linear scenario, we developed a Python script that accepts an integer number, *count*, as input and generates a set of source codes in C++ for $k \in [2, count]$. Each source code is a stand-alone application. The data structure tree depicted in Figure 4.2 is generated statically for each k to allow the compiler apply optimizations on the source codes efficiently. For each k, our script generates nine files: three transfer schemes by three layout schemes. As an example, suppose we pass 10 to our Python script. Then, the total number of files generated by our script is 81 ($(count - 2 + 1) \times 3 \times 3 = 81$).

Algorithm 2 Main program steps

```
1: function MAIN(argc, argv)
2:     1- Allocate memory for whole tree structure
3:     2- Initialize the tree
4:     3- Transfer the tree to the device with a transfer scheme
5:     4- Run the kernel once
6:     5- Transfer the tree back to the host
7:     6- Check the results
8:     7- Measure the wall-clock time
9: end function
```

For the Dense scenario, we developed three different transfer schemes (UVM, marshalling, and `pointerchain`) to perform the selective deep copy. Each scheme accepts two inputs, n and q, which they were previously described in Section 4.2.

Algorithm 2 displays the steps that each benchmark application takes. At the beginning of the application, the memory for the whole data structure tree is allocated and it is initialized with arbitrary values. Then, the whole data structure is transferred to the device based on the various transfer schemes explained in Section 4.2. This allows us to run a computational kernel. The kernel scales every elements of the array A by a constant value. Based on the chosen layout scheme, whether it is *allused* or *LLused*, all or last-level A-arrays are scaled, respectively. After running the kernel, the tree is transferred back to the host and the results are checked.

For both Linear and Dense scenarios, two different metrics are measured: (a) the wall-clock time of the whole application, (b) the kernel execution time. The wall-clock time is measured to investigate the effect of each transfer scheme on each different scenario. The kernel execution time is measured to give us an insight about how different data layouts affect kernel's performance. Not only the execution time, but also the total number of instructions generated by the compiler will be affected by different transfer schemes.

We used Google Benchmark [44] to measure the execution time (i.e., the kernel and the wall-clock time). It is a lightweight, powerful framework for benchmark functions. Through

a set of preliminary testing, the framework learns how many iterations are required to be performed so that a consistent result within a low error margin is observed. Each test case is implemented as a function, and then, the whole function is benchmarked with Google Benchmark. For the results of the kernel time, we benchmarked only the kernel computations on Step 4 (line 5) of Algorithm 2.

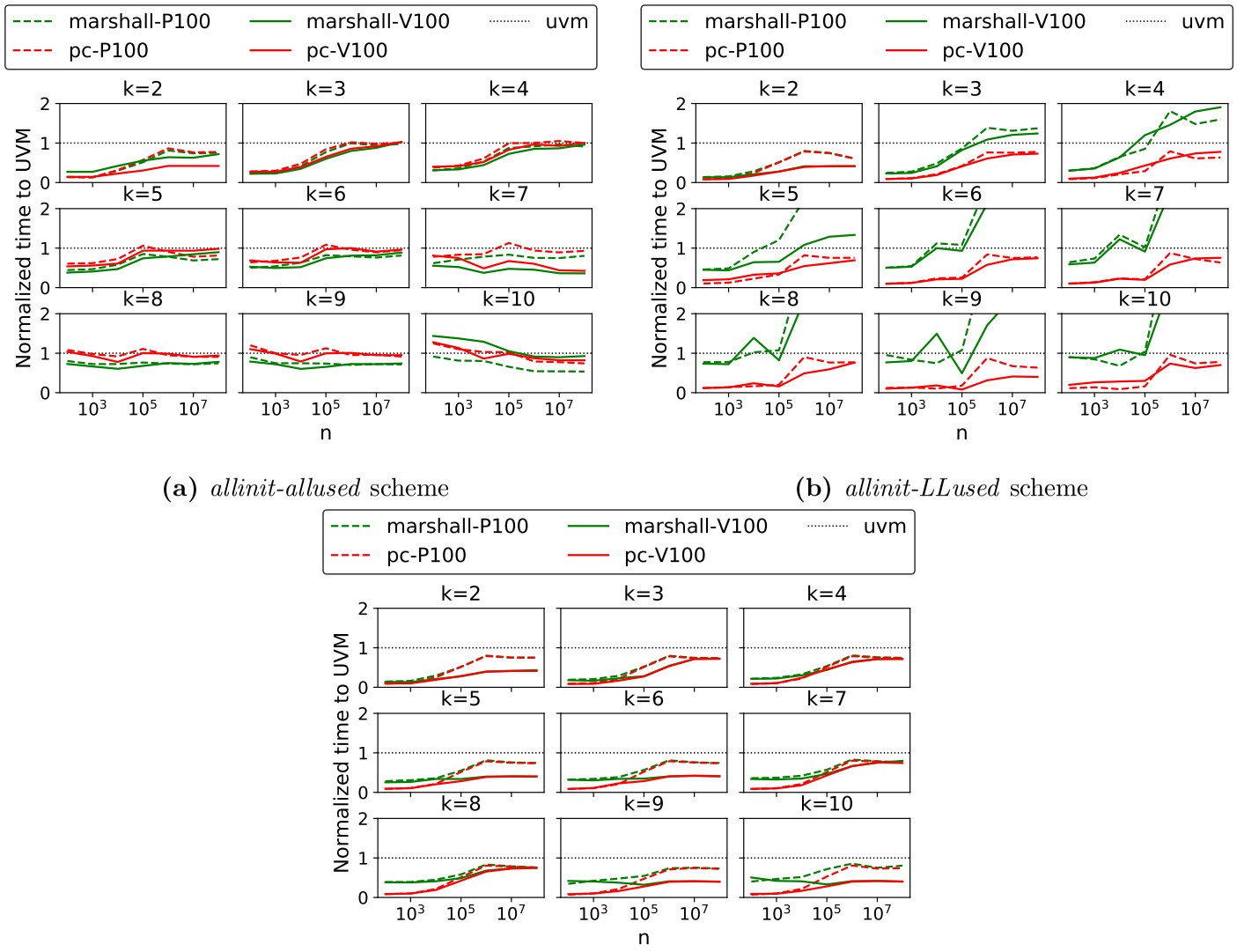
4.4 Results

We performed the experiments in this section on the Sabine systems (P100 and V100). We measured the wall-clock and kernel time of the experiments designed for the Linear scenario. Figure 4.4 shows the wall-clock time for different number of levels and different layout schemes. Results are normalized with respect to the UVM approach.

4.4.1 Linear Scenario

4.4.1.1 Wall-clock Time

Results for the *allinit-allused* transfer scheme reveal how increasing the parameter n leads to performance loss for all values of k. As we increase the total size of the tree (increasing both n and k), there is no performance loss when UVM is utilized, and it has a chance to be a viable option in comparison to other methods. Furthermore, UVM is a feasible approach to transfer data between host and device when applications are dealing with huge amounts of data. It provides developers more productivity with the same level of performance when we are targeting huge data. However, when n is moderately large ($n < 10^5$) and the chain length (k) is small, marshalling and `pointerchain` outperform UVM. Furthermore, there is no subtle difference between different architectures (P100 and V100) for the *allinit-allused* scheme.



(a) *allinit-allused* scheme

(b) *allinit-LLused* scheme

(c) *LLinit-LLused* scheme

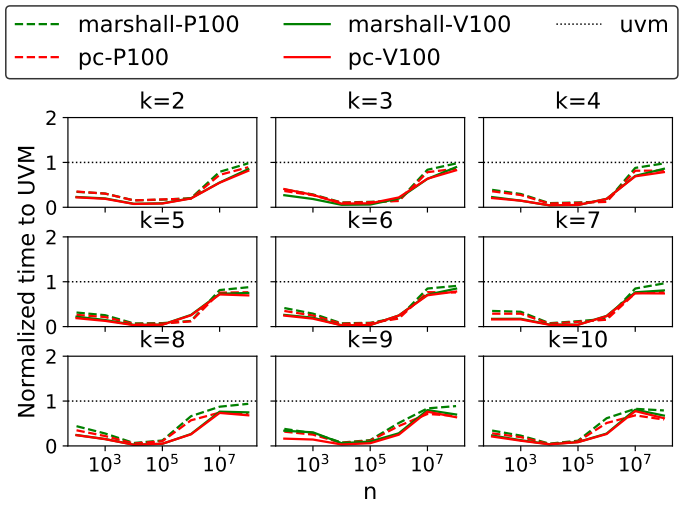
Figure 4.4: Normalized wall-clock time with respect to UVM for the Linear scenario.

On the other hand, the *allinit-LLused* scheme is more susceptible to the transfer scheme rather than the underlying architecture. As n increases in the size, the gap between marshalling and `pointerchain` increases. For larger k values, `pointerchain` outperforms marshalling and UVM. Thus, `pointerchain` is the better option for a deep copy operation in comparison to the other two options when we are dealing with huge data sets. As k increases, the marshalling scheme performs worse while the performance of `pointerchain` is not affected and remains constant. There is no notable difference between different architectures, and the transfer schemes determines the performance. It is the underlying data transfer medium, in our case the PCI-E bus, that determines the upper bound of the performance.

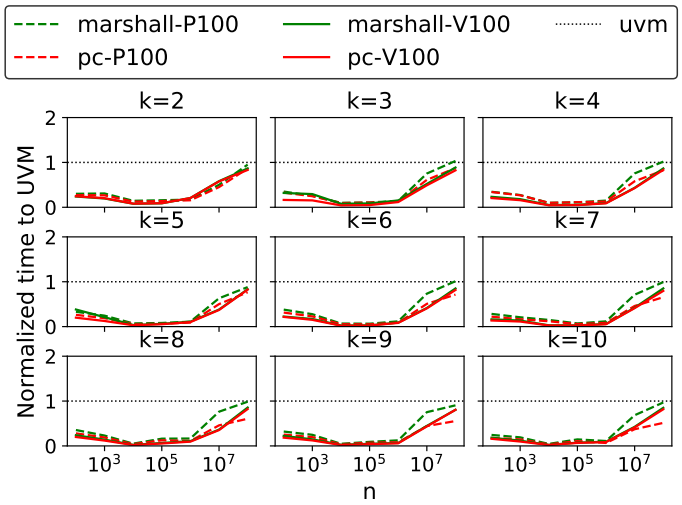
Finally, for the *LLinit-LLused* scheme, UVM has the worst performance results. The results show how in cases that our kernel targets an array at the last-level data structure, utilizing either marshalling or `pointerchain` leads to better performance results. The `pointerchain` scheme shows promising results when $n < 10^5$. However, for $n > 10^5$, the architecture design determines the winner. The V100 architecture shows 2X improvements in performance for marshalling and `pointerchain` schemes, while P100 was able to show 1.25X improvement. For all values of k and n, `pointerchain` performed better than marshalling.

4.4.1.2 Kernel Execution Time

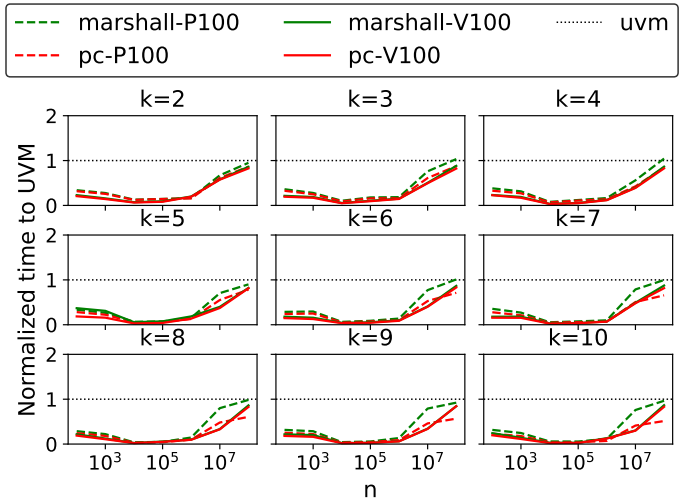
Figure 4.5 shows the normalized kernel time with respect to UVM for different level counts and different layout schemes. There is no subtle difference among different transfer schemes, different layout schemes, and different architectures. Mostly, for all values of n and k, all results follow the same trend. However, we observe the best performance when $n \in [10^4, 10^6]$.



(a) *allinit-allused* scheme



(b) *allinit-LLused* scheme



(c) *LLinit-LLused* scheme

Figure 4.5: Normalized kernel time with respect to UVM for the Linear scenario.

Table 4.1: Total data size of our data structure tree as defined in the Linear scenario for the *allinit-allused* scheme. Equation 4.1 was used to calculate these numbers. *The first row is in KiB, while the rest of the numbers is in MiB.*

	k								
n	2	3	4	5	6	7	8	9	10
10^2	*1.61 KB*	*2.41 KB*	*3.22 KB*	*4.02 KB*	*4.83 KB*	*5.63 KB*	*6.44 KB*	*7.24 KB*	*8.05 KB*
10^3	0.02 MB	0.02 MB	0.03 MB	0.04 MB	0.05 MB	0.05 MB	0.06 MB	0.07 MB	0.08 MB
10^4	0.15 MB	0.23 MB	0.31 MB	0.38 MB	0.46 MB	0.53 MB	0.61 MB	0.69 MB	0.76 MB
10^5	1.53 MB	2.29 MB	3.05 MB	3.81 MB	4.58 MB	5.34 MB	6.10 MB	6.87 MB	7.63 MB
10^6	15.26 MB	22.89 MB	30.52 MB	38.15 MB	45.78 MB	53.41 MB	61.04 MB	68.66 MB	76.29 MB
10^7	152.59 MB	228.88 MB	305.18 MB	381.47 MB	457.76 MB	534.06 MB	610.35 MB	686.65 MB	762.94 MB
10^8	1525.88 MB	2288.82 MB	3051.76 MB	3814.70 MB	4577.64 MB	5340.58 MB	6103.52 MB	6866.46 MB	7629.39 MB

Table 4.1 shows the total size of our data structure tree as we change k and n. For all ks, while $n < 10^5$ the whole data fits in the L2 cache of P100 and V100 GPUs. As we increase n, the L2 cache is not big enough anymore, which results in the mandatory cache eviction process, subsequently, we lose performance. This is the reason that we observe an increasing trend in the execution time in Figure 4.5. This confirms our finding: *when we are dealing with data structures with large sizes, there is no subtle difference in performance between UVM and other transfer schemes for complex data structures.*

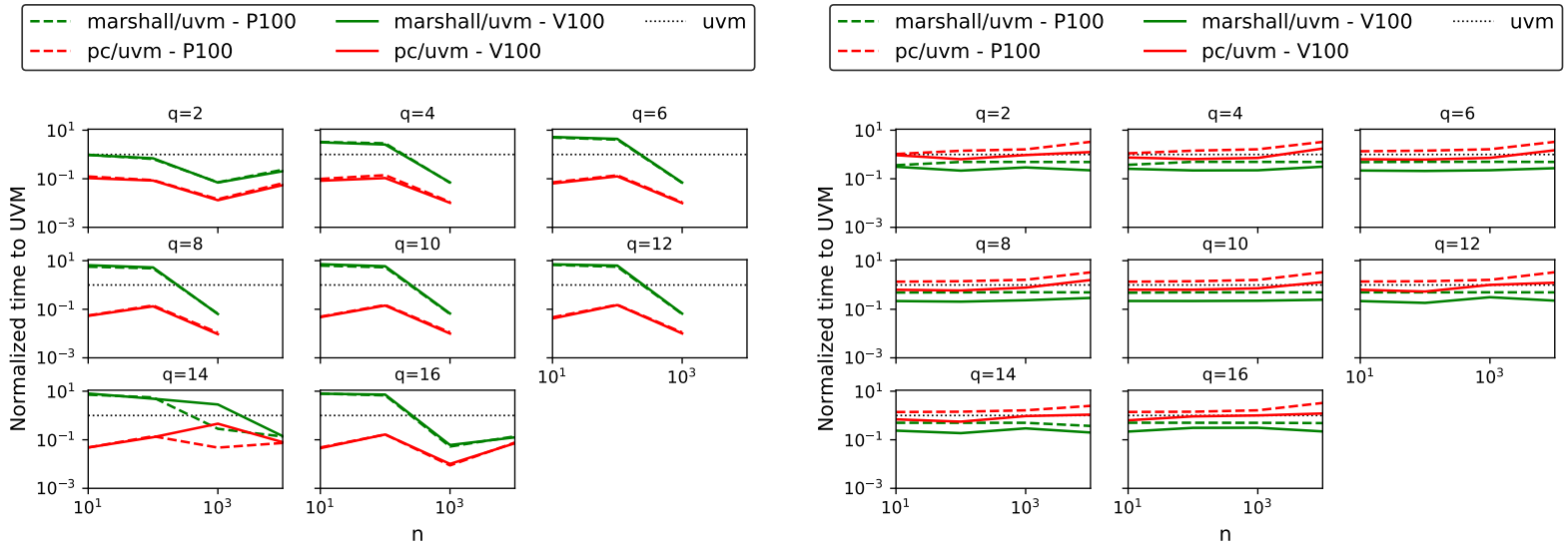
4.4.2 Dense Scenario

We measured the wall-clock and kernel time of the experiments designed for the Dense scenario. Figure 4.6 shows the normalized wall-clock time and kernel time with respect to UVM for different level count and different layout schemes.

4.4.2.1 Wall-clock Time

The key factor that determines the performance of the whole application is the transfer scheme. The `pointerchain` scheme performs consistently better in comparison to the marshalling. In cases like $n = 10$ and $n = 100$, `pointerchain` basically shows two orders of magnitude performance improvements in comparison to marshalling. In such cases, UVM shows close to 10X improvement over marshalling.

However, as q increases, the performance gap between `pointerchain` and marshalling shrinks. Moreover, Figure 4.6 shows how in the Dense scenarios, the underlying architecture does not have any contributions to the performance. It is the transfer scheme that determines the performance. The reason behind such performance deficiency of the marshalling scheme is the extra job required to be done to ensure the pointer consistency on the device. For each pointer, we are required to fix the address in the structure to point to a correct location on



(a) *Wall-clock time* **(b)** *Kernel time*

Figure 4.6: Normalized wall-clock time and kernel time to UVM for Dense scenario. The Y axes are in logarithmic scale. Lower is better.

Table 4.2: Total data size of our data structure tree as defined in the Dense scenario. Equation 4.3 was used to calculate these numbers.

	q							
n	2	4	6	8	10	12	14	16
10^1	1.43 KB	7.88 KB	0.02 MB	0.05 MB	0.10 MB	0.17 MB	0.26 MB	0.39 MB
10^2	0.01 MB	0.07 MB	0.20 MB	0.45 MB	0.86 MB	1.46 MB	2.29 MB	3.39 MB
10^3	0.11 MB	0.65 MB	1.98 MB	4.47 MB	8.49 MB	0.01 GB	0.02 GB	0.03 GB
10^4	1.14 MB	6.49 MB	0.02 GB	0.04 GB	0.08 GB	0.14 GB	0.22 GB	0.33 GB
10^5	0.01 GB	0.06 GB	0.19 GB	0.44 GB	0.83 GB	1.40 GB	2.20 GB	3.26 GB

the memory space of the device.

4.4.2.2 Kernel Execution Time

Figure 4.6 also demonstrates the performance of the kernel with respect to different transfer schemes introduced in Section 4.2. Despite no subtle differences, the marshalling scheme leads to more performance friendly data layout in comparison to `pointerchain` on both architectures. While kernels that are executed on the marshalled data perform better than their UVM counterparts, the `pointerchain` scheme suffers some performance loss. Consequently, in cases that a kernel is executed multiple times on the same data, the data layout of the marshalling scheme results in a better performance. Such an effect is due to the cache friendly layout of our implementation for marshalling. The marshalling scheme places the arrays as close as possible to the pointers that points to them, however, this is not necessarily the case for the `pointerchain` scheme. In `pointerchain`, the arrays are scattered around the global memory of GPUs and they do not necessarily reside in the same memory page as the pointer itself.

Table 4.2 shows the total size of our data structure tree as we change q and n. Increasing n will exponentially increase the data size. However, this is not correct for q. This observation reveals how the size of the internal array affects the total size more than the depth of the tree. This is the reason we observe similar patterns for all qs in Figure 4.6. This figure reveals how the performance remains consistent across all values of q but increasing n leads

to a reduction in performance.

4.4.3 Instruction Count

The process of dereferencing pointers generates a set of instruction to retrieve the effective address of the pointer. For Tesla V100, the PGI compiler generates 2 instructions per each dereference operation: 1) an instruction to load the address from global memory to a register (`ld.global.nc .u64`); 2) an instruction to convert the virtual address to a physical address on the device (`cvta.to.global.u64`). For every chain, the processor has to execute above instructions to extract the effective address.

Tables 4.3 and 4.4 show total number of generated instructions by the PGI compiler for the Linear and Dense scenarios, respectively. To count the number of instructions, we generated the PTX files by enabling the `keep` flag at compile time (`-ta=tesla:cc70,keep`). Then, we counted number of lines (LOC) in the generated PTX file.

The results for the Linear scenario, as shown in Table 4.3, reveals up to 31% reduction in the generated code for GPUs. The LOC for the `LLused` schemes remains constant since we are basically reducing any pointer chains in our application to one pointer. However, for UVM and marshalling schemes, as k increases, the total generated code for them increases as well since we have to dereference the chain of pointers. For the *allinit-**LLused*** *and* *LLinit-**LLused*** schemes, one can observe how the LOC increase by two lines between two consecutive ks. For the *allinit-allused* scheme, since we are dealing with multiple pointer chains, the trend is not linear, however we save more instructions in this case. Table 4.4 shows similar results for the Dense scenario. We have two observations: 1) The marshalling scheme did not increase the number of instructions with respect to UVM. 2) `pointerchain` led to a 25% reduction in generated instructions.

Table 4.3: Total instruction generated by the PGI compiler (for Tesla V100) for the Linear scenario. *Mar.* and *PC* refer to the marshalling and `pointerchain` schemes, respectively. The numbers in parentheses show the increase with respect to UVM.

	allinit-allused			allinit-LLused			LLinit-LLused		
k	UVM	Mar. (%)	PC (%)	UVM	Mar. (%)	PC (%)	UVM	Mar. (%)	PC (%)
2	62	62 (0%)	60 (-3%)	62	62 (0%)	60 (-3%)	62	62 (0%)	60 (-3%)
3	70	70 (0%)	67 (-4%)	64	64 (0%)	60 (-6%)	64	64 (0%)	60 (-6%)
4	78	78 (0%)	74 (-5%)	66	66 (0%)	60 (-9%)	66	66 (0%)	60 (-9%)
5	88	88 (0%)	81 (-8%)	68	68 (0%)	60 (-12%)	68	68 (0%)	60 (-12%)
6	100	100 (0%)	88 (-12%)	70	70 (0%)	60 (-14%)	70	70 (0%)	60 (-14%)
7	114	114 (0%)	95 (-17%)	72	72 (0%)	60 (-17%)	72	72 (0%)	60 (-17%)
8	130	130 (0%)	102 (-22%)	74	74 (0%)	60 (-19%)	74	74 (0%)	60 (-19%)
9	148	148 (0%)	109 (-26%)	76	76 (0%)	60 (-21%)	76	76 (0%)	60 (-21%)
10	168	168 (0%)	116 (-31%)	78	78 (0%)	60 (-23%)	78	78 (0%)	60 (-23%)

Table 4.4: Total instruction generated by the PGI compiler (for Tesla V100) for the Dense scenario. *Mar.* and *PC* refer to the marshalling and `pointerchain` schemes, respectively. The numbers in parentheses show the increase with respect to UVM.

	UVM	Mar. (%)	PC (%)
Dense	80	80 (0%)	60 (-25%)

Chapter 5

CoMD: A Case Study in Molecular Dynamics and Our Optimization Strategies

Previously published content:

1. Ghane, Millad, and Chandrasekaran, Sunita, and Cheung, Margaret S., "pointerchain: Tracing pointers to their roots - A case study in molecular dynamics simulations". *Parallel Comput.*, 2019. [37]

The Co-Design Center for Particle Applications (COPA) [22], a part of Exascale Computing Project (ECP), has established a set of proxy applications for real-world applications [52] that are either too complex or too large for code development. The goal of these proxy applications is for the vendors to understand the application and its workload characteristics and for the application developers to understand the hardware. The tools and software developers need them for expanding libraries, compiler and programming models as well.

CoMD [21] is a proxy application of classical molecular dynamics simulations, which

represents a significant fraction of the workload that the Department of Energy (DOE) is facing [107, 71]. It computes short-range forces between each pair of atoms whose distance is within a cutoff range. It does not include long-range and electrostatic forces inherently. The evaluated forces are used to update atoms characteristics (position, velocity, force, and momenta) via numerical integration [89].

Computations in CoMD are divided into three main kernels for each time step: force computation, advancing position, and advancing velocity. The latter two kernels are considered embarrassingly parallel (EP) kernels since their associated computations are performed on each atom independently. The velocity of an atom is updated according to the exerted force on that atom, and the position of an atom is updated according to its updated velocity. The most time-consuming phase, however, is the force computation phase.

Computing the forces that atoms exert on each other follows the equations of Newton's Laws of Motions, which is based on the distance between every pair of atoms. However, searching for neighbors of all atoms requires an $O(N^2)$ computation complexity, which is utterly inefficient. To overcome such an issue, CoMD exploits the link-cell method. It partitions the system space by a rectangular decomposition method in such a way the size of each cell exceeds the cutoff range in every dimension. In this way, neighbors are extracted from the cell containing the atom and the 26 neighboring cells around that cell. Through using link-cells, the computational complexity decrease to $O(27 \times N)$, which essentially is linear. Figure 5.1 depicts an example of the cutoff range in a two dimensional arrangement in the presence of the corresponding link-cells.

Algorithm 3 describes the CoMD phases. It follows the Verlet algorithm [105] in MD simulations. In each time step, the velocity is advanced at an interval of a half-time step, and then the position is updated for the full-time step based on the computed velocities. Using updated velocity and position updates the forces for all atoms. Later, velocities are updated for the remainder of the time step to reflect a full time step.

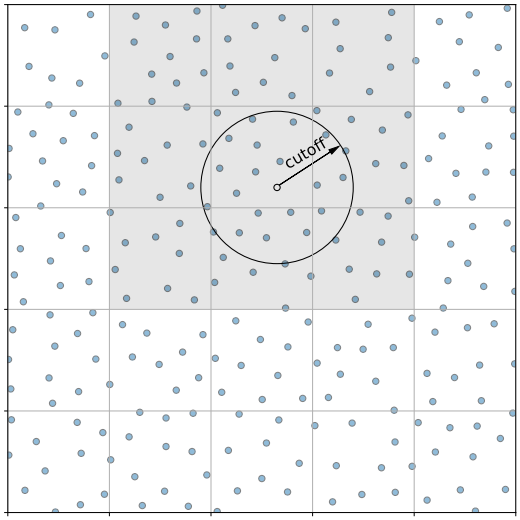


Figure 5.1: Link-cell decomposition of space [105, 14]. The cutoff range is also shown for a specific atom. The 2D space is divided into 5-by-5 cells. The cell containing the atom and its neighboring cells are displayed in gray.

Updating the position of atoms leads to the migration of atoms among neighbor cells and, in many cases, among neighbor processors. After position updates, link-cells are required to be updated locally (intra node/processor) and globally (inter nodes/processors) in each time step too. This process is guaranteed to be done by the RedistributeAtoms function of Algorithm 3.

Force calculations in the Verlet algorithm are derived from the gradient of the chosen potential function. A well-known interatomic potential function that governs relation of atoms and is extensively used in MD simulations is Lennard-Jones (LJ) [51]. CoMD supports an implementation of LJ to represent force interaction between atoms in a system. The LJ force function will be called inside the `ComputeForce` kernel in Verlet algorithm (Algorithm 3). Moreover, CoMD also supports another potential function known as the Embedded Atom Model (EAM), which is widely used in metallic simulations. In this dissertation, due to its simplicity in design and wide use in protein folding applications, we will be focusing on the LJ potential function.

Algorithm 3 MD timesteps in Verlet algorithm

Input: *sim*: simulation object
Input: *nSteps*: total number of time steps to advance
Input: *dt*: amount of time to advance simulation
Output: New state of the system after *nSteps*.

```
1: function TIMESTEP(sim, nSteps, dt)
2:     for i ← 1 to nSteps do
3:         AdvanceVelocity(sim, 0.5*dt)
4:         AdvancePosition(sim, dt)
5:         RedistributeAtoms(sim)
6:         ComputeForce(sim)
7:         AdvanceVelocity(sim, 0.5*dt)
8:     end for
9:     KineticEnergy(sim)
10: end function
```

5.1 Reference Implementations

CoMD was originally implemented in the C language and uses the OpenMP programming model, to exploit the intra-node parallelism, and MPI [72], to distribute work among nodes [21]. Cicotti et al. [19] have investigated the effect of exploiting a multithreading library (e.g., *pthreads*) in contrast to using the OpenMP and MPI approach. In addition to the OpenMP and MPI implementations, a CUDA-based implementation was also developed in the C++ language [71]. These reference versions include all of the three main kernels; force computation, advancing velocity, and advancing position of atoms. Developers used CUDA to be able to fully exploit the capacity of the GPUs. As a result the data layout of the application was significantly changed in order to tap into the rich capacity of the GPUs. Naturally, this puts a lot of burden on the developers and the code cannot be used on any other platforms other than NVIDIA GPUs. Both OpenMP and CUDA implementations were optimized to utilize the full capacity of the underlying hardware. Here, we focus on the optimizations that are beneficial to the OpenACC implementation.

5.2 Parallelizing CoMD with OpenACC

This section is dedicated to the discussion of porting CoMD to a heterogeneous system using the OpenACC programming model. We started with the OpenMP code version for this porting process instead of the serial code. This may not be the best approach because in most cases the OpenMP codes are well-tuned and optimized for shared memory platform but not for heterogeneous systems, especially the codes that have used OpenMP 3.x.

As the first step, we profiled the code and discovered that the force computation (*line 6* in Algorithm 3) was the most time consuming portion of the code. Consequently, it suggests porting the force computations to the device. This requires the transfer of both the computational kernel and its data (the data that the kernel is working on) to the device. However, if we only accelerate the force computation kernel, we need to transfer data back and forth to and from the device for each time step, which will lead to dramatic performance degradation. That is, it imposes two data transfers (between host and device) for each time step. As a result, this pushes us to parallelize other steps (*line 3, 4, and 7*) too. Hence, data transfers can be performed before (*line 2*) and after (*line 8*) the main loop.

The RedistributeAtoms step (*line 5*) guarantees data consistency among different MPI [72] *ranks*. Since MPI functions are only allowed to be called within the host, the data have to be transferred back to the host for synchronization purposes among the ranks. After performing synchronization, the updated data are transferred from the host to the device. The synchronization process is done on every time step to maintain data consistency. Consequently, two data transfers are performed in this step between the host and the device, and, since no remarkable computations were performed in this step, no parallelization was required for this step.

Based on our analysis, the parallelization of the three above-mentioned kernels (`ComputeForce`, `AdvancePosition`, and `AdvanceVelocity`) contributes the most towards the performance of our application because they are the most time-consuming computational kernels. Although the latter two kernels may seem insignificant due to their smaller execution time, they will progressively affect the wall clock time of the application in the long run. Thus, the focus of our study is on applying performance optimization of these three kernels. Our measurements reveal that our OpenACC implementation was able to reach the same occupancy level as that of the CUDA implementation. Force computation, however, is more complex and requires more attention with respect to its optimization opportunities. However, we can safely use OpenACC version of the `ComputeForce` and `AdvancePosition` kernels with their CUDA counterparts with no performance loss.

There are four options to parallelize CoMD: 1) UVM, 2) deep copy, 3) significant code changes to transfer data structures manually, and 4) `pointerchain`. *Step 1* in our proposed steps represents the UVM approach and as elaborated in Section 3.1, it has several disadvantages. Deep copy is a feature that we cannot use right away as it is not yet fully implemented in any compiler. The third option requires significant code changes performed manually; as a result this is not a favorable approach for developers to adopt and it contradicts the philosophy of OpenACC. That brings us to our fourth and the last option, i.e., `pointerchain`. Annotating CoMD's source codes with `pointerchain` directive helps us to easily port CoMD to OpenACC and it also helps us apply the different optimizations listed in the Table 5.1. Please refer to the Supplementary Material of our Parallel Computing paper [37] for a detailed description of each step.

Table 5.1 provides a brief description of the ten steps taken in this paper to parallelize CoMD. Figure 5.2 shows the order in which we took the steps. These steps also provide a roadmap for parallelization of any other scientific applications using OpenACC. The *pointerchain* column shows whether our proposed novel directive has been used for a step or not.

Without the `pointerchain` directive, the source code needs to undergo numerous modifications. Such modifications are error-prone and cumbersome for developers.

Table 5.1: Overview of all steps that were applied to CoMD. The *pc* column designates whether `pointerchain` was applied at that step or not.

S.	Title	*pc*	Description
1	Kernel parallelization	×	Relying on the UVM for data transfer. Annotating the potential kernels with `#pragma acc kernels` for parallelization.
2	Efficient data transfer	✓	Disabling UVM and specifying manual data transfer between host and device. We started using `pointerchain` from this step forward. `#pragma acc kernels` for parallelization.
3	Manual parallelization	✓	Utilizing `#pragma acc parallel` on kernels instead of `#pragma acc kernels`. Designating `gang` and `vector` levels on multi-level loops.
4	Loop collapsing	✓	Collapsing tightly nested loops into one and generating one bigger, flat loop.
5a	Improving data locality (dummy field)	✓	Adding a dummy field to make data layout cache-friendly.
5b	Improving data locality (data reuse)	✓	Improving the locality of the innermost loops by employing local variables in the outermost loops.
5c	Improving data locality (layout modif.)	✓	Modifying layout as described in detail in the Supplementary Material of our Parallel Computing paper [37].
6	Pinned memory effect	✓	Enabling *pinned memory allocations* instead of regular pageable allocations.
7	Parameters of parallelism	✓	Setting `gang` and `vector` parameters for parallel regions.
8	Controlling resources at compilation time	✓	Manually setting an upper limit on the number of registers assigned to a `vector` at compilation time.
9	Unrolling fixed sized loops	✓	Unrolling one of the time consuming loops with fixed iteration count.
10	Rearranging computations	✓	Applying some code modifications to eliminate unnecessary computations.

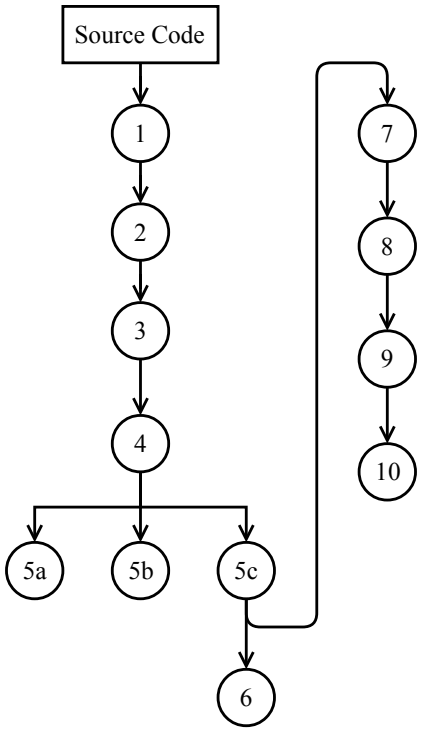


Figure 5.2: The relationship among the optimization steps that were taken to parallelize CoMD. For detailed description of each step, please refer to Table 5.1.

5.3 Porting CoMD: Performance Implications

We have ported CoMD to heterogeneous systems using OpenACC and applied the optimization steps, as mentioned in Table 5.1. We discuss the influence of each step on the final outcome.

5.3.1 Measurement Methodology

We relied on NVIDIA's `nvprof` profiler for device measurement. It provides us with a minimum, a maximum, and an average execution time, driver/runtime API calls, and a memory operation for each GPU kernel. It is a handy tool for those who tune an application to achieve maximum performance of GPUs. All simulations were executed in single precision.

5.3.2 Model Preparation

To extract optimal values for the `gang` and `vector` parameters, we traversed through a parameter search space for them. We also investigated the effect of manually choosing the number of registers at compile time on the performance. Then, we used the extracted optimal values for `gang`, `vector`, and register count parameters. Please refer to the Supplementary Material of our Parallel Computing paper [37] for detailed discussion on characterizing the above-mentioned parameters.

5.4 Results

5.4.1 Speedup for each Parallelization Step

To observe the accumulated effect on the final result, our modifications in each step were implemented on top of the preceding steps unless noted. Please refer to Figure 5.2 for the causal effect between each consecutive step.

Figure 5.3 illustrates the impact of each step on our program by showing the changes in the execution time of the three kernels. We included the results from the CUDA and OpenMP versions. The OpenMP version was compiled with both Intel[1] and PGI[2] compilers, shown as OMP-ICC and OMP-PGI, respectively. Besides targeting OpenACC for NVIDIA GPUs, we also retargeted our OpenACC code for *multicore* systems (ACC-MC in the figures). We did not modify a single line of code when retargeting our code to multicore systems with OpenACC. We only changed the target device from NVIDIA Tesla to multicore at compilation time. Results are shown for both small (bottom) and large (top) data sizes and they are normalized with respect to CUDA.

[1]Intel Compiler flags: `-Ofast -O3 -xHost -qopenmp`
[2]PGI Compiler flags: `-mp -fast -O3 -Mipa=fast`

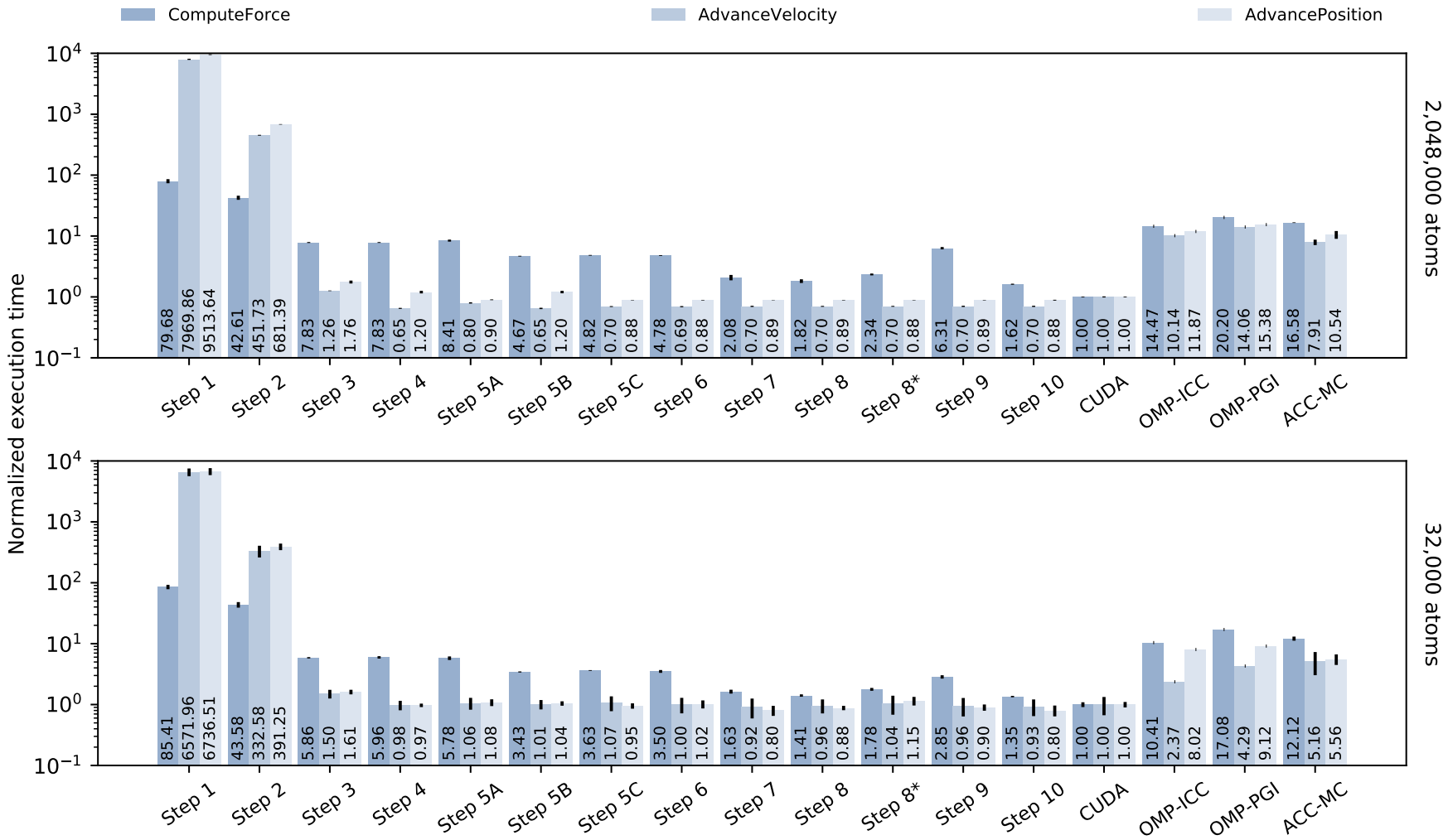


Figure 5.3: Normalized execution time after applying all optimization steps and run on NVIDIA P100. After applying all 10 steps on the OpenACC code, we were able to reach 61%, 129%, and 112% of performance of the CUDA kernels for `ComputeForce`, `AdvancePosition`, and `AdvanceVelocity`, respectively. Results are normalized with respect to *CUDA*. OMP-ICC and OMP-PGI refer to the OpenMP version of the code compiled with the Intel and PGI compilers, respectively. ACC-MC refers to the OpenACC version of the code for the multicore architecture (compiled with the PGI compiler).

Enabling UVM on the memory-intensive kernels impedes performance in the first few steps. The reduction in execution time is several orders of magnitude while proceeding from *Step 1 to 2*. The same trend was observed from *Step 2 to 3* for all three kernels. Due to developers' insight on data layout and parallelism opportunities, the impact of the proposed changes in these steps is significant in comparison to the compiler's insights.

The next significant reduction in execution time happens when data-locality improves by reusing variables (from *Step 5A* to *Step 5B* and *Step 5C*). Such an improvement is due to the reduction in the access of the physical memory by caching it with local variables. In order to compute exerted force on Atom A, we looped through all atoms in the vicinity and computed the force between them. Therefore, instead of redundantly loading Atom A from memory for each loop iteration, we have loaded it once before the inner loop and reused it within the loop as many times as possible.

Step 7 marks the next substantial reduction in the execution time for our compute-intensive kernel. In *Step 7*, we set the `gang` and `vector` parameters to their optimal values from Section 5.3.2 and collect measurements for each kernel. Manually setting these parameters enables the scheduler to issue extra gangs on the device and keep the resources busy at all time (in comparison to the choices by the compiler).

Inefficient utilization of resources leads to performance loss. We see a 16% performance gain from *Step 7 to 8*, which is due to the optimal usage of register per kernel. Increasing the number of utilized registers for all kernels is not beneficial to the performance. Kernels with different traits require different considerations. Our experiments reveal that the memory-intensive kernels do not benefit from a large number of registers. Hence, it is better to limit the register count for such kernels. On the other hand, the compute-intensive kernels highly benefit from a large number of registers since they minimize the access of global memory for temporary variables.

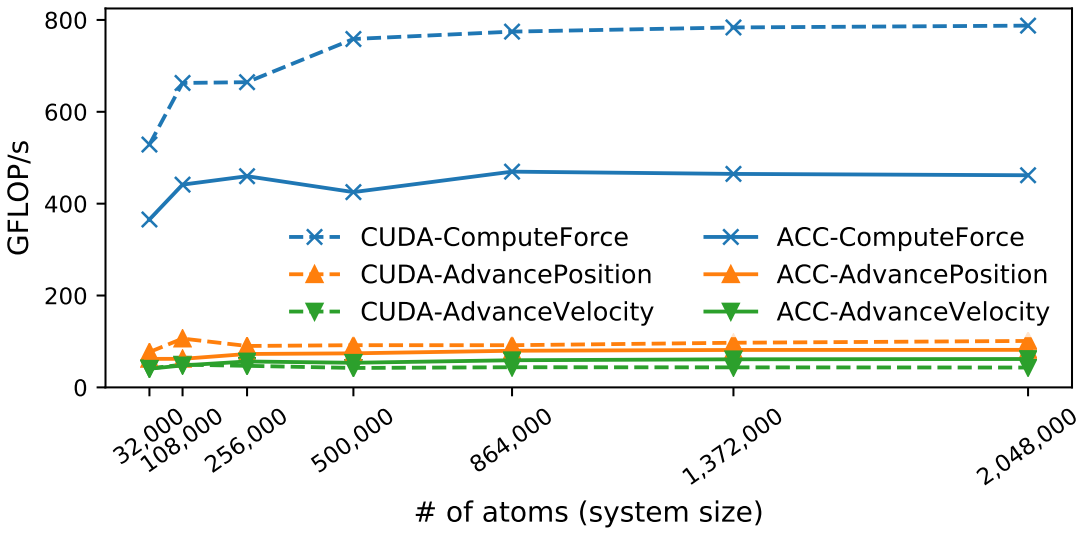


Figure 5.4: Giga floating-point operations per second (GFLOP/s). In case of the `ComputeForce` kernel, despite comparable speedups with respect to CUDA, the number of floating-points operations that OpenACC implementation executes is behind CUDA's performance. The OpenACC implementation of `AdvanceVelocity` performs better than its CUDA's counterpart. Measurements are performed on P100 of Nvidia's PSG cluster.

Elimination of redundant *reduction* operations, as described in *Step 10*, boosted the performance and helped our implementation to reach performance of that of CUDA's. Rearrangement of computations and elimination of unnecessary redundant operations have definitely led to performance gain.

We have discussed ten optimization steps that for our proxy application, CoMD, boosted the *ComputeForce* kernel's performance by 61-74% in comparison to its counterpart written in CUDA. Although OpenACC did not reach CUDA's efficiency, it got close to its performance with a very small code modification footprint. Additionally, our OpenACC code is portable to another architecture without needing to change any portion of the code. In contrast, a CUDA-based application needs to be updated or revisited every time when the architecture is upgraded, thus affecting the maintenance of the code base. The memory-intensive kernels are performing better than their counterparts written in CUDA as noted for *Step 7* for both small and large data sizes. It is probably due to scheduler-friendly instruction generation by the PGI compiler.

5.4.2 Floating-point Operations per Seconds

We measured the floating-point operations per second (FLOPS) of our under-study kernels and compared them with the CUDA implementation for one GPU. There is an increasing gap between the implementations of *ComputeForce* kernel and a decreasing gap for memory-intensive kernels in Figure 5.4. For the latter ones, the difference is negligible and in case of *AdvanceVelocity*, the OpenACC version is performing better than CUDA. However, the case for *ComputeForce* kernel is different. As it becomes complicated for the OpenACC compiler to apply necessary optimization techniques on that kernel, the performance gap between the OpenACC and CUDA implementations increases. When developers take advantage of the interoperability feature of OpenACC to run CUDA kernels within OpenACC code, they are allowed to manually tune the bottleneck kernels that do not necessarily benefit from the compiler-generated code. However, it will adversely affect the portability of OpenACC codes.

Figure 5.4 shows how the OpenACC version maintains the computation sustainability of the floating-point operations, as the number of atoms increases. Similar to the CUDA implementation, the OpenACC implementation does not lose performance as system size increases exponentially.

5.4.3 Scalability with Data Size

We investigated the scalability of our OpenACC implementation with respect to varying system sizes. We varied the system size from 32,000 to 2,048,000 atoms and measured the per-atom execution time for five implementations; OpenACC-GPU (*acc-GPU*), OpenACC-Multicore (*acc-MC*), CUDA, Open MP-ICC (*OMP-icc*), OpenMP-PGI (*OMP-pgi*). The results are depicted in Figure 5.5 for NVIDIA's PSG cluster. Interestingly, our OpenACC implementation scales with the system size without any performance loss. As discussed in

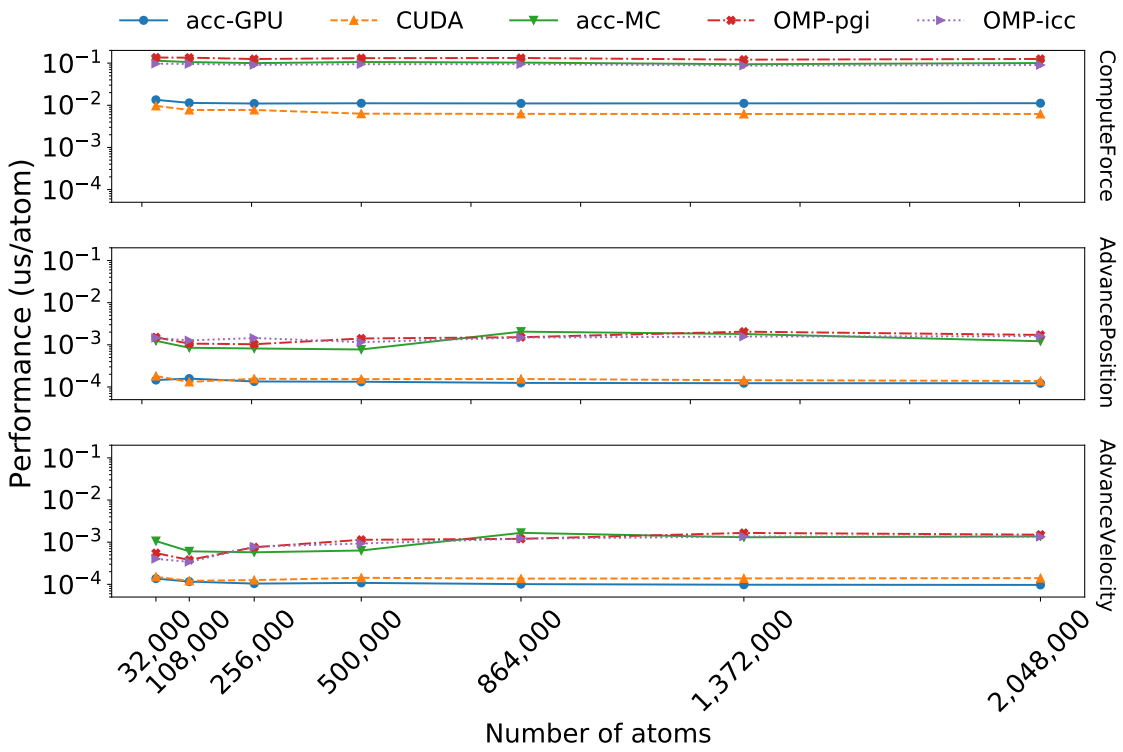


Figure 5.5: Scalability with different data sizes with *one* GPU of NVIDIA P100. One can observe that performance is not lost when data size is increased. OpenACC-Multicore performs better in comparison to OpenMP counterparts. The lower the value, the better the performance results. Measurements are performed on Nvidia's P100 from PSG.

the last section, we experienced better performance with OpenACC than using CUDA for memory-intensive kernels.

Another interesting observation is that there is no significant gap between OpenACC-Multicore and its OpenMP counterparts. In some cases, OpenACC performs better than the Intel optimized OpenMP version for Haswell processors on the PSG platform. In comparison to the generated code for OpenMP by the PGI compiler (OMP-PGI), OpenACC code performs better in the case of the `ComputeForce` kernel.

5.4.4 Scalability Measured at Different Architectures

Scalability plays an important role in utilizing upcoming architectures. We have indicated such scalability with BigRed's K20, UHPC's K80, and PSG's P100 and V100 in Figure

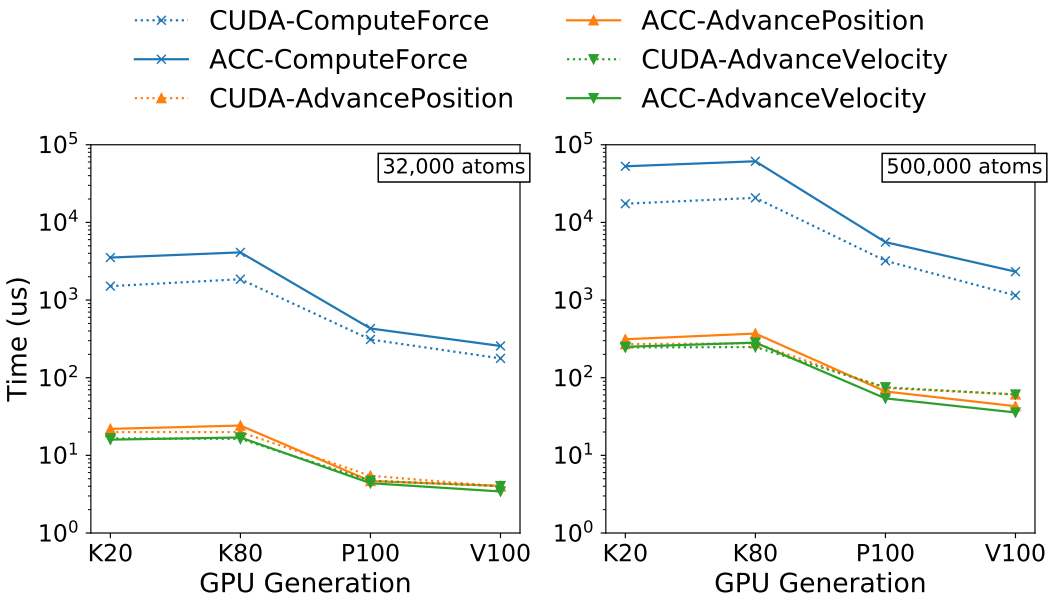


Figure 5.6: Scalability with different architectures while exploiting *one* GPU in the target architecture. With new architectures, performance is improving by shortening time. Lower is better.

5.6. The gap between CUDA and OpenACC implementations narrows when the underlying architecture evolves, particularly for the `ComputeForce` kernel. Results in this section are based on utilization of one GPU in each device.

5.4.5 Scalability with Multiple GPUs

We have investigated the scalability of our OpenACC implementation for more than one GPU. NVIDIA's Pascal P100 has 4 GPUs inside the PCI card. For each GPU, an MPI process is initiated and that process takes control of a single GPU. All processes communicate through the MPI library to distribute workload among themselves. Results, depicted in Figure 5.7, show speedups with respect to 100 timesteps of CoMD with different system sizes. The *ComputeForce* kernel shows promising results for both system sizes. The OpenACC implementation scales better in comparison to its CUDA implementation. However, the other two memory-intensive kernels do not benefit from multi-GPU scalability of OpenACC code. It is due to the fact that they spend most of their time waiting for memory. Consequently,

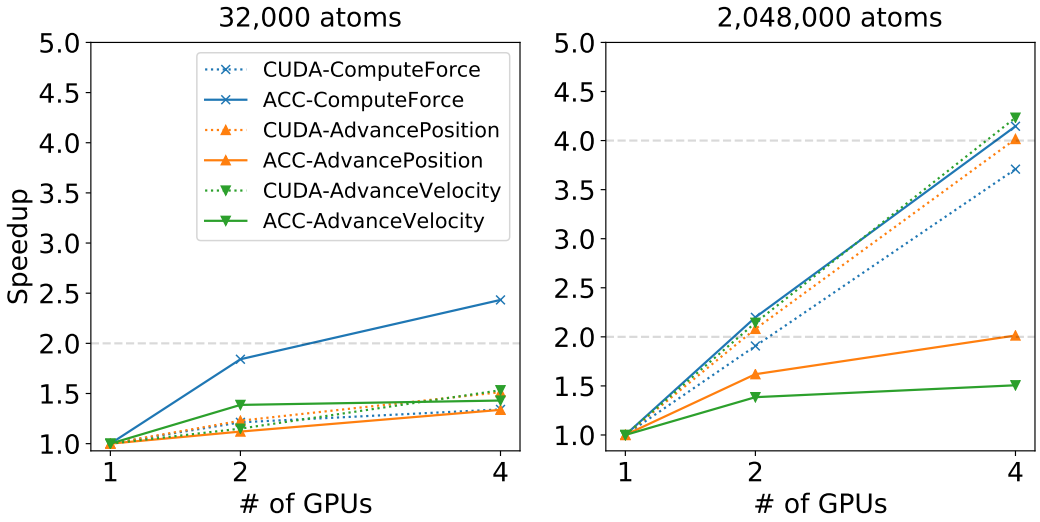


Figure 5.7: Scalability of implementations on NVIDIA P100. The `ComputeForce` kernel is performing linearly and its performance is close to its CUDA counterpart.

they do not benefit from the extra computational resources in comparison to our compute-intensive kernel. Such a conclusion, however, is not true for their CUDA counterparts and they show linear speedup for 2,0480,000 atoms.

Figure 5.8 displays results for V100. Similar to its predecessor, Pascal P100, Volta V100 also possesses 4 GPUs inside the PCI card. All the algorithms show linear (or super-linear) scalability when our system size is large. The scalability of our implementation is comparable to the CUDA's, and in the case of the `ComputeForce` kernel, OpenACC performs better. When our system size is not large enough, OpenACC's scalability of the `ComputeForce` kernel is 59% and 70% better for 2 and 4 GPUs, respectively. In the case of the other two kernels, CUDA and OpenACC's scalability are similar.

Figure 5.7 shows the super-linear scalability for the three kernels with 2,048,000 atoms. The OpenACC's *ComputeForce* kernel is super-linear due to the utilization of a cut-off range within the algorithm. Skipping some iterations of a loop helps the kernels to reach super-linearity. On the other hand, efficient cache utilization of CUDA's *AdvancePosition* and *AdvanceVelocity* kernels has led to a super-linear speedup. Figure 5.8 depicts similar results on the V100 architecture. Due to improvements in cache performance of the V100

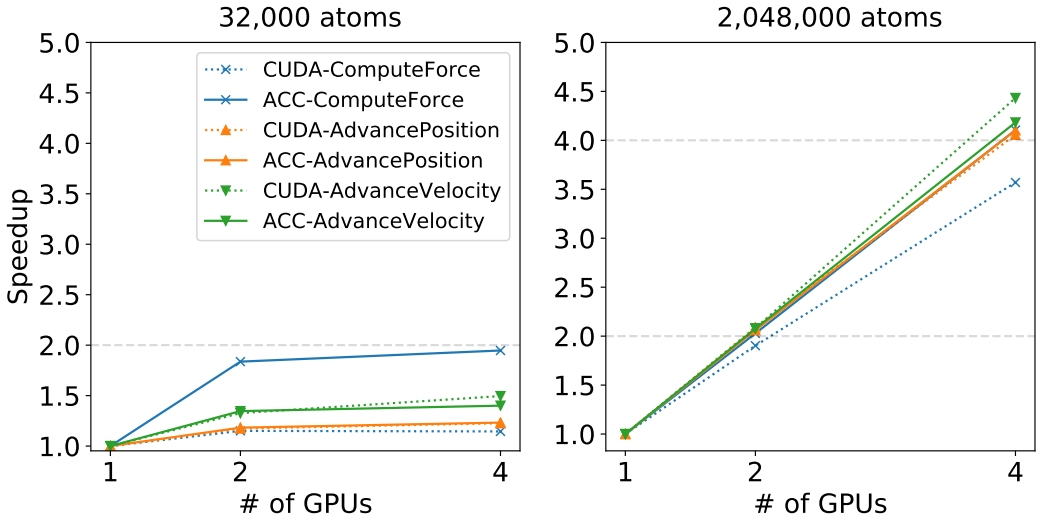


Figure 5.8: Scalability of implementations on NVIDIA V100. For a 2,048,000-atom system, OpenACC and CUDA scale linearly with the number of GPUs. In case of `ComputeForce`, OpenACC shows more scalable performance in comparison to CUDA. The CUDA implementation of the `AdvanceVelocity` kernel displays a super-linear performance.

architecture in comparison to P100[3], the two CUDA kernels that were underutilized on P100 show linear performance. Figures 5.7 and 5.8 show how CoMD shows sub-linear speedups for 32,000 atoms for all three kernels. It is due to the high overhead of workload distribution. When our system size is small, CoMD does not benefit from the multi-device distribution. However, as we increase the system size, we notice an explicit improvement in the speedup of the kernels.

5.4.6 Effects on the Source Code

OpenACC does not impose a significant impact on the source code size and maintenance; thus, it retains the integrity of a complex scientific application. Similar to OpenMP, developers are not required to write excessive lines of code to maintain the state of the application and accelerators. As a result, we exploited *lines of code (LOC)* to quantitatively measure the code complexity. The measurement was performed with the *cloc* [20] tool. Table 5.2

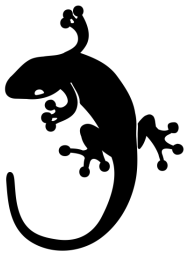
[3]The L2 cache size has increased from 4MB in P100 to 6MB in V100.

Table 5.2: Effect of the OpenACC adaption on the source code – lines of code (LOC) column shows extra line required to implement this step with respect to the OpenMP implementation as the base version. The third column (%) shows the increase with respect to the base version.

Step	LOC	%	Step	LOC	%
OpenMP	3025	-	Step 6	+163	5.39
Step 1	+2	0.07	Step 7	+198	6.55
Step 2	+99	3.27	Step 8	+198	6.55
Step 3	+103	3.4	Step 9	+187	6.18
Step 4	+109	3.6	Step 10	+215	7.11
Step 5A	+109	3.6	CUDA	+4745	**1.57X**
Step 5B	+125	4.13			
Step 5C	+165	5.45			

presents the results for the LOC for each step. We used reference implementation of CoMD (the OpenMP version) as the starting point for our porting process to OpenACC. The LOC column shows that the total extra lines of code required to implement that step with respect to OpenMP implementation as the base version. The third column (%) shows the percentage with respect to the base version. The CUDA implementation doubles the amount of code size in comparison to the OpenMP version. However, for OpenACC, LOC is only less than 8%. These include the LOC from *Step 2 to 10* with extra `pointerchain` lines. In some transitions from one step to the other (e.g., *Step 7 to 8*), there is no difference in LOC. That is, we only changed the compilation flags, which naturally does not count towards the LOC count.

Chapter 6



Gecko: A Hierarchical Memory Model

Previously published content:

1. Ghane, Millad, and Chandrasekaran, Sunita, and Cheung, Margaret S., "Gecko: Hierarchical Distributed View of Heterogeneous Shared Memory Architectures." In Proceedings of the *10th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'19) in conjunction with PPoPP*, 2019. [36]

This chapter will discuss *Gecko*[1], a hierarchical memory model for current and future generations of computing systems. Gecko is a model that represent current diverse memory models in a heterogeneous system. ***Based on this model, we also provide a similar programming model with the similar name as a proof of concept to show the feasibility of our hierarchical model.*** A preliminary investigation of Gecko was discussed in Ghane et al. [38] as well. For a detailed discussion of all available clauses in Gecko's programming language, please refer to Appendix A at the end of this dissertation.

[1]Courtesy of Noun Project: `https://thenounproject.com/` for the gecko icon at the top of the page. The reason behind choosing gecko as the name of our framework is the resemblance of its hands to the hierarchy that our model proposes. Its fingers resemble the leaf nodes in our proposed model.

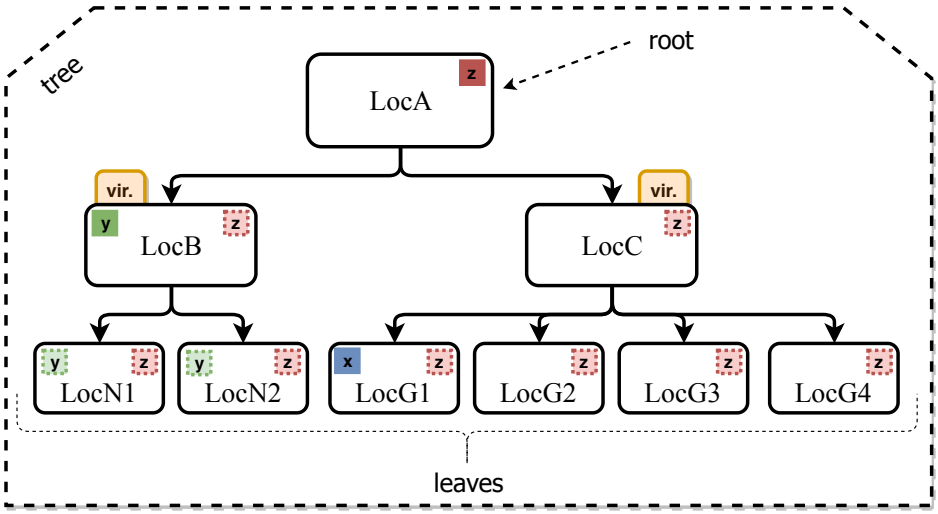


Figure 6.1: An overview of a hierarchical shared memory system. Locations are specified with *Loc* prefix. Small boxes represent variables in the system. The solid lines show the location where variables are defined. The dotted lines represent the locations in hierarchy that have access to that variable. Virtual locations are designated with *"vir." tags*. *Tree* is a hierarchical representation of relationship among locations. The *root* location is the topmost location that has no parent. *Leaves* are locations at the bottom of the tree that have no children.

6.1 The Gecko Model

Locations are the principal constructs in a Gecko model. Locations are an abstraction of available memory and computational resources in the system and are represented as a node in *Gecko*'s tree-based hierarchy model[2] [55]. Similar to the Parallel Memory Hierarchy (PMH) model [5], *Gecko* is a tree of memory modules with *workers* as the leaves of the tree. Workers provide computational capabilities to execute the kernels; they are attached to the memory modules. Figure 6.1 illustrates an example of the *Gecko* model. This model represents a regular cluster/supercomputer node with two non-uniform memory access (NUMA) multi-core processors [56], *LocNi*, and four GPUs, *LocGi*, similar to NVIDIA's PSG cluster [80] and ORNL's Titan supercomputer [85].

[2]In computer science, a tree structure is the way of representing a hierarchy in the form of nodes and relationships. Nodes relate to each other through the child-parent relationship. Each node has at most one parent and zero to many children. The *root* node of the tree is the topmost node in a tree; it is the only node with no parent. *Leaves* are nodes at the bottom of the tree that have no children. In Gecko's terminology, nodes are called locations. Please refer to Figure 6.1 for an example of a tree, a root, and leaves.

The location hierarchy in *Gecko* designates how one location shares its allocated memories with another. When 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 has. However, the allocated memory is not shared with their parent and is considered to be a private memory with respect to their parent. Figure 6.1 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. By the same logic, allocated memory z can be accessed by all locations, while accesses to x is limited to *LocG1*.

In *Gecko*, locations are categorized as follows: 1) a memory module, 2) a memory module with worker, and 3) a *virtual* location. Memory modules are annotated with their attributes (e.g., type and size); *LocA* in Figure 6.1 is a memory module. If a location is a memory module with a worker attached to it, the location will be used to launch a computational kernel by the runtime library; *LocNi* and *LocGi* are examples of memory modules with workers. Finally, the virtual locations, *LocB* and *LocC* in Figure 6.1, 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 such locations as a stand-alone location. Similar approaches have been observed in related work [112, 111]. Virtual locations can also be the target of memory allocation and kernel launch requests as well. Depending on the type of requests and hints from the developers, the runtime library will act upon the requests and perform 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 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) [112], *Gecko* provides facilities to represent any physical machine with different abstractions [112, 111]. The best configuration depends on the characteristics of the application, its locality requirements, and the its workload balance. The developer or auto-tuning libraries can assist *Gecko* in choosing the effective abstraction. Figure 6.2 shows how *Gecko* represent nodes in Summit [84] and Tianhe-2 [61]. 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.

A 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 [31] 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, `Intel MIC`. In cases where an application faces a diverse set of accelerator types in a cluster or supercomputer (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* can adapt to the current accelerator in the system without any code alterations. *Gecko* also is able to adapt to changes in the workload and

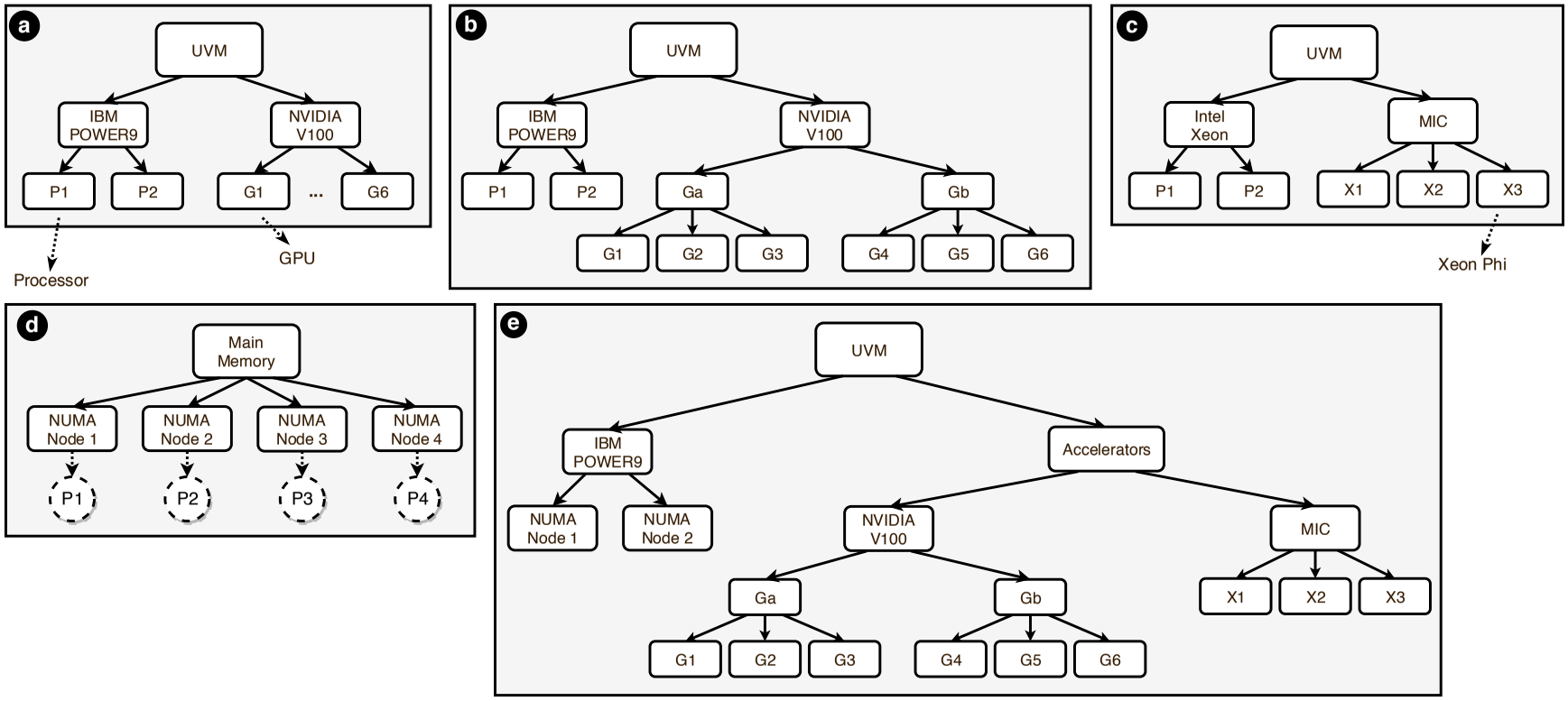


Figure 6.2: *Gecko*'s model representing various system. ORNL's Summit (a and b) with two IBM POWER9 processors and six NVIDIA Volta V100 GPUs – Tianhe-2 (c) with two Intel Xeon processors and three Intel Xeon Phi co-processors – NUMA architecture with four NUMA nodes (d) – A complex platform with multiple types of accelerators (e)

employ more resources, if needed to expedite the processing, by modifying the hierarchy.

Gecko also supports NUMA-only architectures in symmetric multiprocessor (SMP) systems. Model ⓓ depicts a NUMA system with four NUMA nodes and their corresponding processors. Processors (with multiple cores) are drawn to show how leaf nodes contain compute resources. They are not a part of the model. The locality that NUMA architecture provides in configuration in ⓓ can be exploited using *Gecko*, which is not possible with the flat models of OpenMP and OpenACC. Finally, Model ⓔ shows a sophisticated design that includes all the models in ⓑ, ⓒ, and ⓓ. With this design, applications target `Accelerators` location to utilize all available GPUs and MICs in a system.

Gecko is implemented on top of the OpenMP [83] and OpenACC [82] programming models to generate code for the computational kernels. Using these programming models helps us to achieve portability with one single source code for the computational kernels in our applications. To use Gecko, developers will write their kernels once and then Gecko will use those kernels to target different architectures. This is only achievable only through the above-mentioned programming models. Section 6.5 discusses how we implemented Gecko and put to use OpenMP and OpenACC.

6.2 Key Features of *Gecko*

This section is dedicated to key features that *Gecko* provides, which makes it superior compared to the current flat models.

6.2.1 Minimum Code Changes

Gecko's hierarchical tree leads to minimum source code modification. 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 6.3a is another representation of the model in Figure 6.1; the same configuration with an extra virtual location: *LocV*. The dotted lines represent the 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 are able to launch their parallel regions in the code on this location while they basically know nothing about the hierarchy structure beneath *LocV*. At execution time, an application is able to change the potential subtrees deliberately. By enabling the left (case **b**) or right (case **c**) relationship in the tree, 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.

6.2.2 Dynamic Hierarchy Tree

Unlike Sequoia [28] and HPT [112] that build their hierarchy at the compile time, Gecko *dynamically* constructs its hierarchy at execution 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. This enables an application to react to the changes in the workload size by adding and removing resources accordingly, as needed. This feature also enables applications to adapt themselves to the workload type. For instance, for applications that benefit from multicore architectures, like traversing a linked list in a graph, Gecko helps the applications to use multicore processors instead of accelerators.

Transforming from the case **a** to the cases **b** and **c** in Figure 6.3 shows the polymorphic capabilities of Gecko. Similarly, Gecko enables the vertical expansion of the hierarchy. Figure 6.3d shows the equivalent model of the model represented in Figure 6.3a. The only

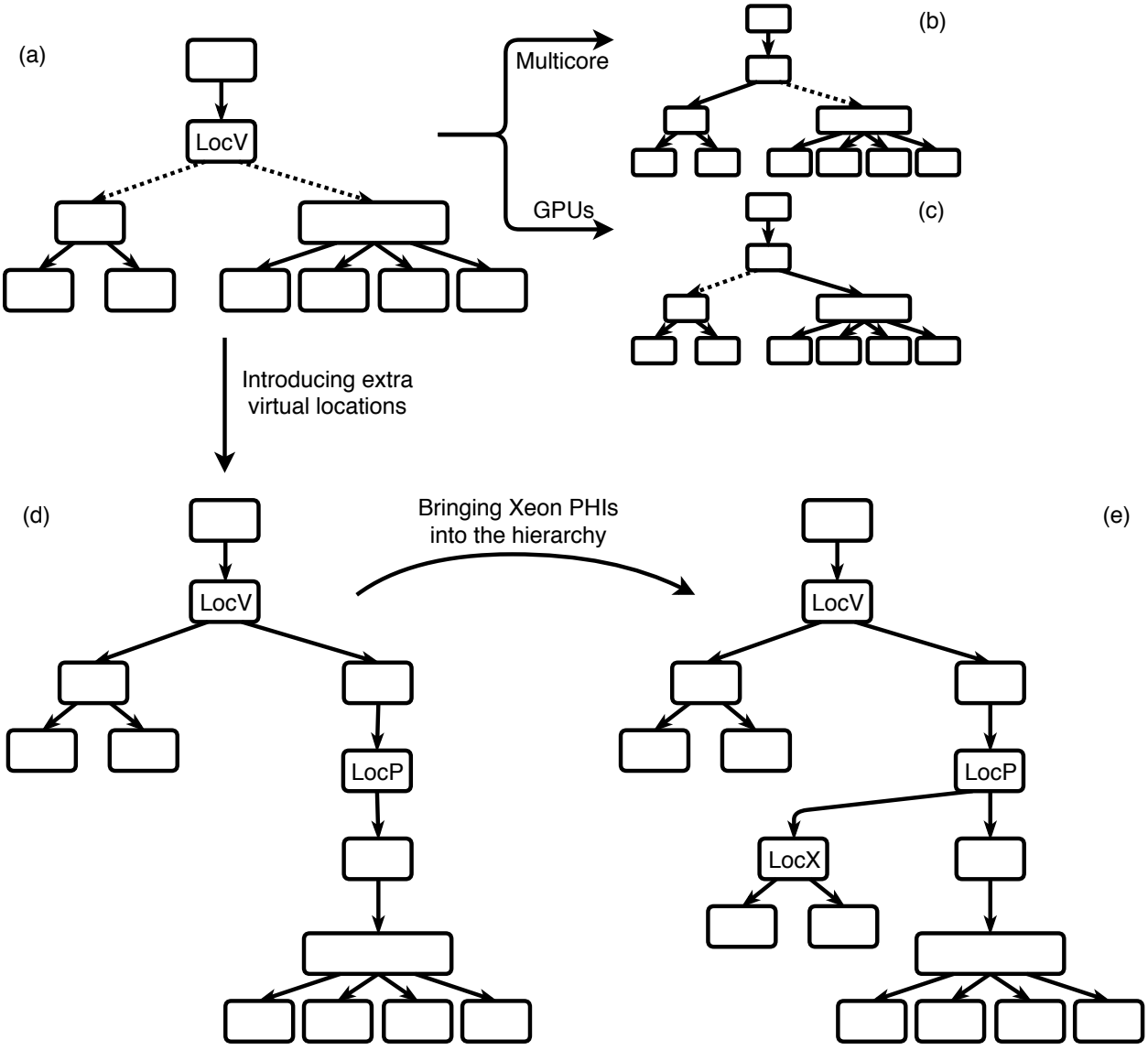


Figure 6.3: Polymorphic capabilities of *Gecko* lead to fewer source code modifications. We can change the location hierarchy at run time. Our computational target can be chosen at runtime: processors (b) or GPUs (c). *Gecko* also supports deep hierarchies in order to provide more flexibility to applications (d). We are able to extend the hierarchy as workload size changes (e).

difference is the extra virtual locations in the hierarchy. 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 ways since virtual locations do not have any physical manifestations. The workload submitted to *LocV* is simply passed down to its children and distributed according to the chosen execution policy (will be discussed in Section 6.5). Virtual locations are also effective in the adaptation of an application to changes in the environment or the 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 the workload. The application defines a virtual location, *LocX*, and declares the new Xeon Phis as its children. Then, by declaring *LocX* as a child of *LocP* in Figure 6.3d, *Gecko* is able to utilize the Xeon Phis to distribute the workload. 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 6.3e shows the new model with two Xeon Phis included in the hierarchy. Later, one can detach the *LocX* location from the tree and return to the model shown in Figure 6.3d.

Gecko offers a *location coverage* feature that helps extend the adaptation capabilities of virtual locations. Location coverage creates a virtual location to represent all resources with the same location type. In many cases, the number of locations of a specific type is unknown till 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 the new environments.

6.3 Challenges Raised by the Key Features

Gecko addresses such flexibility with its novel hierarchical approach to represent available resources. However, this representation raises many challenges that need to be addressed such as: 1) Given a set of variables scattered in various locations on the hierarchy tree, which location is responsible to execute a kernel? 2) How to distribute execution among children of a location? 3) Considering the dynamism that we introduce, how is memory allocated since the targeted location is only known at the execution time? We discuss how Gecko addresses some of these challenges.

6.3.1 Finding Location to Execute Code

Data placement is not a trivial job. Agarwal et al. [1] and Arunkumar et al. [8] emphasize how either an "expert programmer" needs to do it or extensive profiling is required to find the efficient placement. Each approach has its own advantages and drawbacks. For this work, Gecko relies on the expertise of the programmer to place the data in their proper location. The programmer, using Gecko's directives, allocates a block of memory in any location. This allows Gecko to have up-to-date knowledge about where each allocated memory is placed.

As an application progresses over time, it allocates memory in different locations. The location that is chosen depends on various criteria (e.g., requesting bandwidth- or capacity-optimized memory). This, in turn, causes the allocated memory to be scattered around different locations within the hierarchy. Therefore, if a computational kernel utilizes multiple memory allocations that are not in the same location, a location that is the most suitable has to be chosen, which has to be accessible by all the allocated memory throughout the hierarchy. We follow the Most Common Descendent (MCD) algorithm (described in Section 6.3.1.2) to execute this strategy.

6.3.1.1 Data Locality

Moving data around is quite a costly operation, which can be addressed by adopting better data locality optimization strategies. As a result, Gecko migrates the computations to the targeted location instead of moving data. Therefore, when a kernel is initiated on previously-allocated memory, Gecko locates them in the hierarchy, and then it finds the proper location to execute the computational kernel. Choosing the proper location depends on where the memory is scattered within the hierarchy. Section 6.3.1.2 proposes an efficient algorithm that finds the final location in the hierarchy to execute the computational kernel.

6.3.1.2 Most Common Descendent (MCD) Algorithm

Given a hierarchy tree and a set of locations within the tree, the Most Common Descendent (MCD) algorithm finds a location in the set that is the child and/or grandchild of all other locations in the input set. Such a location is also the deepest location in the hierarchy among the locations in the input set. Algorithm 4 shows this algorithm in detail.

The MCD algorithm is the exact opposite of the Lowest Common Ancestor (LCA) algorithm [3]. While LCA traverses upwards in the tree to find the most common parent among a given set of locations, MCD traverses the tree downwards in order to find the most common child. Unlike LCA, MCD may not have a final answer for a given input set.

Algorithm overview: Figure 6.4 demonstrates three scenarios that will happen when we run the MCD algorithm as shown in Algorithm 4. First, we set the first location as *the final answer* and record its path to the root as *the final path* (shown as **a**). Then, we loop over other locations. For each location, there are three possible scenarios, which are shown in **(b)**, **(c)**, and **(d)** of Figure 6.4. If a location is found on the final path (as shown in **b**), we already have the final answer, so we will skip this location. If a location has a path to the root that does not overlap with the current final path (as shown in **c**), there is no final

Algorithm 4 Most Common Descendent (MCD) Algorithm

Input: T: *Gecko*'s hierarchical tree structure.
Input: L: List of locations.
Input: $pathToRoot(t, n)$: returns the locations on the path from node n to root of the Tree t
Output: The most common descendent or NULL

```
1: function MCDALGORITHM(T, L)
2:     commonChild ← L_0
3:     commonPath ← pathToRoot(T, L_0)
4:     for each L_i ∈ L do
5:         if L_i ∉ commonPath then
6:             newPath ← pathToRoot(T, L_i)
7:             if commonChild ∉ newPath then
8:                 return NULL
9:             end if
10:            commonChild ← L_i
11:            commonPath ← commonPath
12:        end if
13:    end for
14:    return commonChild
15: end function
```

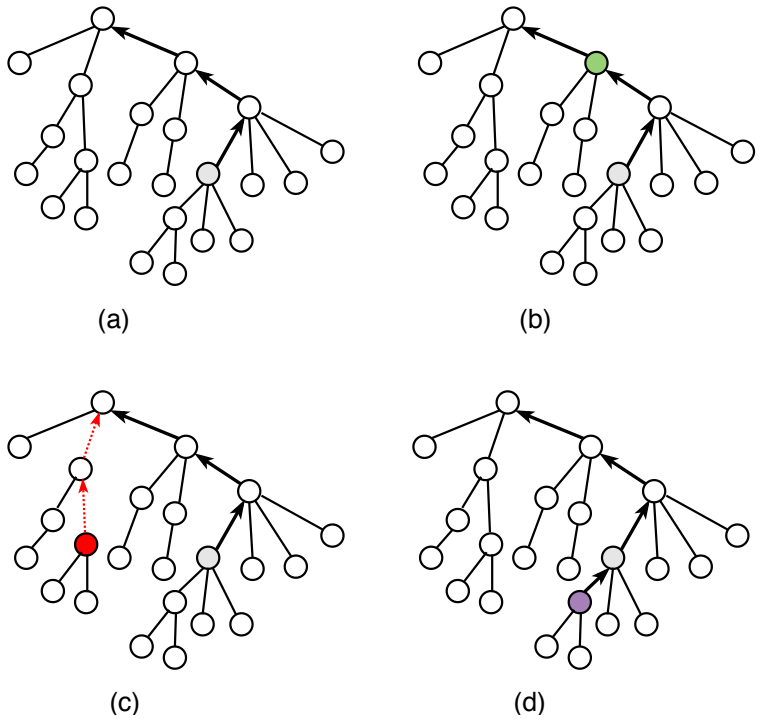


Figure 6.4: Four different possible scenarios in the MCD algorithm.

answer that satisfies the provided input, and consequently, the program halts. And finally, if a location has a path to the root that includes the current final answer (as shown in **d**), the final answer and the final path are changed to this new location and path. To put it briefly, the MCD algorithm tries to find a common path among all input locations. If one is found, the deepest location in the hierarchy is returned. Otherwise, there is no result with respect to the provided input.

Complexity: The complexity of the MCD algorithm in the worst case scenario is $O(n\ log(n))$. The for-loop on Line 4 of Algorithm 4 has to check every node in the list. The complexity of Line 6 (`pathToRoot(T,` L_i`)`) of Algorithm 4 is related to the height of the tree. For a complete tree where each location has m children, extracting a path from any arbitrary location to root has a complexity of $O(log_m(n))$. In the worst case, m is 2. Hence, the complexity becomes $O(n\ log(n))$.[3]

6.3.2 Workload Distribution Policy

The next step after determining where to execute the kernel is to determine how the workload should be distributed among the children of that location. Assigning all the iteration space to one single child of a location will heavily underutilize our computational resources. While one of them is doing all the work, all the other ones are idle. This is not an ideal scenario for the current HPC systems. The efficient scenario is to partition the iteration space equally. If all children are of the same kind (e.g., all of the children are of the same GPU device), the iteration space should be partitioned in equal sizes. However, in case the children are of different kinds, the iteration space should be distributed according to their computational capabilities. If one of them is computationally more powerful that the rest, we should assign more iteration space to that location in comparison to the other children. As a result, we

[3]Declaring paths as sets (e.g., `unordered_set` in C++ STL) leads to an $O(1)$ complexity for searching a location in the path.

need different strategies in workload distribution to accommodate with different cases.

Gecko provides a set of distribution policies (or in other words, strategies) that a programmer is able to select among them. These policies enable the application developers to support the above-mentioned cases to keep all the resources busy at all times. Gecko partitions the iterations of a `for`-loop and assigns each partition to a location. The partitioning process is governed by our distribution policies. Figure 6.5 demonstrates how these policies partition a `for`-loop with 1,000,000 iterations. These policies are called `static`, `flatten`, `percentage`, `range`, and `any`. They are discussed in detail in the following subsections.

6.3.2.1 Static

In the *static* distribution, the iteration space is divided evenly among children of that location. This policy is similar to the default distribution approach in the directives-based approaches OpenACC [82] and OpenMP [83]. Figure 6.5 shows how the iteration space, shown as a box, is partitioned among location. Since the destination location has two children, the space is partitioned into two parts. The partition assigned to each child is further divided among its children until the leaf nodes of the tree are reached. As a result, the leaf nodes that are closer to the root will have bigger shares in comparison to the others.

6.3.2.2 Flatten

The *flatten* distribution is similar to the static distribution in its approach. In *static*, all siblings of a location take an equal share of the workload. However, in the case of the *flatten* distribution, all *the leaf nodes* contribute equally. Figure 6.5b shows how the iteration space is partitioned into eight equally-sized parts since we have eight leaf nodes in total. Each partition is assigned to a single location.

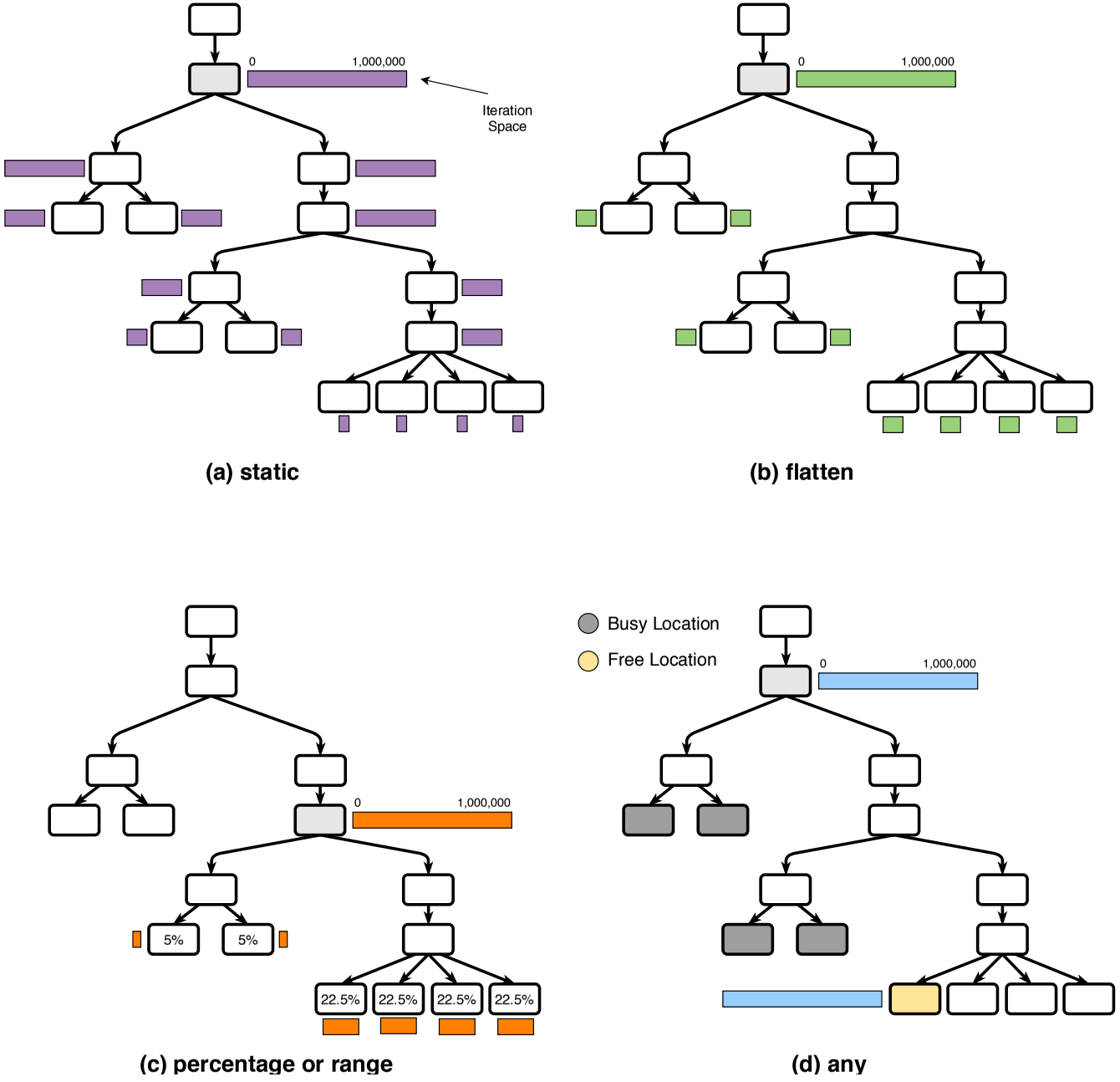


Figure 6.5: Four different workload distribution policies that are supported in Gecko.

6.3.2.3 Percentage or Range

Gecko also provides customized workload distribution among locations. An application developer is able to partition the iteration space among children of a location. The *range* policy accepts an array of integers that specifies the partition size for each child. The *percentage* policy accepts an array of percentages that specifies the partition in percentages with respect to the whole iteration space. In case the number of children of a location is fewer than the partitions specified by the developer, Gecko assigns the rest of the partitions in a round-robin or work-stealing fashion. Figure 6.5**c** shows an example of how the range and percentage policies partition the iteration space. The example shows the `percentage:[5,5,22.5,22.5,22.5,22.5]` case. The locations on the left are assigned only 5% of the iteration space, while each location on the right is assigned 22.5% of the whole iteration space.

6.3.2.4 Any

In some cases, we are interested in engaging only one of the children in the execution process. In such cases, Gecko finds an idle location among children of the chosen target. Alternatively, based on the recorded history, Gecko can choose the best architecture for this kernel if we are targeting a multi-architecture virtual location. Figure 6.5**d** shows an example of how the *any* policy works. One can observe how Gecko chooses the yellow location since the first four gray ones are busy with other jobs, and the yellow location is the first available child of the light-gray location.

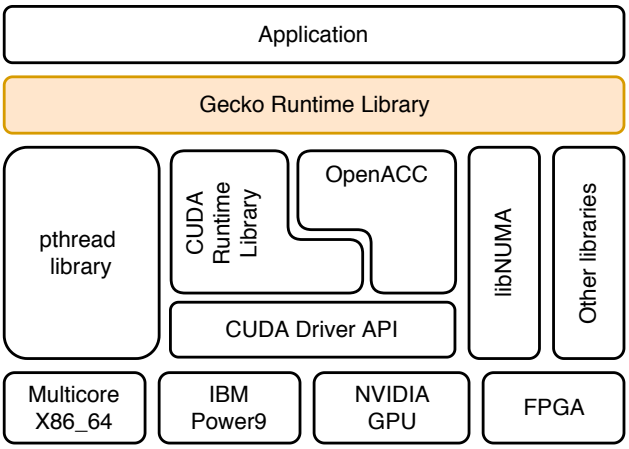


Figure 6.6: An overview of Gecko's architecture. This figure shows how the Gecko Runtime Library (GRL) sits on top of other libraries to abstract the application from the various hardware and software combinations.

6.3.3 Gecko Runtime Library

Figure 6.6 displays how Gecko is placed in the software stack of the high-performance computing. In this architecture, the application sits on top of Gecko; as a result, Gecko provides a high-level of abstraction to the application. The application on top of the architecture deals with a set of abstract entities. At execution time, such entities would be associated with the available hardware in the system.

Below Gecko, there are many different software libraries and programming models that let Gecko target various architectures and platforms (shown at the bottom of Figure 6.6). In a nutshell, Figure 6.6 shows how an application does not need to change when the underlying platforms are changed.

6.3.3.1 Hierarchy Maintenance

The Gecko Runtime Library (GRL) utilizes an internal *tree data structure* to maintain the hierarchy that Gecko proposes. Each location in the tree has another location as its parent and multiple (or no) locations as its children. The *root* location of the tree has no parent.

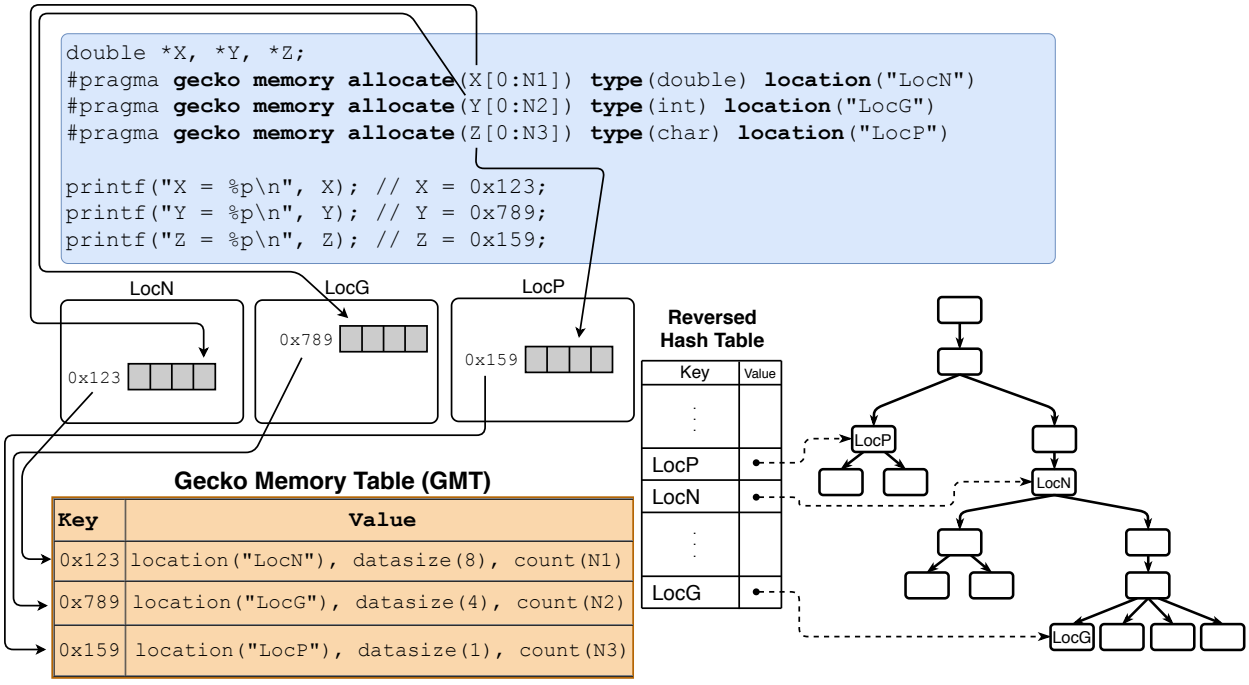


Figure 6.7: Reversed Hash Table (RHT) with a code snippet that demonstrates the `memory allocate` clause in Gecko and its effect on the Gecko Memory Table (GMT).

Locations are accessed by their unique name. However, traversing the tree each time to find a location is not a performance-friendly approach. Hence, Gecko uses a Reversed Hash Table (RHT) to find and access a location. RHT is a key-value-based container that maps a string of characters (representing the name of a location) to its corresponding location[4] in the tree. Figure 6.7 shows RHT for a sample tree in Gecko. Every location has an entry in RHT. When we add or remove a location to or from the hierarchy, RHT is updated with the changes. Such a design allows Gecko to access a location in $O(1)$ complexity instead of $O(n)$. In our implementation, we declared RHT as an `unordered_map` from C++ Standard Template Library (STL).

[4]The *value* column in RHT contains a pointer that points to its corresponding node in the internal tree.

Algorithm 5 The Region Pseudocode

Input: T: *Gecko*'s hierarchical tree structure.
Input: *varList*: List of variables
Input: *policy*: The chosen distribution policy
Input: *kernel*: The computational kernel

```
1: function REGION(T, varList, policy)
                                   ▷ Refer to Algorithm 6 for binding algorithm.
2:     threadList ← bindThreadsToLeafLocations(T)
3:     locList ← extractLoc(varList)
4:     loc ← mcdAlgorithm(T, locList)
5:     leafList, configs ← splitIterSpace(loc, policy)
6:     for each th ∈ threadList do
7:         if th.loc ∈ leafList then
8:             S ← configs[th.loc]
9:             kernel.execute(th.loc, S.begin, S.end)
10:        end if
11:    end for
12:    threadList.waitAll()
13: end function
```

6.3.3.2 Workload Maintenance

GRL is also responsible for workload distribution among locations. As a program encounters a for-loop that is decorated with Gecko directives to distribute the workload, the iteration space is partitioned among the children based on the execution policy that is chosen.

Algorithm 5 shows the steps that are followed by Gecko. First up, as shown in Line 2, all leaf locations in the hierarchy are extracted, and a thread is assigned to them as we will discuss in Section 6.3.3.3. Each thread is responsible for initiating a job on the location and waiting for the location to finish its job.

Secondly, we use the MCD algorithm to find the proper location to execute the kernel. We extract the list of locations from the variables used in the region (Line 3). Then, Line 4 in Algorithm 5 shows how to call the MCD algorithm to choose our target location. Thirdly, on Line 5, the iteration space is partitioned among the children of a location based on the execution policy chosen as discussed in Section 6.3.2. Fourthly, Line 6 of Algorithm 5

Algorithm 6 The Binding Algorithm

Input: T: *Gecko*'s hierarchical tree structure.
Output: Allocated Threads

```
1: function BINDTHREADSTOLEAFLOCATIONS(T)
2:     if T.isModifiedSinceLastVisit() then
3:         numLocs, listLocs ← findAllLeafNodes(T)
4:         allocatedThreads.releaseAll()
5:         allocatedThreads.create(numLocs)
6:         i ← 0
7:         for each loc ∈ listLocs do
8:             allocatedThreads[i].assign(loc)
9:             i ← i + 1
10:        end for
11:    end if
12:    return allocatedThreads
13: end function
```

specifies how threads dispatched on Line 2 take control of their corresponding location and execute their share of iteration space. And finally, in Line 12, Gecko waits for all threads to finish their assigned job. After Line 12, the devices are free for the next round of execution.

6.3.3.3 Thread Assignment

The assignment of threads follows the steps in Algorithm 6. If the tree structure that represents the hierarchy in Gecko remains unmodified since our last visit, Gecko does nothing, since threads are already assigned to the locations. However, the hierarchy may alter between the execution of consecutive regions. In such cases, the updated leaf locations are extracted from the hierarchy (Line 3 in Algorithm 6). In Lines 4-5, threads are released, and then, in Lines 6-10, threads are updated with their new assignments.

6.3.4 Memory Allocation in Gecko

This section discusses challenges that Gecko faces with respect to memory allocation.

6.3.4.1 Uncertainty in Location Type

Uncertainty in location type makes memory allocation a challenging problem. The process of allocation has to be postponed to execution time since only then does Gecko have enough knowledge to perform the allocation. Consequently, the memory allocation process is not straightforward and becomes a challenge.

Algorithm 7 shows how Gecko allocates memory. It starts by recognizing if the location chosen is a leaf location in the tree or not. Allocated memory in leaf locations is private memory that is only accessible to the location. Based on the location type, the `malloc` or `cudaMalloc` APIs are called.

If the location chosen is not a leaf location, Gecko traverses the subtree beneath the location and determines whether all of its children are multicore or not. If they are all multicore, we will use a host-based memory allocation API [30, 27], such as `malloc` and `numa_alloc` [56]. Otherwise, Gecko allocates memory from the memory domain introduced by CUDA known as Unified Virtual Memory (UVM) [57].

Gecko utilizes another hash table, known as Gecko Memory Table (GMT), to trace the memory allocations within the system. Figure 6.7 displays a code snippet that utilizes Gecko's directives to allocate memories in the system. It shows the sequence of actions that takes place when memory is allocated. First up, Gecko allocates a block of memory to the designated location with the data type and the total number of elements that the user requested (known as memory traits). For instance, in the second line of the code snippet in Figure 6.7, the programmer requests `N1` double-precision elements in `LocN`. Algorithm 7 returns the suitable allocation mechanism. After using the API function returned by Algorithm 7, the target variable (in this case, `X`) holds the memory address (`0x123`). Then, Gecko inserts an entry into GMT with 0x123 as the key and the memory traits as the value. The `extractLoc` function in Algorithm 5 utilizes GMT to find the location in which each

Algorithm 7 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 ← multiCoreAlloc
 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 ← multiCoreAlloc
13:         else if gTree.getType(loc) ∈ {GPU, MULTICORE} then
14:             allocFunc ← cudaMallocManaged
15:         else
16:             return Err_UnrecognizedLocationType
17:         end if
18:     end if
19:     return allocFunc
20: end function
```

variable in its input parameter, `varList`, resides.

6.3.4.2 Distance-based Memory Allocations

Gecko provides distance-based allocations to the programmer. Unlike the ordinary allocations that were discussed in Section 6.3.4.1, the programmer does not specify the target location. In the case of ordinary allocations, a programmer manually specifies the target location for the memory block. However, in distance-based allocations, GRL should infer the target location at execution time. Such memory allocations are performed with respect to the location that the computational kernel is targeted to be executed.

Distance-based allocations in Gecko are declared, as either `close` or `far`. Close allocations are performed within the location targeted for running the kernel. However, far

allocations are performed within the parent (or grandparents) of the targeted location. For instance, in Figure 6.7, if location `LocH` is chosen for kernel execution, declaring a memory as `close` will allocate memory within `LocH`, while declaring it as `far:3` will allocate it in `LocD` (since it is its third grandparent).

Gecko also provides `realloc` and `move` keywords for distance-based memory allocations. The `realloc` keyword causes a memory block to be allocated when entering a region and to be freed when exiting a region. However, with `move`, a memory block is allocated on the first touch. Then, if required, it is moved around within the hierarchy between the subsequent Gecko regions in the source code.

Gecko addresses the distance-based memory allocation challenge by *declaring* and *utilizing* the distance-based allocations with minimum code changes. Figure 6.8 shows the sequence of actions that takes place so that Gecko performs a distance-based allocation. The code snippet in Figure 6.8, annotated with Gecko directives, is utilizing `X` and `Y` variables where each variable points to `N` double precision floating-point numbers that are allocated in `LocN` and `LocG`, respectively. We declare a distance-based memory space, named `d`, that is designated as a `close` memory (❶). Gecko starts by allocating a dummy memory block on the heap (❷). The dummy block is basically a handle to distinguish the distance-based allocations from the regular ones. Then, Gecko inserts an entry into GMT to record the memory request (❸), and allocates a memory block as soon as it determines the destination location. We updated the structure of GMT, as shown in Figure 6.8, to handle distance-based allocations. The two new fields, `distance` and `real_addr`, hold the distance parameter and the address of allocated memory block in the destination target, respectively.

As we reach the `region` section in our code (❹), Gecko finds the target location to run the kernel. Based on the fact that the `X` and `Y` variables are our non-distance-based variables in the Gecko region (❺), the MCD algorithm will choose `LocG` as the location to execute the region and the target location to allocate variable `d`. Then, Gecko allocates a memory

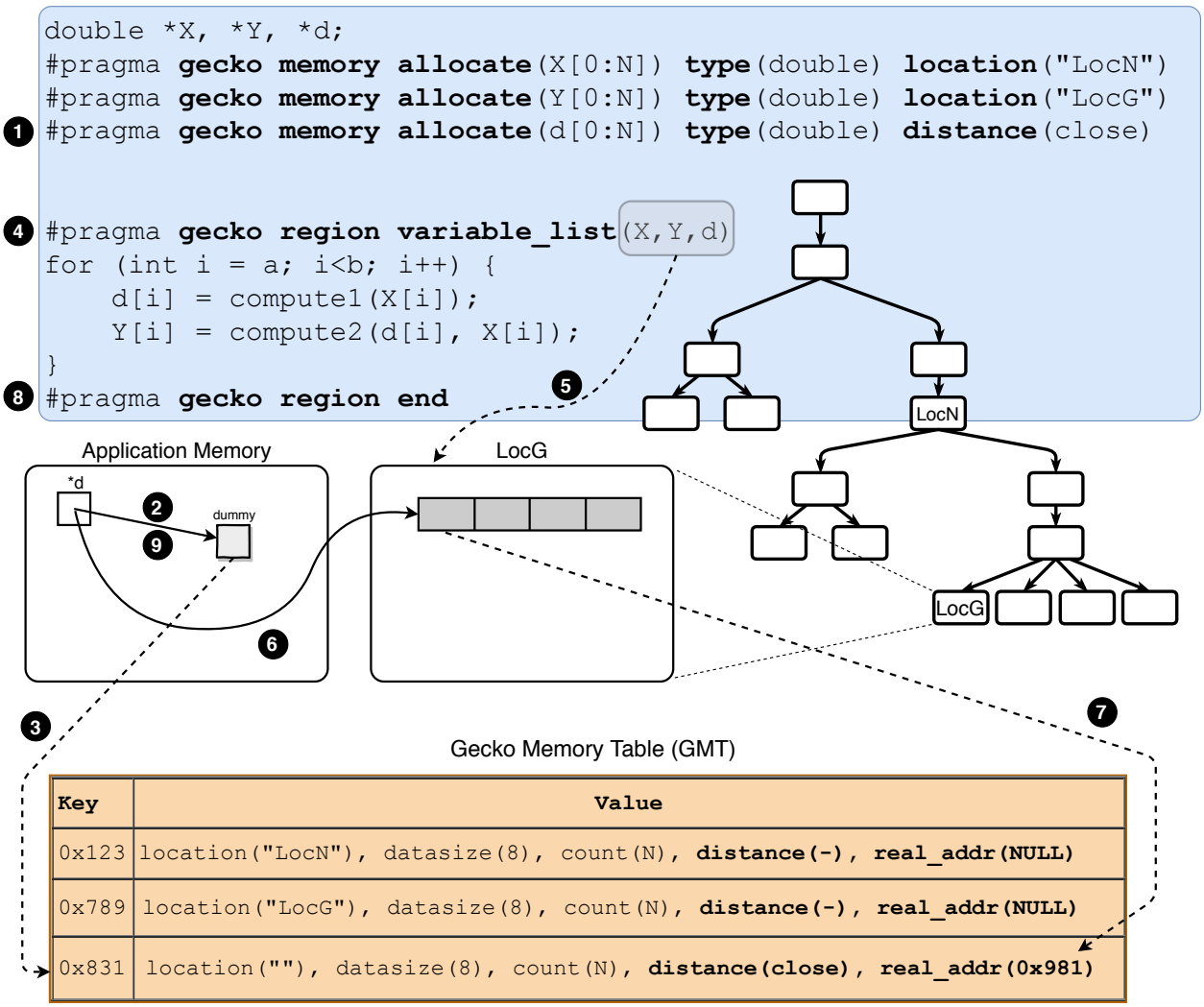


Figure 6.8: Steps taken by Gecko that show how distance-based memory allocations are performed with minimum code modification. By simply annotating the memory allocation clause with `distance`, Gecko governs the correct state of the pointers internally.

block within `LocG`, reassigns the variable `d` to the new allocation (❻), and finally, updates GMT with the new address (populating the `real_addr` field as shown with ❼). Until the end of the region, the variable `d` points to the valid memory block in `LocG`. As we reach the end of the `region` (❽), the variable `d` reverts to its original value, which was the dummy variable allocated before (❾).

6.4 Gecko in Use

This section is dedicated to demonstrate how to write a simple application with Gecko. We will show how a single source code can be used to target single or multiple CPUs, single or multiple GPUs, and a combination of them. For a complete list of Gecko's capabilities, please refer to Gecko's Github repository[5].

The right side of Figure 6.9 shows the source code of the Stream benchmark in Gecko. We will go through the lines of this code and clarify what each line does. Line 1 loads the configuration file from the disk. The configuration file includes the definition of location types, the definition of locations, and the declaration of hierarchies among locations. The top left of Figure 6.9 shows an example of a configuration file for Configuration **a** in the bottom left of Figure 6.9. Lines 3-5 will allocate memory with `array_size` elements and type `T` at the location "`LocH`".[6] We hard-coded the destination location to "`LocH`" for the three memory allocations. This provides greater flexibility to the application. We can place "`LocH`" anywhere in the hierarchy since we have defined it to be a virtual location in our configurations.

We tested our application with different configurations. A list of all configurations that we targeted is shown in Figure 6.9 (bottom left). For example, Configurations **a** and **b** target only multicore systems. However, Configurations **c**-**e** target single- and multi-GPU systems. Finally, Configuration **f** targets a multi-architecture system to execute our application. If we change the configuration file so that it represents any of the configurations of **a** to **f**, without recompiling the source code, our program is able to target different architectures without significant performance loss.

Lines 7-13 show a computational kernel that initializes three arrays that were allocated

[5] `https://github.com/milladgit/gecko`

[6] The `array_size` parameter is an input to the program. The `T` type is a template parameter that accepts a data type. We chose `double`.

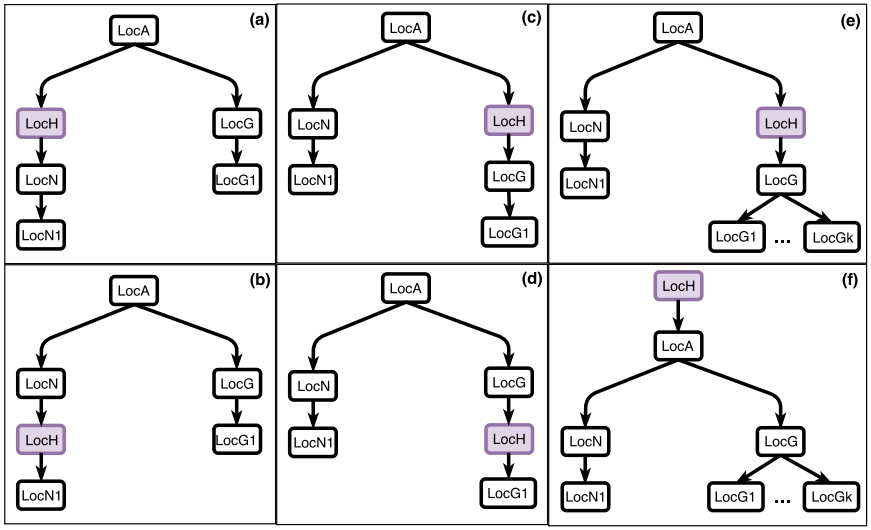
```
# Content of 'gecko.conf' file that represents
# Configuration (a)

loctype;name,host;kind,x64,Skylake;num_cores,4;mem,4MB;
loctype;name,tesla;kind,NVIDIA,CC7.0;mem,4GB
loctype;name,NODE_MEMORY;kind,Unified_Memory;size,16GB

location;name,LocA;type,NODE_MEMORY;
location;name,LocN,LocG;type,virtual
location;name,LocN1;type,host;
location;name,LocG1,LocG2;type,tesla

location;name,LocH;type,virtual

hierarchy;children,+,LocH,LocG;parent,LocA
hierarchy;children,+,LocN;parent,LocH
hierarchy;children,+,LocG1;parent,LocG
hierarchy;children,+,LocN1;parent,LocN
```




```
#pragma gecko config file("gecko.conf")
T *a, *b, *c;
#pragma gecko memory allocate(a[0:array_size]) type(T) location("LocH")
#pragma gecko memory allocate(b[0:array_size]) type(T) location("LocH")
#pragma gecko memory allocate(c[0:array_size]) type(T) location("LocH")

#pragma gecko region gang vector exec_pol("runtime") variable_list(a,b,c)
for (int i = 0; i < array_size; i++) {
    a[i] = initA;
    b[i] = initB;
    c[i] = initC;
}
#pragma gecko region end
#pragma gecko region pause

std::chrono::high_resolution_clock::time_point t1, t2;
// Run triad in loop
t1 = std::chrono::high_resolution_clock::now();
for (unsigned int k = 0; k < num_times; k++) {
    #pragma gecko region gang vector exec_pol("runtime") \\
                variable_list(a,b,c)
    for (int i = 0; i < array_size; i++) {
        a[i] = b[i] + scalar * c[i];
    }
    #pragma gecko region end
    #pragma gecko region pause
}
t2 = std::chrono::high_resolution_clock::now();
double runtime = duration_cast<duration<double>>(t2 - t1).count();

#pragma gecko memory free(a,b,c)
```

Figure 6.9: Right: A snapshot of the Stream benchmark with Gecko's directives. **Top Left:** A sample configuration file that represents Configuration (a) on the bottom left. The first two lines are comments. **Bottom Left:** Visualization of different configurations. Note how placing "LocH" in different positions in the hierarchy results in targeting different architectures. Configuration **(a)** and **(b)** target general-purpose processors. Configurations **(c)** and **(d)** target one single GPU. Configurations **(e)** and **(f)** target multi-GPU and multi-architecture systems, respectively.

previously. The `runtime` execution policy specifies that Gecko will extract the *main* policy from an environmental variable (known as `GECKO_POLICY`) at the execution time. One should set this variable to any of the policies previously defined in Section 6.3.2. The `pause` statement in Line 14 asks Gecko to wait on all computational resources (processors and GPUs in our case) to finish their assigned job before continuing with the next statement in the code.

Lines 19-27 show a loop that contains the main TRIAD kernel of the Stream benchmark and calls the kernel `num_times` times. Similar to the original TRIAD kernel, it is a `for`-loop that multiplies each element in array c to an scalar value, adds it to an element in array b, and stores the final value in array a. Depending on the chosen configuration and execution policy, at the run time, Gecko splits the iterations of the main loop in Line 22 (from 0 to `array_size`) among the processors and GPUs. For instance, for Configuration **e** where the number of GPUs is four, each GPU will process $array\_size/4$ iterations.

The benchmark calls the high resolution timers in Lines 18 and 28 before and after the for-loop to measure the total execution time of the TRIAD kernel. And finally, Line 34 asks Gecko to free all memories allocated in the system.

For a detailed description of Gecko's directives and their clauses, please refer to Appendix A at the end of this dissertation.

6.5 Gecko's Implementation

To implement Gecko, we developed a Python script that takes a Gecko-annotated source code as input and generates a conformed C++ source code with the OpenMP and OpenACC directives as output. Utilizing these directive-based programming models leads to minimizing source code modification. Since our model is developed as a language feature, it can be easily extended to other languages, like C and Fortran, as well.

Figure 6.10 shows the compilation framework that is used to compile a Gecko-annotated source code. After the transformation process, we will set the compiler flag to generate code for both multicore and GPU. During the execution time, Gecko will choose the correct device (multicore or GPUs) accordingly. Our motivation behind utilizing a script rather than a compiler is to minimize the prototyping process and implement our proof-of-concept approach.

The output executable file is a fat binary file which contains the executable code for all kernels in the code and for both multicore and GPU architectures. At runtime, Gecko will choose the correct version of the code.

Similar to the OpenACC and OpenMP programming models, Gecko is a directive-based programming model. 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 their allocated job[7]. Directives provide a level of flexibility that library-based approaches do not necessarily provide. Directives also require users to add fewer additional lines most of the times to the code thus not increasing the Lines of Code (LOC) by a large number. Using directives mostly means fewer code alterations and developers can start from a serial version of the code. Furthermore, directives can be ignored in some cases (like debugging, testing, targeting multicore) without any code modification.

6.6 Results

This section is dedicated to assessing the performance of Gecko on homogeneous and heterogeneous platforms. At first, the experimental setup that was used to assess Gecko is described briefly. Then, we will utilize the Stream benchmark developed in the previous

[7]In addition to the above capabilities, Gecko also provides copying and moving data among different locations in the hierarchy. It also provides facilities to register and unregister already allocated memory to Gecko. Please refer to the Github repository of Gecko for more information.

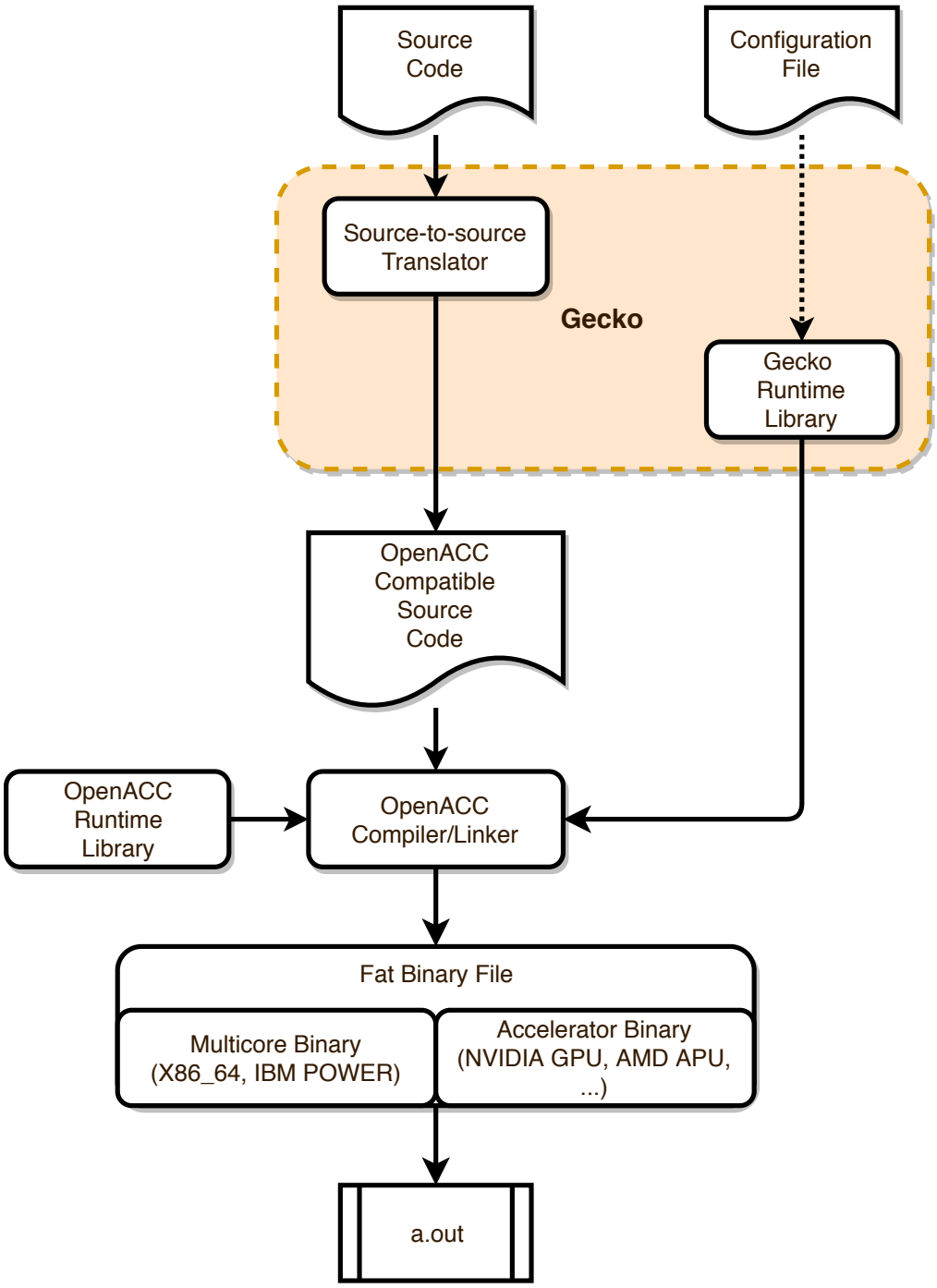


Figure 6.10: An overview of the compilation stack in Gecko.

Table 6.1: List of all the benchmarks from Rodinia that were ported to Gecko and their associated domains

Benchmark	Brief Description
bfs	Breadth-First search (Graph traversal)
cfd	Computational fluid dynamics (CFD) solver (Fluid dynamics)
gaussian	Gaussian elimination (Linear algebra)
hotspot	Hotspot for chip design (Physics simulation)
lavaMD	N-Body simulation (Molecular dynamics)
lud	Lower-upper (LU) Decomposition (Dense linear algebra)
nn	k-Nearest Neighbor (Data mining)
nw	Needleman-Wunsch (Bioinformatics)
particlefilter	Medical imaging
pathfinder	Grid traversal
srad_v2	Image processing

section to measure the sustainable bandwidth of different architectures. Furthermore, the Rodinia suite [16, 17] was ported to Gecko [36] to assess the effect of utilizing multiple GPUs in a single-GPU benchmark with minimum code intervention.

The Rodinia benchmarking suite is a collection of scientific benchmarks to assess the performance of heterogeneous computing infrastructures. Benchmarks in Rodinia have been developed in OpenMP, OpenCL, and CUDA. For comparison purposes with the above-mentioned programming models, we ported benchmarks in the Rodinia suite to Gecko [36]. They are available online on Github[8]. Table 6.1 shows the list of the ported benchmarks with a brief description of the domain they belong to.

6.6.1 Steps in Porting Applications to Gecko

We successfully ported benchmarks from the Rodinia suite to Gecko by annotating their source code with Gecko's proposed directives.[9] The annotation process is as follows: 1) The

[8]https://github.com/milladgit/gecko-rodinia

[9]We were unable to compile backprop, hearwall, kmeans, leukocyte, myocyte, streamcluster of the 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*.

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 certain locations in the code to ensure the consistency of the algorithm; and finally 6) All the memory deallocations are replaced with `memory free` clauses. Basically, these are the modifications required for any code to use *Gecko*.

6.6.2 Experimental Setup

We use the NVIDIA Professional Services Group (PSG) cluster [80] and Sabine [97]. PSG is a dual socket 16-core Intel Haswell E5-2698v3 at 2.30GHz with 256 GB of RAM. Four NVIDIA Volta V100 GPUs are connected to this node through a PCI-E bus. Each GPU has 16 GB of GDDR5 memory. We used CUDA Toolkit 10.1 and PGI 18.10 (community edition) for the OpenACC and CUDA codes, respectively.

Sabine is a dual socket 14-core Intel Haswell E5-2680v4 at 2.40GHz with 256 GB of RAM. Two NVIDIA Pascal P100 GPUs are connected to this node through a PCI-E bus. Each GPU has 16 GB of GDDR5 memory. We used CUDA Toolkit 10.1 and PGI 18.10 for the OpenACC and CUDA codes, respectively.

6.6.3 Sustainable Bandwidth

Figure 6.11 shows the sustainable memory bandwidth by different programming models and libraries. We used the BabelStream benchmark [23] to measure the memory bandwidth. BabelStream provides a set of Stream benchmarks in various programming models and libraries: OpenMP, CUDA, OpenACC, Kokkos, and RAJA. We also implemented a version of Stream benchmark based on Gecko, to compare the sustainable bandwidth each approach

provides. The Gecko version of Stream is shown on the right side of Figure 6.9. Results are shown for both systems (PSG and Sabine) and for three different scenarios: (1) multicore execution, (2) multi-GPU execution, and (3) heterogeneous execution. In our experiments with Stream, we set the array size to 256,000,000 double-precision elements for each array, which results in 6 GB of data in total. We ran the TRIAD loop 200 times and took the average of their wall-clock time to report the execution time. The multicore results reveal a negligible difference between the Gecko version of Stream with other methods (less than 5 GB/s difference in comparison to OpenACC for both PSG and Sabine). Gecko's results are reported for configurations **(a)** and **(b)**. Sabine's main processor provides 18 GB/s more bandwidth to access the main memory in comparison to PSG. It is due to a subtle difference in their memory bandwidth. The theoretical peak memory bandwidth for Sabine's dual processors is 153.6 GB/s, however, the peak bandwidth for the dual processors in PSG is 136 GB/s.

Gecko's memory allocation algorithm provides a better locality for Configuration **(b)** in comparison to Configuration **(a)**. Moreover, despite the better locality of Configuration **(b)** in comparison to Configuration **(a)**, we did not see a notable difference in the results. The maximum difference between those configurations for both PSG and Sabine was less than 0.5 GB/s in our experiments.

Similarly, Gecko's performance is not significantly affected when we target GPUs. Single- and multi-GPU results are shown in Figure 6.11. Despite the difference in their hierarchy, Configurations **c** and **d** provide the same bandwidth since our memory allocation algorithm returns the same API function in both cases. However, since CUDA, Kokkos [15], RAJA [46], and OpenACC versions of Stream use non-UVM (Unified Virtual Memory) memory to perform the benchmark, there is a 17 GB/s difference in the bandwidth. In addition, Gecko provides single- and multi-GPU execution with one single source code while other programming models and libraries support only one single GPU in their implementation. Multi-GPU

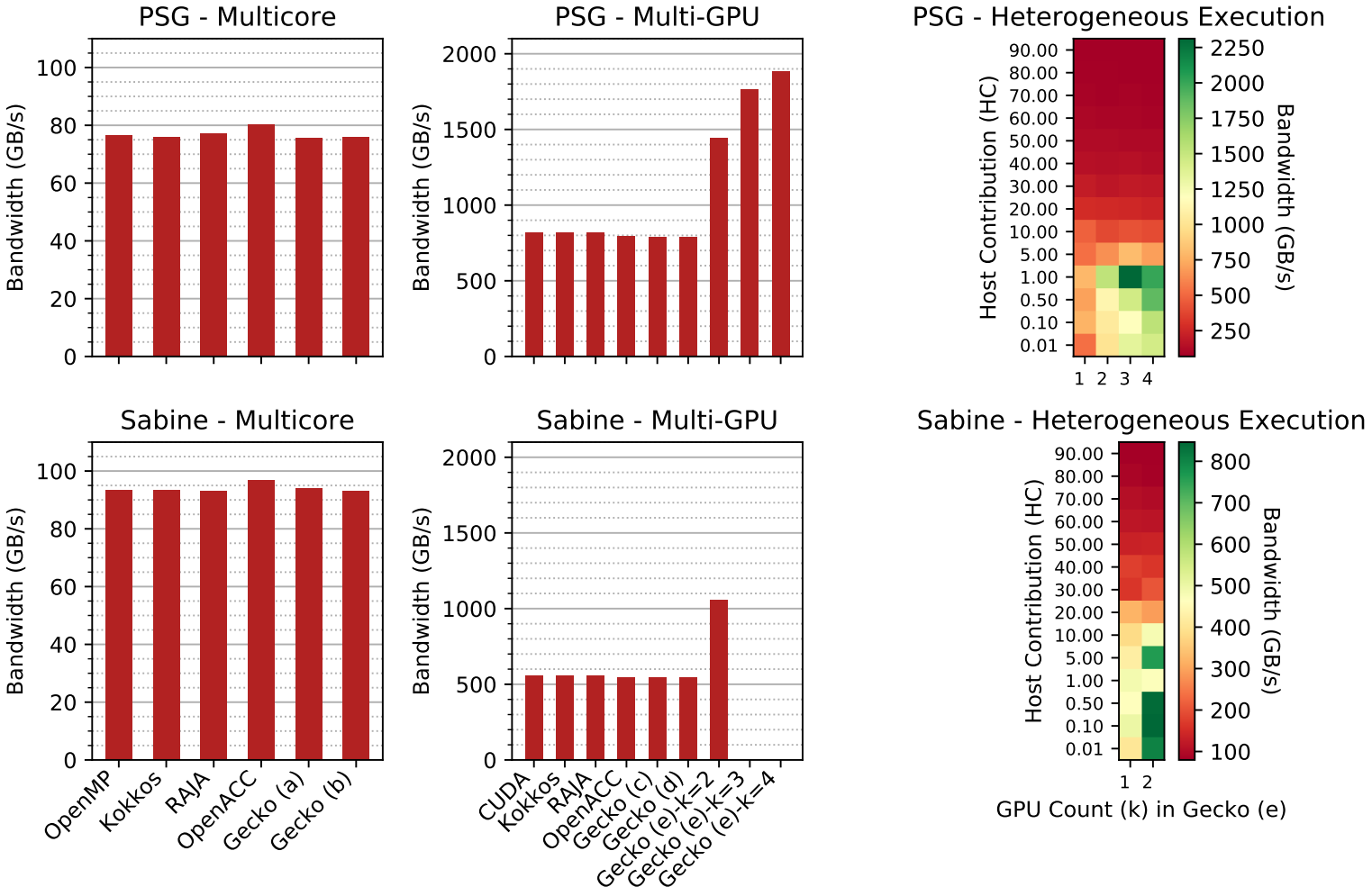


Figure 6.11: Sustainable bandwidth of the Stream benchmark on PSG and Sabine. **Left:** multicore systems, **Center:** single- and multi-GPU systems, and **Right:** heterogeneous systems.

execution of Stream for above-mentioned methods requires significant source code modification.

Gecko's Stream source code, as shown in Figure 6.9, is able to utilize GPUs by only modifying the configuration file. Gecko was able to provide 1.8 TB/s and 1.9 TB/s with four GPUs on PSG and Sabine, respectively. Figure 6.11 shows the results of GPU execution (**Center**), which are represented with "Gecko (e)-k", where k specifies the number of GPUs in the hierarchy. Gecko supports heterogeneous execution as well with no code alteration. Heatmap plots in Figure 6.11 (**Right**) show the sustainable bandwidth for the heterogeneous execution of Stream on the main processor and GPUs in the system, simultaneously. Host contribution (HC), which specifies the amount of workload assigned to the host processor, varies from 0.01% to 90% of the total iterations of the TRIAD kernel, and the rest is divided equally between the GPUs. For instance, in the case of K=4 and HC=1%, 2,560,000 out of 256,000,000 iterations are assigned to the host processor and the rest (253,440,000 iterations) is divided among four GPUs; it means Gecko assigns 63,360,000 iterations to each GPU. We utilized the `percentage` execution policy to represent the above-mentioned cases. For instance, the equivalent execution policy in Gecko for the above example would be "percentage:[1.00,24.75,24.75,24.75,24.75]".

The low values of HC (less than 1%) show promising bandwidth results. Sabine's heterogeneous execution is able to reach 838 GB/s when HC is 0.5% and both GPUs are utilized. In the case of Sabine, the heterogeneous execution does not improve the bandwidth since the bandwidth of multi-GPU execution of Stream has surpassed 1 TB/s (1056 GB/s as shown in the **Center** plot for Sabine). However, it is not the case for PSG. Heterogeneous execution has improved the bandwidth on PSG. While multi-GPU execution on all four GPUs of PSG has reached 1.88 TB/s, the bandwidth for heterogeneous execution has improved and reached 2.31 TB/s, where HC is 1% and three GPUs are utilized.

Table 6.2: List of benchmarks in the Rodinia Suite that were 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	Input	Data Type	A	B
bfs	1,000,000-edge graph	Mixed	5	39
cfd	missile.domn.0.2M	Mixed	5	9
gaussian	4096 × 4096 matrix	SP	3	12285
hotspot	1024 data points	DP	2	20
lavaMD	50 × 50 × 50 boxes	Mixed	1	1
lud	2048 data points	SP	2	4095
nn	42764 elements	SP	1	1
nw	2048 × 2048 data points	int	4	4095
particlefilter	1024 x 1024 x 40 particles	Mixed	9	391
pathfinder	width: 1,000,000	int	1	499
srad	2048 x 2048 matrix	Mixed	7	12

6.6.4 Rodinia Benchmarks

The Rodinia benchmark suite includes a number of benchmark applications where each one acts as a representative of various domains. Our justification behind choosing these benchmarks was their diverse computational pattern and their state-of-the-art algorithms, which will be discussed below.

Recent advances in artificial intelligence and machine learning owe most of their success to the linear algebra methods (e.g., Gaussian elimination and LU[10] decomposition). Moreover, advances in autonomous driving and health domain were achievable due to the advances in image processing and data mining. Last but not least, we also considered the workloads of the scientific simulation frameworks, like N-body simulations in molecular dynamics and CFD[11] solvers in fluid dynamics simulations, in our investigations. Table 6.1 shows the list of benchmarks that we picked to be ported to Gecko.

We used our version of the updated Rodinia suite [36]. Table 6.2 shows the list of benchmarks used in this paper with their corresponding input, data type, the total number

[10]Lower-upper
[11]Computational fluid dynamics

of kernels, and the total kernel launches at the execution time. Figure 6.12 shows the multi-GPU speedup of all applications (**Left**) and heterogeneous execution time of *cfd* and *srad_v2* (**Right**) on PSG and Sabine. Speedup results were obtained by utilizing the `static` execution policy to equally distribute the workload among the GPUs. For the heterogeneous execution, we followed a similar approach as we did for the Stream benchmark. For the speedup results, the X axis shows the applications of the Rodinia benchmark, and the Y axis shows the speedup with respect to a single GPU. Each bar represents different number of GPUs (represented by K).

The results indicate that multi-GPU utilization is not a suitable option for all benchmarks in Rodinia. The *cfd* and *srad_v2* benchmark applications show promising results in scalability. The performance of *cfd* improves with each additional GPU. This performance increase ceases after three GPUs are in use. Adding a fourth GPU to the configuration does not improve performance. However, *srad_v2* performs differently. As we increase the number of GPUs, the overall speedup improves as well. As shown on the right side of Figure 6.12, the heterogeneous execution of *cfd* on PSG does not lead to a performance improvement. However, the heterogeneous execution of *cfd* on Sabine (two GPUs and 50% host contribution) leads to a 2X speedup with respect to one single GPU. The reason behind the superiority of Sabine over PSG is due to two reasons: 1) Sabine has better host memory bandwidth. Results in Figure 6.11 reveal the 17 GB/s difference in main memory bandwidth (Left) between Sabine and PSG. 2) Utilizing all available GPUs is not always a good idea. Splitting the iteration space among many GPUs leads to less amount of work to be done by GPU, which implicitly leads to more overhead. The *cfd* results on PSG confirm our finding as well. For all values of HC (except HC=0), utilizing two GPUs leads to better performance in comparison to three or four GPUs. The *srad_v2* benchmark benefits more from the heterogeneous execution as the heatmap results show. In comparison to one single GPU (one GPU and HC=0), if we utilize three or four GPUs while HC is 90%, the speedup is 15X for

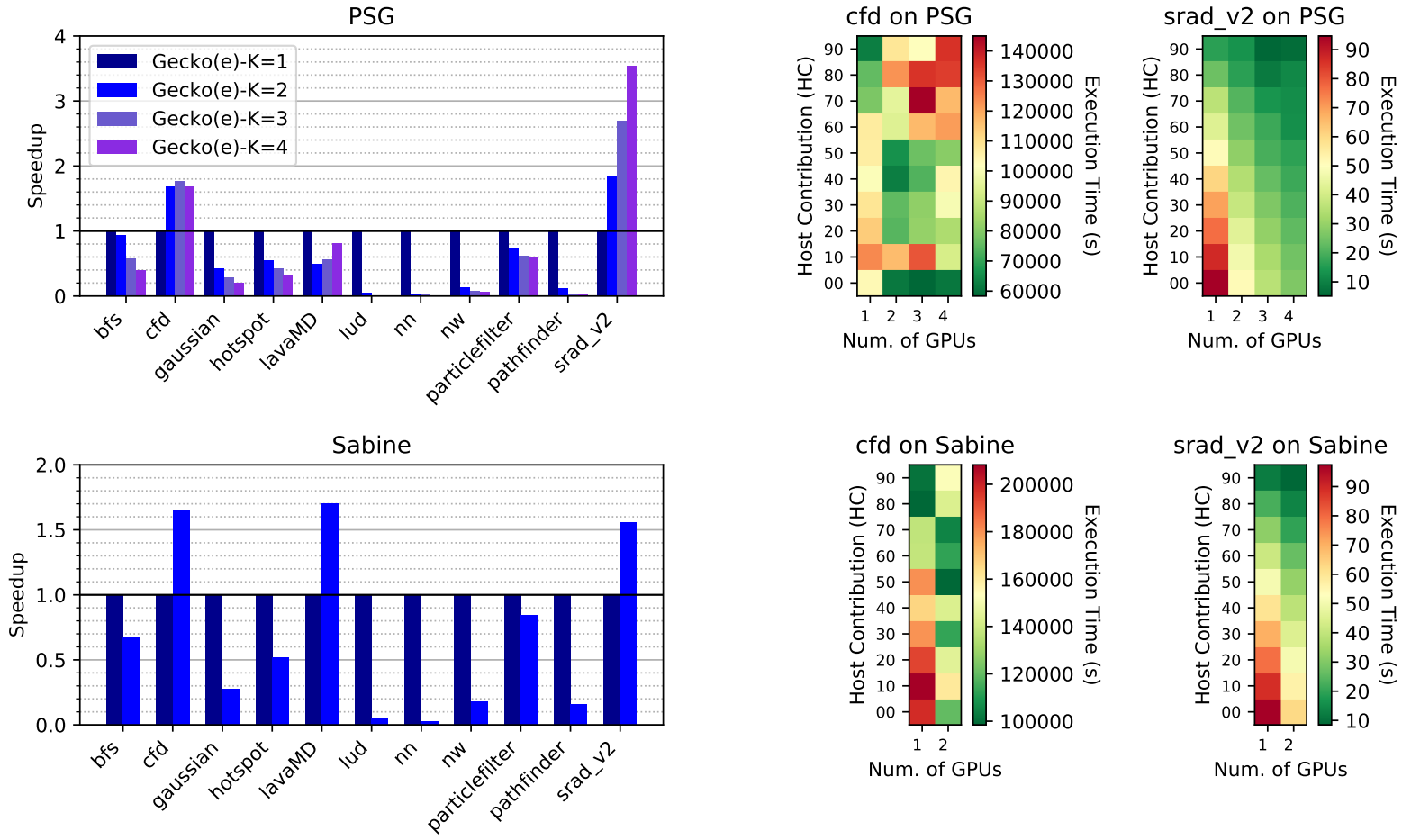


Figure 6.12: Left: Speedup results of the multi-GPU execution of the Rodinia benchmarks on PSG and Sabine, specified with Configuration **(e)** in Figure 6.9. **Right:** Heatmap of the execution time of *cfd* and *srad_v2* for different host contributions and number of GPUs.

PSG. Similarly, for Sabine, the speedup becomes 11.5X, when two GPUs are utilized and HC is 90%.

Other benchmark applications (*bfs*, *lavaMD*, and *particlefilter*) do not scale as we increase the number of utilized GPUs. The performance degradation is due to uncoalesced and random memory accesses in such applications. The *bfs* benchmark traverses all the connected components in a graph. Thus, the memory accesses follow a random pattern. The *lavaMD* benchmark goes through all atoms in the system and computes the force, velocity, and new position of each atom. It uses a cutoff range to limit unnecessary computations. However, such cutoff ranges may include atoms that are currently residing in another GPU device. The *particlefilter* benchmark visits elements of a matrix using two nested *for*-loops. In all of the above-mentioned benchmarks, the false sharing [40] effect on the inter-device level is the primary source of performance degradation. It is highly probable that when device d_1 is executing iteration i, it needs to access other data that are currently residing on device d_2. In such cases, many memory pages have to be invalidated to perform iteration i. The invalidated page has to travel via the PCI-E and NVlink buses, which are not performance-friendly. Heterogeneous execution of *bfs*, *lavaMD*, and *particlefilter* benchmarks does not improve the speedup either. Utilizing the host processor has caused a gradual performance degradation for these benchmarks. In the case of other benchmarks (*gaussian*, *hotspot*, *lud*, *nn*, *nw*, and *pathfinder*), the performance loss is severe. Algorithms that follow a very random memory access pattern like *bfs* and *gaussian* are not a suitable option for either multi-GPU or heterogeneous execution.

Chapter 7

Conclusion

Hardware design of High Performance Computing (HPC) systems are getting more complex and complicated as demand for more performance is increasing. Consequently, the software should adapt itself to these changes and provide facilities to the application to utilize such resources efficiently and easily.

7.1 Current Effort

As scientific applications evolve during their development lifetime, they become more complex in their design. Current applications are developed with nested data structures in their source code. These nested structures adversely affect software development for heterogeneous systems due to the unique nature of such systems: having two separate memory spaces, one for the conventional processor and one for the accelerator. Hence, developers are required to keep track of the data objects in the above-mentioned separate memory spaces and transfer the data between spaces back and forth at arbitrary times. This is a cumbersome task for developers if we are dealing with nested data structures. To that end, we proposed a

novel high-level directive, *pointerchain*, to reduce the burden of data transfers in a scientific application that executes on heterogeneous systems. We developed a source-to-source transformation script to transform the `pointerchain` directive to a number of conformed statements in the C/C++ languages. We observed that using the `pointerchain` directive leads to 36% reduction in both generated and executed code (assembly and binary codes) on the GPU devices. We evaluated the proposed directive using CoMD, a Molecular Dynamics (MD) proxy application. By exploiting OpenACC directives on the CoMD code, the `pointerchain` implementation outperforms the CUDA implementation on two out of three kernels while it achieves 61% of the CUDA performance on the third kernel. We show a linear scalability with growing system sizes when utilizing OpenACC. We have provided a step-by-step approach readily available for any other application.

Additionally, we designed and developed *Gecko*, a novel hierarchical portable model. Gecko is able to target heterogeneous shared memory architectures which are commonly found in modern platforms. Following are some of the unique features of Gecko: (1) The model allows developers to dynamically define available memory spaces in the system in a hierarchical manner, and it provides the flexibility to allocate memories anywhere in the system. (2) Once the developer scatters data around the system, the decision on the "executor" location is relegated to the runtime library. With this feature, Gecko provides the concept of 'moving code to data' to minimize data transfers within a system. (3) Gecko is highly user-friendly, dynamic, and flexible, and it responds to any changes in the underlying hardware, minimizing source code alteration whenever the architecture and the program requirements vary.

Gecko, due to its high-level features, is a potential candidate for 'X' in the 'MPI+X' programming model in the exascale era. *Gecko paves the way to utilize every level of the memory hierarchy in current architectural advancements of HPC systems with less code intervention.* The increasing trend in designing new memory hierarchies to tackle the memory

wall makes their utilization a challenging job. Application developers need a better approach for the upcoming future systems. Results of the experiments with Gecko reveals how it is a well-suited method for a multi-GPU platform as it delivers a portable and scalable solution primarily for benchmarks where the false sharing effect among devices is minimal.

To investigate the effectiveness of Gecko on real applications, we used the Rodinia benchmark and ported its applications to Gecko. The suite contains a number of benchmarks that have real applications in scientific domains. All the benchmarks represent different scientific domains that range from solving a system of linear algebra equations to traversing and parsing graphs in graph-based algorithms. With the help of Gecko, these applications were targeted to multiple architectures without any code modifications. We were able to achieve this with only one single codebase for each application.

7.2 Next Steps Looking Forward

Our model has the potential to be extended to support the rich functionalities of Processing-In-Memory (PIM) architectures. By extending the Gecko Runtime Library, applications will be able to easily utilize the PIM-enabled memory domains while little to no code modification is required by the main applications. Supporting other memory technologies, like Hybrid Memory Cube (HMC) [88], Non-volatile memory (NVM) [58], and Intel Persistent Memory [47, 26], can be added to Gecko so that a broad range of hardware is targeted.

Additionally, the feasibility of automatic data transfer between different locations can be explored in Gecko. Currently, allocated memory is not relocated automatically and an explicit request from the developer is required to move data among locations. However, explicit data movements will lead to inefficient performance since they are required to be performed every time. To address this shortcoming, Gecko has to support implicit data movements among locations. As a result, Gecko has to take the following criteria into

account when it needs to perform data movements: (1) the data size, (2) the bandwidth between source and destination locations, (3) the data size of other allocated memories targeted by that compute region, (4) predicted execution time of a compute region, and so on. This is a multi-objective optimization problem, which requires further investigation.

7.2.1 Big Picture

Enabling automatic data movement as discussed above will significantly improve the usability and portability of Gecko, especially with the current diversity in computer architecture and hardware design. An intelligent run time library will be able to decide which location would be the optimal location to keep a particular data item in steady state. As the application runs for a long period of time, an intelligent runtime library will observe different accesses among locations in the system and learn from those accesses. Building on top of such knowledge, future accesses will trigger the decision mechanism on whether the data or the kernel should be moved; in other words, which one would be more efficient to transfer, data or code?

7.2.2 Other Scientific Areas

Gecko also enables scientists from other domains to utilize their future high-end systems with a single codebase. For instance, scientists will be able to parallelize their model developments for weather prediction [33, 41] with Gecko. Astrophysical modeling will benefit from directive-based methods like Gecko to parallelize their adaptive mesh refinement kernels in their code [116]. The authors in [116] mention how portability is *the key design goal* of their development. Their goal was to target different platforms with the same kernel code. Aerospace engineers are also able to utilize Gecko to parallelize their applications. Gecko enables them to improve the performance of their computational fluid dynamics (CFD) kernels [68]. Our experiments show that Gecko is a suitable option to parallelize applications

and enable them to utilize different devices. The above-mentioned examples also show how Gecko's use cases are not limited to only the molecular dynamics use case and other scientific domains will benefit from it as well.

Bibliography

[1] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler. Page Placement Strategies for GPUs within Heterogeneous Memory Systems. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 607–618, New York, NY, USA, 2015. ACM.

[2] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. PIM-enabled Instructions. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 336–348, New York, NY, USA, 2015. ACM.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On Finding Lowest Common Ancestors in Trees. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pages 253–265, New York, NY, USA, 1973. ACM.

[4] A. Almgren, P. DeMar, J. Vetter, K. Riley, K. Antypas, D. Bard, R. Coffey, E. Dart, S. Dosanjh, R. Gerber, J. Hack, I. Monga, M. E. Papka, L. Rotman, T. Straatsma, J. Wells, D. E. Bernholdt, W. Bethel, G. Bosilca, F. Cappello, T. Gamblin, S. Habib, J. Hill, J. K. Hollingsworth, L. C. McInnes, K. Mohror, S. Moore, K. Moreland, R. Roser, S. Shende, G. Shipman, and S. Williams. Advanced Scientific Computing Research Exascale Requirements Review. An Office of Science review sponsored by Advanced Scientific Computing Research. , Technical Report, Argonne National Lab. (ANL), Argonne, IL, 2017.

[5] B. Alpern, L. Carter, and J. Ferrante. Modeling Parallel Computers as Memory Hierarchies. In *Proceedings of Workshop on Programming Models for Massively Parallel Computers*, pages 116–123, Berlin, Germany, 1993.

[6] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill, and Others. Exascale Software Study: Software Challenges in Extreme Scale Systems. , Technical Report, DARPA IPTO, Air Force Research Labs, 2009.

[7] 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. Abstract Machine Models and Proxy Architectures for Exascale Computing. In *Proceedings of Co-HPC 2014: 1st International Workshop on Hardware-Software Co-Design for High Performance*

Computing - Held in Conjunction with SC 2014: The International Conference for High Performance Computing, Networking, Storage and Analysis, pages 25–32, New Orleans, LA, 2014. IEEE.

[8] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 320–332, New York, NY, USA, 2017. ACM.

[9] K. Asadi, H. Ramshankar, H. Pullagurla, A. Bhandare, S. Shanbhag, P. Mehta, S. Kundu, K. Han, E. Lobaton, and T. Wu. Vision-based Integrated Mobile Robotic System for Real-time Applications in Construction. *Automation in Construction*, 96:470–482, 2018.

[10] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, pages 73–78, Estes Park, CO, 2002. IEEE.

[11] L. Benini and G. De Micheli. Networks on Chips: A New SoC Paradigm. *Computer*, 35(1):70–78, 2002.

[12] J. Beyer, D. Oehmke, and J. Sandoval. Transferring User-defined Types in OpenACC. In *Cray User Group*, CUG '14, 2014.

[13] S. Borkar and A. A. Chien. The Future of Microprocessors. *Communications of the ACM ACM*, 54(5):67–77, 2011.

[14] D. Brown, J. H. R. Clarke, M. Okuda, and T. Yamazaki. A Domain Decomposition Parallelization Strategy for Molecular Dynamics Simulations on Distributed Memory Machines. *Computer Physics Communications*, 74(1):67–80, 1993.

[15] H. Carter Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling Manycore Performance Portability Through Polymorphic Memory Access Patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization*, IISWC '09, pages 44–54. IEEE, 2009.

[17] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–11. IEEE, 2010.

[18] T. Chen, Z. Sura, and H. Sung. Automatic Copying of Pointer-Based Data Structures. In *Languages and Compilers for Parallel Computing*, pages 265–281, Rochester, NY, 2017. Springer International Publishing.

[19] P. Cicotti, S. M. Mniszewski, and L. Carrington. An Evaluation of Threaded Models for a Classical MD Proxy Application. In *Proceedings of Co-HPC 2014: 1st International Workshop on Hardware-Software Co-Design for High Performance Computing*, pages 41–48, New Orleans, LA, 2014. IEEE.

[20] Cloc. https://github.com/AlDanial/cloc. Accessed: 2018-04-10.

[21] CoMD Proxy Application. https://github.com/ECP-copa/CoMD. Accessed: 2018-04-02.

[22] COPA: Codesign Center for Particle Applications. Exascale Computing Project (ECP), 2018.

[23] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith. GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models. In *International Conference on High Performance Computing*, pages 489–507. Springer International Publishing, 2016.

[24] R. H. Dennard, F. H. Gaensslen, Y. U. Hwa-Nien, V. Leo Rideout, E. Bassous, and A. R. Leblanc. Design of Ion-Implanted MOSFETs with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 87(4):668–678, 1999.

[25] J. Dongarra and A. L. Lastovetsky. *High Performance Heterogeneous Computing*, volume 78 of *Wiley Series on Parallel and Distributed Computing*. John Wiley & Sons, 2009.

[26] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–15. ACM, 2014.

[27] D. Elias, R. Matias, M. Fernandes, and L. Borges. Experimental and Theoretical Analyses of Memory Allocation Algorithms. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1545–1546, New York, NY, USA, 2014. ACM.

[28] K. Fatahalian, W. J. Dally, P. Hanrahan, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, and A. Aiken. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[29] M. Feig, I. Yu, P. H. Wang, G. Nawrocki, and Y. Sugita. Crowding in Cellular Environments at an Atomistic Level from Computer Simulations. *Journal of Physical Chemistry B*, 121(34):8009–8025, 2017.

[30] T. B. Ferreira, R. Matias, A. Macedo, and L. B. Araujo. An Experimental Study on Memory Allocators in Multicore and Multithreaded Applications. In *Proceedings of Parallel and Distributed Computing, Applications and Technologies*, PDCAT '11, pages 92–98, Gwangju, South Korea, 2011.

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

[32] R. Friedman, K. Boye, and K. Flatmark. Molecular Modelling and Simulations in Cancer Research. *Biochimica et Biophysica Acta (BBA) - Reviews on Cancer*, 1836(1):1–14, 2013.

[33] O. Fuhrer, C. Osuna, and X. Lapillonne. Towards A Performance Portable, Architecture Agnostic Implementation Strategy for Weather and Climate Models. *Supercomputing Frontiers and Innovations*, 1(1):45–62, 2014.

[34] M. Ghane, M. Arjomand, and H. Sarbazi-Azad. An Opto-electrical NoC with Traffic Flow Prediction in Chip Multiprocessors. In *Proceedings of 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP '14, pages 440–443, Torino, Italy, 2014. IEEE.

[35] M. Ghane, S. Chandrasekaran, and M. S. Cheung. Assessing Performance Implications of Deep Copy Operations via Microbenchmarking. *arXiv preprint*, 2019.

[36] M. Ghane, S. Chandrasekaran, and M. S. Cheung. Gecko: Hierarchical Distributed View of Heterogeneous Shared Memory Architectures. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '19, pages 21–30, New York, NY, USA, 2019. ACM.

[37] M. Ghane, S. Chandrasekaran, and M. S. Cheung. Pointerchain: Tracing Pointers to Their Roots A Case Study in Molecular Dynamics Simulations. *Parallel Computing*, 85:190–203, 2019.

[38] M. Ghane, S. Chandrasekaran, R. Searles, M. Cheung, and O. Hernandez. Path Forward for Softwarization to Tackle Evolving Hardware. In *Proceedings of SPIE - The International Society for Optical Engineering*, SPIE '18, Orlando, Florida, 2018. SPIE.

[39] M. Ghane, J. Larkin, L. Shi, S. Chandrasekaran, and M. S. Cheung. Power and Energy-efficiency Roofline Model for GPUs. *arXiv preprint*, 2018.

[40] M. Ghane, A. M. Malik, B. Chapman, and A. Qawasmeh. False Sharing Detection in OpenMP Applications Using OMPT API. In *International Workshop on OpenMP*, IWOMP '15, pages 102–114, Aachen, Germany, 2015. Springer International Publishing.

[41] C. Gheller, P. Wang, F. Vazza, and R. Teyssier. Numerical Cosmology on the GPU with Enzo and Ramses. In *Journal of Physics: Conference Series*, volume 640, Boston, MA, 2015. IOP Publishing.

[42] G. Giupponi, M. J. Harvey, and G. De Fabritiis. The impact of accelerator processors for high-throughput molecular modeling and simulation. *Drug Discovery Today*, 13(23-24):1052–1058, 2008.

[43] J. Goodacre and A. N. Sloss. Parallelism and the ARM Instruction Set Architecture. *Computer*, 38(7):42–50, 2005.

[44] Google Benchmark. https://github.com/google/benchmark. Accessed: 2018-04-10.

[45] J. A. Herdman, W. P. Gaudin, O. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. Achieving Portability and Performance Through OpenACC. In *Proceedings of the 1st Workshop on Accelerator Programming Using Directives*, pages 19–26, New Orleans, LA, 2015.

[46] R. D. Hornung and J. A. Keasler. The RAJA Portability Layer: Overview and Status. , Technical Report, Lawrence Livermore National Laboratory (LLNL-TR-661403), 2014.

[47] Intel. https://software.intel.com/en-us/persistent-memory. Accessed: 2019-04-10.

[48] Intel Corp. *Intel® 64 and IA-32 Architectures Software Developer Manuals*.

[49] D. Jaggar. ARM Architecture and Systems. *IEEE Micro*, 17(04):9–11, 1997.

[50] JEDEC. High Bandwidth Memory (HBM) DRAM - http://www.jedec.org/standards-documents/results/jesd235. Accessed: 2019-04-10.

[51] J. E. Jones. On the Determination of Molecular Fields - II. From the Equation of State of a Gas. In *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, volume 106, pages 463–477, Trinity College, Cambridge, UK, 1924. The Royal Society.

[52] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still. Exploring Traditional and Emerging Parallel Programming Models Using A Proxy Application. In *Proceedings of International Parallel and Distributed Processing Symposium*, IPDPS '13, pages 919–932, Boston, MA, 2013. IEEE.

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

[54] Khronos OpenCL Working Group. *The OpenCL Specification 1.1*. Khronos Group, 2011.

[55] D. E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[56] C. Lameter. NUMA (Non-Uniform Memory Access): An Overview. *Queue*, 11(7):40, 2013.

[57] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt. An Investigation of Unified Memory Access Performance in CUDA. In *IEEE High Performance Extreme Computing Conference*, HPEC '14, pages 1–6, Waltham, MA, 2014.

[58] M. H. Lankhorst, B. W. Ketelaars, and R. A. Wolters. Low-cost and Nanoscale Non-volatile Memory Concept for Future Silicon Chips. *Nature Materials*, 4(4):347–352, 2005.

[59] A. Lastovetsky. Heterogeneity in Parallel and Distributed Computing. *Journal of Parallel and Distributed Computing*, 73(12):1523–1524, 2013.

[60] S. Lee and J. S. Vetter. Early Evaluation of Directive-based GPU Programming Models for Productive Exascale Computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 1–11, Salt Lake City, UT, 2012. IEEE.

[61] X. Liao, L. Xiao, C. Yang, and Y. Lu. MilkyWay-2 Supercomputer: System and Application. *Frontiers of Computer Science*, 8(3):345–356, 2014.

[62] Y. Lin, C. Chuang, C. Yen, S. Huang, P. Huang, J. Chen, and S. Lee. Artificial Intelligence of Things Wearable System for Cardiac Disease Detection. In *IEEE International Conference on Artificial Intelligence Circuits and Systems*, AICAS '19, pages 67–70, Hsinchu, Taiwan, 2019.

[63] E. Lindahl, B. Hess, and D. van der Spoel. A Package for Molecular Simulation and Trajectory Analysis. *Journal of Molecular Modeling*, 7(8):306–317, 2001.

[64] G. H. Loh. 3D-stacked Memory Architectures for Multi-core Processors. In *Proceedings of International Symposium on Computer Architecture*, ISCA '08, pages 453–464, Beijing, China, 2008.

[65] G. H. Loh, N. Jayasena, M. H. Oskin, M. Nutter, D. Roberts, M. Meswani, D. Ping, and Z. Mike. A Processing-in-Memory Taxonomy and a Case for Studying Fixed-function PIM. In *Proceeding of the Workshop on Near-Data Processing, in conjunction with MICRO*, WoNDP '13, pages 1–5, Davis, CA, 2013.

[66] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra, A. Geist, R. Haring, J. Hittinger, A. Hoisie, D. M. Klein, P. Kogge, R. Lethin, V. Sarkar, R. Schreiber, J. Shalf, T. Sterling, R. Stevens, J. Bashor, R. Brightwell, P. Coteus, E. Debenedictus, J. Hiller, K. H. Kim, H. Langston, R. M. Murphy, C. Webster, S. Wild, G. Grider, R. Ross, S. Leyffer, and J. Laros III. DOE Advanced Scientific Computing Advisory Subcommittee (ASCAC) Report: Top Ten Exascale Research Challenges. , Technical Report, USDOE Office of Science, United States, 2014.

[67] A. Luckow, K. Kennedy, M. Ziolkowski, E. Djerekarov, M. Cook, E. Duffy, M. Schleiss, B. Vorster, E. Weill, A. Kulshrestha, and M. C. Smith. Artificial Intelligence and Deep Learning Applications for Automotive Manufacturing. In *IEEE International Conference on Big Data*, Big Data '18, pages 3144–3152, Seattle, WA, 2018.

[68] L. Luo, J. R. Edwards, H. Luo, and F. Mueller. Advanced Optimizations of An Implicit Navier-Stokes Solver on GPGPU. In *53rd AIAA Aerospace Sciences Meeting*, Kissimmee, Florida, 2015.

[69] A. Marowka. Back to Thin-core Massively Parallel Processors. *Computer*, 44(12):49–54, 2011.

[70] A. Mishra, X. Dong, G. Sun, Y. Xie, N. Vijaykrishnan, and C. Das. Architecting On-chip Interconnects for Stacked 3D STT-RAM Caches in CMPs. In *Proceeding of the 38th annual international symposium on Computer architecture*, volume 39 of *ISCA '11*, pages 69–80, San Jose, CA, 2011.

[71] J. Mohd-Yusof and N. Sakharnykh. Optimizing CoMD: A Molecular Dynamics Proxy Application Study. In *GPU Technology Conference*, GTC '14, San Jose, CA, 2014.

[72] MPI Forum. MPI: A Message-passing Interface Standard. Version 2.2. Accessed: 2018-04-10.

[73] National Strategic Computing Initiative (NSCI). https://nsf.gov/cise/nsci/. Accessed: 2016-07-01.

[74] Nuwan Jayasena, Dong Ping Zhang, Amin Farmahini-Farahani and M. Ignatowski. Realizing the Full Potential of Heterogeneity through Processing in Memory. In *3rd Workshop on Near-Data Processing, in conjunction with MICRO*, WoNDP '15, Waikiki, Hawaii, 2015.

[75] NVidia. CUDA C Programming Guide. Accessed: 2019-08-18.

[76] NVidia. NVidia Autonomous Machines. Accessed: 2019-08-18.

[77] NVidia. NVidia Clara. Accessed: 2019-08-18.

[78] NVidia. NVidia DGX Systems. Accessed: 2019-08-18.

[79] NVidia. NVidia Jetson Systems. Accessed: 2019-08-18.

[80] NVidia PSG Cluster. http://psgcluster.nvidia.com/trac. Accessed: 2017-12-03.

[81] OpenACC Committee. Technical Report: Deep Copy Attach and Detach (TR-16-1). Accessed: 2017-12-03.

[82] OpenACC Committee. *OpenACC Application Programming Interface*, 2019.

[83] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*, 2019.

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

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

[86] S. Páll, M. J. Abraham, C. Kutzner, B. Hess, and E. Lindahl. Tackling Exascale Software Challenges in Molecular Dynamics Simulations with GROMACS. *Solving Software Challenges for Exascale*, 8759:3–27, 2015.

[87] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *Proceedings of the International Conference on Parallel Architectures and Compilation*, PACT '16, pages 31–44, Haifa, Israel, 2016.

[88] J. T. Pawlowski. Hybrid Memory Cube (HMC). In *IEEE Hot Chips 23 Symposium*, HCS '11, pages 1–24, Stanford, CA, 2011. IEEE.

[89] O. Pearce, H. Ahmed, R. W. Larsen, and D. F. Richards. Enabling Work Migration in CoMD to Study Dynamic Load Imbalance Solutions. In *Proceedings of the 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems*, PMBS '16, pages 98–107, Piscataway, NJ, USA, 2016. IEEE Press.

[90] D. A. Pearlman, D. A. Case, J. W. Caldwell, W. S. Ross, T. E. Cheatham III, S. DeBolt, D. Ferguson, G. Seibel, and P. Kollman. AMBER, A Package of Computer Programs for Applying Molecular Mechanics, Normal Mode Analysis, Molecular Dynamics and Free Energy Calculations to Simulate the Structural and Energetic Properties of Molecules. *Computer Physics Communications*, 91(1-3):1–41, 1995.

[91] J. R. Perilla, B. C. Goh, C. K. Cassidy, B. Liu, R. C. Bernardi, T. Rudack, H. Yu, Z. Wu, and K. Schulten. Molecular Dynamics Simulations of Large Macromolecular Complexes. *Current Opinion in Structural Biology*, 31:64–74, 2015.

[92] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten. Scalable Molecular Dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[93] S. Plimpton. Fast Parallel Algorithms for ShortRange Molecular Dynamics. *Journal of Computational Physics*, 117:1–42, 1995.

[94] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, Austin, TX, 2009.

[95] R. Reyes, I. López, J. J. Fumero, and F. de Sande. Directive-based Programming for GPUs: A Comparative Study. In *IEEE 14th International Conference on High Performance Computing and Communication and IEEE 9th International Conference on Embedded Software and Systems*, HPCC '09, pages 410–417, Liverpool, UK, 2012.

[96] K. Rupp. https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/. Accessed: 2019-02-27.

[97] Sabine Cluster. https://www.uh.edu/cacds/resources/hpc/sabine/. Accessed: 2019-02-20.

[98] R. R. Schaller. Moore's Law: Past, Present and Future. *IEEE Spectrum*, 34(6):52–59, 1997.

[99] A. Singharoy and C. Chipot. Methodology for the Simulation of Molecular Motors at Different Scales. *The Journal of Physical Chemistry B*, 121(15):3502–3514, 2017.

[100] J. E. Stone, D. J. Hardy, I. S. Ufimtsev, and K. Schulten. GPU-accelerated Molecular Modeling Coming of Age. *Journal of Molecular Graphics and Modelling*, 29(2):116–125, 2010.

[101] H. Sutter. The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[102] A. Tate, A. Kamil, A. Dubey, A. Größlinger, B. Chamberlain, B. Goglin, C. Edwards, C. J. Newburn, D. Padua, D. Unat, E. Jeannot, F. Hannig, T. Gysi, H. Ltaief, J. Sexton, J. Labarta, J. Shalf, K. Fürlinger, K. O'brien, L. Linardakis, M. Besta, M.-C. Sawley, M. Abraham, M. Bianco, M. Pericàs, N. Maruyama, P. H. J. Kelly, P. Messmer, R. B. Ross, R. Cledat, S. Matsuoka, T. Schulthess, T. Hoefler, and V. J. Leung. Programming Abstractions for Data Locality. In *Workshop in Programming Abstractions for Data Locality*, Lugano, Switzerland, 2014.

[103] Top500. https://www.top500.org. Accessed: 2018-04-10.

[104] 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. Pericas. Trends in Data Locality Abstractions for HPC Systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):1–1, 2017.

[105] L. Verlet. Computer "Expermiments" on Classical Fliuds I: Thermodynamical Properties of Lennard Jones Molecules. *Physical Review*, 159(1):98–103, 1967.

[106] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman, B. Van Essen, S. Yoo, A. Aiken, D. Bernholdt, S. Byna, K. Cameron, F. Cappello, B. Chapman, A. Chien, M. Hall, R. Hartman-Baker, Z. Lan, M. Lang, J. Leidel, S. Li, R. Lucas, J. Mellor-Crummey, P. Peltz Jr., T. Peterka, M. Strout, and J. Wilke. Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity. , Technical Report, USDOE Office of Science (SC), Washington, D.C., 2018.

[107] O. Villa, D. R. Johnson, M. O'Connor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S. W. Keckler, and W. J. Dally. Scaling the Power Wall: A Path to Exascale. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 830–841, New Orleans, LA, 2014. IEEE.

[108] S. Wienke, P. Springer, C. Terboven, and D. an Mey. OpenACC - First Experiences with Real-World Applications. In *Euro-Par 2012 Parallel Processing*, pages 859–870, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[109] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. Phase Change Memory. In *Proceedings of the IEEE*, volume 98, pages 2201–2227, 2010.

[110] D. H. Woo, N. H. Seong, D. L. Lewis, and H.-H. S. Lee. An Optimized 3D-stacked Memory Architecture by Exploiting Excessive, High-density TSV Bandwidth. In *The Sixteenth International Symposium on High-Performance Computer Architecture*, HPCA '10, pages 1–12, Bangalore, India, 2010.

[111] Y. Yan, R. Brightwell, and X.-H. Sun. Principles of Memory-Centric Programming for High Performance Computing. In *Proceedings of the Workshop on Memory Centric Programming for HPC*, MCHPC '17, pages 2–6, New York, NY, USA, 2017. ACM.

[112] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *Lecture Notes in Computer Science*, pages 172–187, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[113] Z. Yang, F. Gao, and S. Shen. Real-time Monocular Dense Mapping on Aerial Robots Using Visual-inertial Fusion. In *IEEE International Conference on Robotics and Automation*, ICRA '17, pages 4552–4559, Singapore, Singapore, 2017.

[114] H. Zhang, G. Hou, M. Lu, J. Ahn, I. J. L. Byeon, C. J. Langmead, J. R. Perilla, I. Hung, P. L. Gor'Kov, Z. Gan, W. W. Brey, D. A. Case, K. Schulten, A. M. Gronenborn, and T. Polenova. HIV-1 Capsid Function Is Regulated by Dynamics: Quantitative Atomic-Resolution Insights by Integrating Magic-Angle-Spinning NMR, QM/MM, and MD. *Journal of the American Chemical Society*, 138(42):14066–14075, 2016.

[115] H. Zhao and A. Caflisch. Molecular Dynamics in Drug Design. *European Journal of Medicinal Chemistry*, 91:4–14, 2014.

[116] M. Zingale, A. S. Almgren, M. G. B. Sazo, V. E. Beckner, J. B. Bell, B. Friesen, A. M. Jacobs, M. P. Katz, C. M. Malone, A. J. Nonaka, D. E. Willcox, and W. Zhang. Meeting the Challenges of Modeling Astrophysical Thermonuclear Explosions: Castro, Maestro, and the AMReX Astrophysics Suite. *Journal of Physics: Conference Series*, 1031:12024, 2018.

Appendix A

Gecko's Directives

This appendix describes the Gecko's programming model in detail. All directives in Gecko and its clauses are described below. All keywords mentioned in this Appendix are mandatory unless they are put inside `[ ]`. In that case, they are considered **optional**. The most up-to-date version of this manual is accessible on its associated Github page[1].

A.1 Location Type

The first step in using Gecko is to declare all the location types at the beginning of the application. This is achieved with the `loctype` construct. Below we show the syntax of `loctype`.

```
1  #pragma gecko loctype name(char*) kind(char*, char*)
```

It accepts the following clauses. The type of the input to each clause is specified in the parentheses.

[1] `https://github.com/milladgit/gecko`.

- `name`: the user-defined name for our type (`char*`).
- `kind`: the system-defined type for the location type (`char* , char*`).

Note: The "`virtual`" location name is reserved for the **virtual** location type.

Note 2: For a list of supported memory kinds, please see *the following example.*

Example: The first line in the following code snippet declares that our application needs a conventional processor with 4 cores and of Intel's Skylake type. The second line declares a minimum NVidia GPU with Compute Capability of 5.0 ('cc50') and calls it 'tesla' for future reference. The third line specifies the main memory module in our system with 16 GB in size. The last line declares the location type for permanent storage. In the current implementation, it is a file on the file system tree.

```
1 #pragma gecko loctype name("host") kind("x64", "Skylake") num_cores(4) mem("4MB")
2 #pragma gecko loctype name("tesla") kind("NVIDIA", "cc50") mem("4GB")
3 #pragma gecko loctype name("NODE_MEMORY") kind("Unified_Memory") size("16GB")
4 #pragma gecko loctype name("HDD") kind("Permanent_Storage")
```

A.2 Location

Locations in Gecko are defined using the `location` construct as shown below. An example of its syntax is shown below.

```
1 #pragma gecko location name(char*) type(char*) [all]
```

It accepts the following clauses. The type of the input to each clause is specified in the parentheses.

- `name:` the name of the location (`char*`).

- `type:` the type of the location (`char*`) from the declared ones by `loctype`.
- `all:` defining all devices with similar location type under one umbrella. This will result in the following naming conversion for the devices: `<name>[i]` where `<name>` is the name of the location and `[i]` provides a way to distinguish the locations.

Example: The following lines define the locations used in the model shown above. The code snippet below declares `LocA` as our main memory location. Virtual locations in this snippet are `LocB` and `LocC`. Other locations in the snippet are from the `host` and `tesla` type, which were defined previously with the `loctype` clause in the last code snippet. The fourth line of the code snippet shows how we are declaring four GPUs in the system. However, if the number of available devices from that type are unknown at the time of writing the code, one can ask the runtime library to utilize all available devices at the execution time. One can enable this feature by using the `all` clause (as shown in Line 5).

```
1 #pragma gecko location name("LocA") type("NODE_MEMORY")
2 #pragma gecko location name("LocB","LocC") type("virtual")
3 #pragma gecko location name("LocN1", "LocN2") type("host")
4 #pragma gecko location name("LocG1", "LocG2", "LocG3", "LocG4") type("tesla")
5 #pragma gecko location name("LocG") type("tesla") all
```

A.3 Hierarchy

The hierarchy in Gecko determines the relationship among the locations with respect to each other. The relationship between locations is declared with the `hierarchy` clause. Every location in Gecko has a parent and a number of children. An example of its syntax is shown below.

```
1 #pragma gecko hierarchy children(<op> : <list>) parent(char*) [all]
```

It accepts the following clauses. The type of the input to each clause is specified in the parentheses.

- `parent:` the parent location in the relationship (`char*`).
- `children:` the list of all locations to be the children of `parent`. The keyword accepts an operation and the children list: `(<op> : <list>)`. The `op` can be `+` and `-` signs or a `char` variable that is either `+` or `-`. The `<list>` is the comma-separated list of defined locations (name of the locations in `char*`).
- `all:` similar to the `all` keyword in the `hierarchy`, this keyword is used to include `all` locations under this hierarchy.

Example: The lines below shows how to use the hierarchy construct. Lines 2-3 add and remove children to locations *LocA* and *LocB*, respectively. The statement in Line 4 introduces the *LocG**i*** locations as the children of *LocC*. One can reverse the operation at run time by changing the value of the `op` variable. Please note how `+` and `'+'` are equivalent.

```
1 char op = '+';
2 #pragma gecko hierarchy children(+:"LocB","LocC") parent("LocA")
3 #pragma gecko hierarchy children('-':"LocN1","LocN2") parent("LocB")
4 #pragma gecko hierarchy children(op:"LocG1","LocG2","LocG3","LocG4") parent("LocC")
```

A.4 Configuration File

The whole structure of the hierarchy tree can be stored within a configuration file. Gecko can load such a file and populate the tree automatically. This brings a great degree of flexibility to application developers and makes the application extremely portable. An example of its syntax is shown below.

- `file:` the configuration file name.
- `env:` the `GECKO_CONFIG_FILE` environment variable contains the path to the file. Please refer to the section describing the environmental variables below.

It accepts the following clauses. The type of the input to each clause is specified in the parentheses.

```
1 #pragma gecko config env
2 #pragma gecko config file("/path/to/config/file")
```

Note: The `file` and `env` cannot be chosen simultaneously.

Example: An example of a configuration file for above-mentioned hierarchy tree is shown below:

An example of a configuration file in Gecko

```
loctype;kind,x64,Skylake;num_cores,4;mem,4MB;name,host;

loctype;name,tesla;kind,CC7.0,Volta;mem,4GB

loctype;name,NODE_MEMORY;kind,Unified_Memory;size,16GB


location;name,LocA;type,NODE_MEMORY;

location;name,LocB,LocC;type,virtual

location;name,LocN1,LocN2;type,host;

location;name,LocG1,LocG2,LocG3,LocG4;type,tesla


hierarchy;children,+,LocB,LocC;parent,LocA

hierarchy;children,+,LocN1,LocN2;parent,LocB

hierarchy;children,+,LocG1,LocG2,LocG3,LocG4;parent,LocC
```

A.5 Drawing

For convenience, Gecko can generate the hierarchical tree for visualization purposes. Using the `draw` construct, at any point in executing the program, the runtime library will generate a compatible DOT file. One can convert a DOT file to a PDF file using the dot command: `dot -Tpdf gecko.conf -o gecko-tree.pdf`

```
1 #pragma gecko draw filename(char*)
```

It accepts the following clauses. The type of the input to each clause is specified in the parentheses.

- `filename:` the target DOT file name (`char*`). It can be an absolute or relative path. The default value for this keyword is `"gecko.dot"`.

Example:

```
1 #pragma gecko draw filename("/path/to/DOT/file")
```

A.6 Memory Operations

A.6.1 Allocating/Freeing Memory

Memory operations in Gecko are supported by the `memory` construct. To allocate memory, use the `allocate` keyword and to free the object, use `free`. Optional features are specified inside brackets (`[ ]`).

It accepts the following clauses. The type of the input to each clause is specified in the parentheses.

```
1 #pragma gecko memory allocate(<ptr>[0:<count>]) type(<datatype>) location(char*) [
      distance(<dist>) [realloc/auto]] [file(char*)]
2 #pragma gecko free(<ptr_list>)
```

- `allocate(<ptr>[0:<count>])`: the input to the `allocate` keyword accepts a pointer (`<ptr>`) and its number of elements (`<count>`). `<count>` can be a constant or a variable. *Please see the example below.*

- `datatype`: the data type of the `ptr` variable.

- `<ptr_list>`: the comma-separated list of allocated variables with the `allocate` construct.

- `<dist>`: specifies the distance of the allocation in distance-based allocations. For these types of allocations, the allocation is performed when the destination location to execute the region is chosen. As a result, the allocation is postponed until the region is ready to be executed. It accepts the following values:

  - `near`: the allocation is performed in the chosen execution location.

  - `far[:<n>]`: the allocation is performed in the `n`-th grandparent with respect to the chosen execution location. For `n==0` and `n==1`, the immediate parent is chosen. In cases that `n` causes the location to be chosen to go further than the root location, the root location is chosen.

- `<realloc/auto>`: the policy to perform the allocation.

  - `realloc`: the allocated memory is freed after the associated region is finished.

  - `auto`: the allocated memory is not freed after the region is finished and it is moved around the hierarchy as needed. The allocated memory can be used with other regions of the application.

- `file:` the file name in case the location type is `Permanent_Storage`. It is the path to a file in the file system.

Example:

```
1 int N = 2000;
2 // place-holders for our arrays
3 double *X, *Y, *Z, *W;
4 #pragma gecko memory allocate(X[0:N]) type(double) location("LocA")
5 #pragma gecko memory allocate(Y[0:N]) type(double) location("LocB")
6 #pragma gecko memory allocate(Z[0:N]) type(double) location("LocC")
7 #pragma gecko memory allocate(W[0:N]) type(double) location("LocG1")
8 //...<some computation>...
9 #pragma gecko memory free(X, Y, Z, W)
```

Note: Please refer to the `region` section to see an example of distance-based allocations.

A.6.2 Copy and Move

One can copy memory between two different memory locations. It is similar to the regular `memcpy` operations; however, it is performed between different platforms and architectures.

```
1 // Copying elements from src[s1, e1] to dst[s2, e2]
2 #pragma gecko memory copy from(src[s1:e1]) to(dst[s2:e2])
```

It accepts the following clauses. The type of the input to each clause is specified in the parentheses.

- `src:` the source memory location to perform the copy operation from index *s1* to *e1*.
- `dst:` the destination memory location to perform the copy operation from index *s2* to *e2*.

Example:

```
1 #pragma gecko memory copy from(X[0:N]) to(Y[0:N])
```

In some cases, we have to move a set of data elements from Location P to Location Q. In such cases, the source location no longer possesses the variable and the destination location has to own the variable.

```
1 #pragma gecko memory move(<var>) to(char*)
```

It accepts the following clauses. The type of the input to each clause is specified in the parentheses.

- `var:` the source memory location to perform the move operation.
- `to:` the destination location for the move operation.

Example:

```
1 // Moving the Q array from its current location
2 // to LocA
3 #pragma gecko memory move(Q) to("LocA")
```

A.6.3 Register/Unregister

There are many cases where we are dealing with variables that were allocated before and we want to use them with Gecko. With the `register/unregister` clauses one can introduce them properly to Gecko.

It accepts the following clauses. The type of the input to each clause is specified in the parentheses.

- `<var>:` the already allocated memory.

```
1 #pragma gecko memory register(<var>[<start>:<end>]) type(<type>) loc(char*)
2 #pragma gecko memory unregister(<var>)
```

- `<type>:` type of the memory.
- `loc:` the name of the proper location that this variable originated from.

Note: Developers are responsible to free the registered arrays with Gecko. Gecko does not free them automatically.

Example: Line 3 registers the already allocated memory space by `vector` class to Gecko in `LocN`. We assume that `LocN` is a host location since the vector class allocates its memories in the host memory. Lines 5-6 show how we used `register` to register GPU-allocated memories. Lines 8-9 show the `unregister` operations for both the host and device locations.

```
1 vector<double> v(100);
2 double *v_addr = (double*) v.data();
3 #pragma gecko memory register(v_addr[0:100]) type(double) loc("LocN")
4 double *d_addr;
5 cudaMalloc((void**) &d_addr, sizeof(double) * 100);
6 #pragma gecko memory register(d_addr[0:100]) type(double) loc("LocG1")
7 // ...
8 #pragma gecko memory unregister(d_addr)
9 #pragma gecko memory unregister(v_addr)
```

A.7 Region

Gecko recognizes the computational kernels with the `region` construct. The end of the region is recognized with the `end` keyword.

It accepts the following clauses. The type of the input to each clause is specified in the parentheses.

- `datatype:` the execution policy to execute the kernel. *Please refer to execution policy section for more details.*

```
#pragma gecko region exec_pol(char*) variable_list(<ptr_list>) \\
    [gang[(<gang_count>)]] [vector[(<vector_count>)]] [independent] \\
    [reduction(<op>:<var>)] [at(char*)]
for(...) {
  /* some computations */
}
#pragma gecko region end
```

- `<ptr_list>`: list of utilized variables within the region.
- `gang, vector, independent, reduction`: please refer to the OpenACC specification to learn more about these keywords.
  - **Note:** Gecko relies on OpenACC to generate code for different architectures.
- `at`: *[optional]* the destination location to execute the code.
  - **Note:** In the new version of Gecko, the destination location is chosen based on the variables used in the region (`<ptr_list>`). However, the developer can override Gecko and specify where to execute the code.

Example: The following example shows how to multiply every elements of the `X` array by `coeff` and store the results in the `Z` array for all the indices between `a` and `b`. The `static` execution policy is chosen and the location that executes the code is chosen with respect to the arrays used in this kernel: `X` and `Z`.

```
double coeff = 3.4;
int a = 0;
int b = N;
#pragma gecko region exec_pol("static") variable_list(Z,X)
for (int i = a; i<b; i++) {
  Z[i] = X[i] * coeff;
}
#pragma gecko region end
```

Example of distance-based allocations: Listing A.1 shows an example of distance-based allocations. Arrays `T1, T1_realloc, T1_auto, T2, T2_far2, T2_far_ variable`,

and `Perm` are distance-based arrays, whose allocation location is determined at run time. For this scenario, their location depends on the location of variables `X` and `Z` since these variables determine the execution location of the kernel at Line 15.

Listing A.1: An example of utilizing distance-based allocation in Gecko.

```
1  double *T1, *T1_realloc, *T1_auto, *T2, *T2_far2, *T2_far_variable;
2  #pragma gecko memory allocate(T1[0:N]) type(double) distance(near)
3  #pragma gecko memory allocate(T1_realloc[0:N]) type(double) distance(near) realloc
4  #pragma gecko memory allocate(T2[0:N]) type(double) distance(far) file("T2.obj")
5  #pragma gecko memory allocate(T2_far2[0:N]) type(double) distance(far:2) file("T2_far
       .obj")
6  int far_distance = 10;
7  #pragma gecko memory allocate(T2_far_variable[0:N]) type(double) distance(far:
       far_distance) file("T2_far_variable.obj")
8
9  double *Perm;
10 #pragma gecko memory allocate(Perm[0:N]) type(double) location("LocHDD") file("perm.
       obj")
11
12 a = 0;
13 b = N;
14 long total = 0;
15 #pragma gecko region exec_pol("static") variable_list(Perm,Z,X,T1,T2_far_variable)
       reduction(+:total)
16 for (int i = a; i<b; i++) {
17   Z[i] = X[i] * coeff;
18   T1[i] *= 2;
19   T2_far_variable[i] *= 2;
20   total += (i+1);
21   Perm[i] = i;
22 }
23 #pragma gecko region end
```

A.8 Synchronization Point

By default, regions in Gecko are executed asynchronously. Synchronization points in Gecko are expressed with the `pause` construct. The granularity at which the synchronization happens can be controlled is specified with the `at` keyword. The location is an optional input. If the location is not specified, Gecko waits for all resources to finish their work.

```
1  #pragma gecko region pause [at(char*)]
```

It accepts the following clauses. The type of the input to each clause is specified in the parentheses.

- `at`: the location to wait on

Example: The following line ensures that the application will wait on all the computational resources for the child of `LocA` to finish its assigned job.

```
1  #pragma gecko region pause at("LocA")
```

Appendix B

Funding and Source Code

B.1 Funding

This material is based upon work supported by the National Science Foundation (NSF) Grant numbers 1531814 (NSF MRI[1]) and 1412532, and the Department of Energy (DOE) Grant No. DE-SC0016501. This research was also supported in part by the Lilly Endowment, Inc., through its support for the Indiana University Pervasive Technology Institute, and in part by the Indiana METACyt Initiative. We are also very grateful to NVIDIA for providing us access to its PSG cluster and thankful to the OpenACC technical team, especially Mat Colgrove, Pat Brooks, and Michael Wolfe.

B.2 Source Code

All source code is available online at the following URLs:

`pointerchain`:

[1] `https://uhpc-mri.uh.edu/`

`https://github.com/milladgit/pointerchain`

Deep copy microbenchmark:

`https://github.com/milladgit/deepcopy-benchmark`

Gecko:

`https://github.com/milladgit/gecko`

The STREAM benchmark ported to Gecko:

`https://github.com/milladgit/Gecko-BabelStream`

The Rodinia Benchmark Suite in Gecko:

`https://github.com/milladgit/gecko-rodinia`