2016

Scheduling and Tuning Kernels for High-performance on Heterogeneous Processor Systems

Ye Fang
Louisiana State University and Agricultural and Mechanical College

SCHEDULING AND TUNING KERNELS FOR HIGH-PERFORMANCE ON HETEROGENEOUS PROCESSOR SYSTEMS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The School of Electrical Engineering and Computer Science

by
Ye Fang
M.S., Louisiana State University, 2016
B.S., Shandong University at Weihai, 2009
May 2017

Acknowledgments

My first and foremost acknowledgment is dedicated to my advisor Prof. J. (Ram) Ramanujam. Ram has always been a great example of a wise and knowledgeable computer scientist. From him, I learned how to elegantly extract models from the real word and resolve them use mathematical tools. Ram provided a great amount of freedom in my research, and encouraged me to access many aspects of the computing community.

I would like to send a big thanks to my co-advisor Prof. Michal Brylinski. Being the science leader of a bioinformatics high-performance computing project, he gave intense advices on the domain knowledge and taught me the workflows of academic writing. I also want to thank Prof. Mark Jarrel and Juana Moreno, without whom, the Louisiana Alliance for Simulation-Guided Materials Applications (LA-SiGMA) would nots be established. Most of my funding and collaborations should contribute to LA-SiGMA.

Throughout my study at LSU, Dr. Ka-Ming Tam has been a great tutor. He introduced me to numeral computing techniques, which leads to my first project on parallel tempering Monte Carlo simulations for spin glass. I would like to thank Prof. David Koppelman, Prof. Bin Chen, Dr. Zhifeng Yun, Dr. Wei Feinstein and Dr. Jian Tao, for direct instructions on my projects and research. I also sincerely appreciate the help from my closely collaborated colleagues, including Sameer Abu Asal, Mohammad Rastegar Tohid, Yun Ding, Sheng Feng, and Troy Loeffler. Meanwhile, my academic life is incomplete without these friends: Sahar Navaz, Zahra Khatami, Shaoming Chen, Sam Irving, Mengxi Wu, Shayan Shams, etc. I would remain grateful for all their supports.

Many thanks have to be given to my thesis defense committee. Their time, patience, and intellectual contributions have inspired my researches and thesis works. To Prof Michal Brylinski who read through my dissertation proposal in the first place, helped figured out a better way of ranging the sections. The suggestion of using scatter plot instead histogram helped me better understand different visualization techniques. To Prof. Xin Li who suggested the possibility of power capping, defines one of the research topics in

this thesis. To Prof. Gerald Baumgartner, who raised a question regarding the generality of my scheduling algorithms. It triggered me to study distributed resource management systems and their applications. To Prof. Kerry Dooley who kindly shared his wisdom of how end-users are expecting from high-performance computing community.

My parents, Qi Fang and Hong Lei, receive my deepest gratitude and love for their dedication and many years of support. Thank you!

TABLE OF CONTENTS

List of Tables

List of Figures

Abstract

Accelerated parallel computing techniques using devices such as GPUs and Xeon Phis (along with CPUs) have proposed promising solutions of extending the cutting edge of high-performance computer systems. A significant performance improvement can be achieved when suitable workloads are handled by the accelerator. Traditional CPUs can handle those workloads not well suited for accelerators. Combination of multiple types of processors in a single computer system is referred to as a heterogeneous system.

This dissertation addresses tuning and scheduling issues in heterogeneous systems. The first section presents work on tuning scientific workloads on three different types of processors: multi-core CPU, Xeon Phi massively parallel processor, and NVIDIA GPU; common tuning methods and platform-specific tuning techniques are presented. Then, analysis is done to demonstrate the performance characteristics of the heterogeneous system on different input data. This section of the dissertation is part of the GeauxDock project, which prototyped a few state-of-art bioinformatics algorithms, and delivered a fast molecular docking program.

The second section of this work studies the performance model of the GeauxDock computing kernel. Specifically, the work presents an extraction of features from the input data set and the target systems, and then uses various regression models to calculate the perspective computation time. This helps understand why a certain processor is faster for certain sets of tasks. It also provides the essential information for scheduling on heterogeneous systems.

In addition, this dissertation investigates a high-level task scheduling framework for heterogeneous processor systems in which, the pros and cons of using different heterogeneous processors can complement each other. Thus a higher performance can be achieve on heterogeneous computing systems. A new scheduling algorithm with four innovations is presented: Ranked Opportunistic Balancing (ROB), Multi-subject Ranking (MR), Multi-subject Relative Ranking (MRR), and Automatic Small Tasks Rearranging (ASTR). The new algorithm

consistently outperforms previously proposed algorithms with better scheduling results, lower computational complexity, and more consistent results over a range of performance prediction errors.

Finally, this work extends the heterogeneous task scheduling algorithm to handle power capping feature. It demonstrates that a power-aware scheduler significantly improves the power efficiencies and saves the energy consumption. This suggests that, in addition to performance benefits, heterogeneous systems may have certain advantages on overall power efficiency.

Chapter 1
Introduction

1.1 THE LANDSCAPE OF HETEROGENEOUS COMPUTING

The increasing complexities of scientific models and big data applications demand extreme scale of computing power. Parallel high-performance computers (HPC) currently handle these requirements. The TOP500 list is a list released twice a year, which details the 500 most powerful computers in the world. For example, see the 46th HPC TOP500 list released in November 2015 [1, 2]. In that list, the top spot was taken by Tianhe-2 cluster supercomputer; Tianhe-2 has 16,000 compute nodes where each node has 2 "Ivy bridge-EP" Xeon multi-core CPUs as well as 3 "Knights Corner" Xeon Phi co-processors. Tianhe-2 achieves a performance of 33.8 petaFLOPS on the HPL benchmark; but Tianhe-2 uses a lot of energy. As much as 24 megawatts (MW) power (with cooling) is consumed by the whole system. Actually, the number one spot on the TOP500 list has been held by Tianhe-2 for six consecutive times.

The research and practice of building the powerful HPCs faces three main challenges. These challenges come from three directions, which are commonly called three "walls."

The first wall that blocks the performance progress is the single-core performance ceiling that emerged a decade ago [3]. The researchers can hardly implement higher clock rate through advancing circuit technologies, and have difficulty extracting more performance per clock cycle. This problem was addressed by the advent of the chip multiprocessor (CMP) or multi-core CPUs techniques. The cores of the processor are not designed to grow faster, but to replicate and grow in number. In addition to programmability challenges, contemporary multi-core CPUs suffer a burden from power constraints.

The second wall is right ahead of the computing research community. The dark silicon

model [4] showed that regardless of chip organization and topology, multi-core scaling is increasingly power-limited. "Even at 22nm, 21% of a chip must be powered off, and at 8nm, this number grows to more than 50%" [4]. Given this scenario, more power-efficient architectures are badly needed.

The desire for power optimization also comes from the economics of energy usage, which is the third wall. Tianhe-2 computer consumes 24 MW power in total, and each megawatt watt of electric power cost approximately 1 million U.S dollar per year [5]. The budget to pay the electric bill is right at the boundary of even the most demanding facilities are willing to afford. The HPC community has long been planning for an exascale computer (1 exaFLOPS = 1000 petaFLOPS) at 30 megawatts power budget. This means the new supercomputer must improve its power efficiency by at least 15 times over the current generation. To achieve this goal, heterogeneous systems with CPUs and accelerators play a major role.

In principle, a heterogeneous accelerator means an attached computing device on top of the traditional CPU and RAM (denoted as host). Unlike a homogeneous system, such as the Symmetric multiprocessing (SMP) or Massively Parallel Processor Array (MPPA), the architectures of accelerators are different from that of the host processor. This implies the programmer cannot freely partition workloads on heterogeneous computers without any performance consequences. nfortunately, no architecture is ultimately efficient for all workloads. To program an accelerated heterogeneous computer system, the programmer must strive to design the optimal partition of the program to maximize the suitability, and at the same time minimize the overheads from data transfers and synchronization. The process of moving a workload from host to accelerator is usually called offloading.

There are two types of heterogeneous accelerators. One is special-purpose hardware. Application-specific integrated circuits (ASICs) are commonly deployed in network and encryption applications, and recently have demonstrated successful experience in accelerating neural network modeling [6]. Field-programmable gate arrays (FPGAs) are a lower cost

alternative to ASICs, and have recently been utilized for machine learning [7] and image processing applications [8]. There is on-going research on quantum computing devices [9] that could also be considered as a special kind of heterogeneous computing system. Another type of heterogeneous accelerator is built on the concept of massively parallel processing. In contrast to traditional CPU architectures designed to minimize the execution latency on serial codes, this kind of accelerator features massive amount of highly simplified cores, and are generally optimized for high-throughput computations. Therefore, their performance on latency-sensitive applications is often poor. Consequently, the common programming strategy is to leave control-intensive irregular code on the CPU, and offload highly parallel highly regular computations onto these devices for acceleration. This genre of massive parallel accelerator is currently more mature and widely adopted in HPC. In this work, I will exclusive discuss this type of accelerators. Section 2.1 gives background information on their architectures and programming models.

1.2 CHALLENGE I: PERFORMANCE TUNING

The architectures of heterogeneous accelerators require extensive performance tuning, where the combination of tuning techniques remain highly application-specific and is sensitive to input data. In the first part of my dissertation, I describe my efforts of tuning multi-core CPUs, Xeon Phi and NVIDIA GPUs, using the newly developed *GeauxDock* modular docking software as a case study. I'll present the performance tuning methodologies and algorithms, as well as a detailed analysis of the performance characteristics.

1.3 CHALLENGE II: PERFORMANCE MODELING AND PREDICTION

In the second part of this dissertation, I'll be resolving the performance prediction problem. My work on this leverages machine learning models and the architectures of heterogeneous computers. It helped better understand why a certain task would be favored

on a certain processor.

1.4 CHALLENGE III: SCHEDULING FOR HETEROGENEOUS SYSTEMS

Thirdly, programming for contemporary heterogeneous computers relies on the intuition of programmers for code decompositions. Specifically, programmers need to explicitly construct the partition of the code as well the partition of the data. A few research projects try to automate this process. However, some [10, 11, 12, 13, 14] ignore performance heterogeneity, simply partitioning the data into various sized chunks. Others [15, 16] select only the most suitable heterogeneous processor for work and let the other processors idle all the time. None of them achieve system-wise optimization. My research is based on the view that in order to fully unleash the computational capabilities and maximize efficiency, a good system must consider the heterogeneity of both possessor characteristics and workload characteristics. Meanwhile, activating more processors is always better if power is not a constraint. Accordingly, I'll be resolving the following problem: Let there be many independent computational tasks and many heterogeneous processors. Assuming each task could run on any processor, how does one minimize the overall computation wall time? In section 4, I propose a scheduling algorithm to optimize the computational throughput. Additionally, in section 5, I study power and energy as additional metrics. The scheduling algorithm is also extended to optimize for power efficiency.

Chapter 2
Performance Tuning For Heterogeneous Processors

2.1 PROGRAMMING HETEROGENEOUS COMPUTERS

A CPU core is a fully functional module that could execute one instruction streams at a time. Traditional CPU has one core, whereas, multi-core CPUs systematically integrate more than one cores and therefore simultaneously support multiple instruction streams. Modern CPU cores deploy both Single Instruction Single Data (SISD) [17] computation model and Single Instruction Multiple Data (SIMD) [17] computation mode. Thus a core is able to operate on scalar data or vector data. Intel "Ivy Bridge" Xeon E5 2680 v2 is an example of modern multi-core CPU. It has 10 cores, each core features two 256bit wide SIMD unites. On every cycle, it could compute one 256 bit AVX addition instruction coupled with one 256 bit AVX multiplication instruction.

GPUs evolve from the dedicated hardware to accelerate graphics processing Application Programming Interfaces (APIs) like OpenGL [18] and DirectX [19]. GPUs use many vector operations and offer up to hundreds of times more raw computation power than contemporary CPUs. The result is that GPU has been redirected from graphics processing to general-purpose computations. The community use the terminology "GPGPU computing" to describe this kind of practice in the early day when re-targeting graphics oriented APIs. In the current era, numerous languages (such as CUDA, OpenCL), libraries and tools are specifically built for this purpose; as a result, the term "GPU programming" is used to describe this activity.

Many Integrated Core (MIC) is the initial result of Intel's effort in pushing its x86 architecture for graphics computing. Their effort later adapted towards the HPC market,

with the feature set quite similar to traditional CPUs. The Intel Xeon Phi is the band name of Intel's MIC offering.

Heterogeneous computing is a major player in today's HPC community, and is currently dominated by two vendors: NVIDIA GPU and Intel Xeon Phi. These two offerings share some common features, but also have unique characteristics. With respect to hardware, both accelerators as well as contemporary multi-core CPUs share a two-level parallel architecture principle. The coarse-grained outer level constructs a computation cluster whose processing elements provide the fine-grained inner level of parallelism. For example, from the perspective of GPU computing, the GPU can be viewed as a cluster of vector processors. Each GPU contains an array of Streaming Multiprocessors (SMs), each of which consists of many Scalar Processors (SPs), transcendental function units, registers, and fast on-chip memory. I can closely match GPU's hierarchical architecture model to that of the multi-core CPU. Just as the core is the building block unit of a CPU, a SM is a building block of a GPU. Similar to the lanes in a CPU core's SIMD unit, SPs are SM's vector lanes.

With regard to software, each coarse-grained cluster handles its own programming context known as a thread on CPU and Xeon Phi, and a thread block defined by the GPU Compute Unified Device Architecture (CUDA) [20] paradigm. On the CPU and Xeon Phi, the inner level exposes data parallelism, viz. SIMD operations. NVIDIA GPU uses CUDA threads inheriting a similar principle of vector processing. For instance, a bundle of 32 consecutive CUDA threads, denoted as a warp, are scheduled together. Consequently, CUDA threads may go to predication when a small, conditionally protected piece of code is encountered, forcing the execution of all instructions. When different CUDA threads take different paths in multiple-path branches, more cycles are consumed leading to a lower device utilization. Although SIMD instructions on CPU and Xeon Phi have similar characteristics, the vector width is about one-quarter to one-half of that on GPU and the code generation heuristic can vary significantly. Therefore, irregular codes may perform dramatically differently on these platforms. Another major difference between CPU and Xeon

Phi, and GPU is that the former implement hardware multi-threading at the outer level, whereas multi-threading on GPU is at the inner level demanding more data parallelism. Comparing Xeon Phi with CPU, it delivers roughly equal amount of raw compute power per core in terms of the number of data operations per cycle. However, because of a larger number of computing cores, Xeon Phi offers certain advantages over CPU in processing regular, highly parallel workloads. On the other hand, CPU cores typically perform better for irregular workloads.

In addition, the compute performance is also affected by memory operations. CPU and Xeon Phi heavily rely on caches that enforce coherence, and are easy to program. Since cache implementations are costly, they are hard to scale and can lead to low performance. On the other hand, GPUs expose their fast on-chip memory to programmers, known as the CUDA shared memory. This design makes life harder for programmers. But if done right, it could be highly efficient.

Parallel programming models fall into two broad categories: 1) small groups of tightly coupled processors sharing a common memory space, and 2) large, scalable systems that do not share a common memory. Both models often coexist in a high-performance computing (HPC) environment; for instance, many HPC systems use the distributed memory model to scale up to thousands of multi-processor nodes, each employing the shared memory model. Common programming practices to program multi-core CPU in the shared memory systems are to use libraries or parallel programming languages (extensions) or compiler pragmas. Examples are Pthreads [21], TBB [22], HPX [23], Boost::thread [24]. OpenMP [25]. In contrast, distributed memory systems require manually implemented message-passing procedures, e.g., using Message Passing Interface (MPI) protocols [26]. In the HPC community, OpenMP and MPI completely dominate because they are open standards, language neutal, and can be applied to existing code commonly implemented in Fortran, C, or C++.

In the early stages, GPGPU computing was achieved using graphics oriented APIs like

OpenGL shader [18] or Cg [27]. The emergence of GPU computing introduce stream programming model such as StreamIt [28], Sh [29], RapidMind [30], Brook[31], and PeakStream[32]. These low level APIs have evolved into two contemporary dominant languages. CUDA is NVIDIA's extension to C, and lately includes C++ features like template, range-based iterators, auto type, and lambda function. CUDA GPU code is officially supported only by NVIDIA's compiler, and only targets NVIDIA GPUs. OpenCL [33] is a widely supported open standard, which offers a similar model and usage like CUDA. But different from CUDA handled by specific compiler, OpenCL is a library based solution. Low level GPU programming typically comprises several stages, (1) identify parallel workloads, (2) copy data from the host to the device, (3) map workloads to computing cores, (4) determine a suitable memory access for CUDA threads, (5) synchronize the execution between GPU and CPU, and (6) copy data back to the host. Significant efforts are directed at automating these steps. Compiler pragma solutions includes CUDA-lite [34], hiCUDA [35], OpenMPC [36], HMPP [37] and PGI accelerator [38]. The last two have merged into OpenACC [39] as an open standard, and is most influential today. Besides OpenACC, OpenMP has been extended for heterogeneous platforms by introducing similar features since OpenMP version 4 [40]. Transparent GPU code generation research projects, such as Par4All [41] and PPCG [42] concentrate on regular codes in synthetic benchmarks, and without significant additional effort are difficult to generate good performance on real applications in general. Overall, high-level GPU programming languages are not yet versatile enough to fully unleash the power of GPU for complex applications.

In contrast, Xeon Phi is designed to provide massive parallelism at considerably reduced programming effort. Xeon Phi programming could be done in low level using OpenCL. The high level programming model, promoted by Intel, uses a handful of Intel's proprietary pragmas [43] to denote the desired code transformation. With the pragmas supported by Intel compilers, Xeon Phi accelerated binaries can be generated in a similar way as compiling traditional CPU codes [43]; therefore, programming Xeon Phi could be fairly

comparable to coding for multiple-core CPUs. This is denoted as *native* mode. Similar to GPU, Xeon Phi also offers the *offload* mode, where only selected portions of the code marked by compiler pragmas are executed on the accelerator. OpenMP can be used in both native and offload modes alleviating the need for hand-coded parallelization.

2.2 GEAUXDOCK

The goal of drug discovery is to identify, optimize and clinically validate those compounds that bind and modulate the function of a target protein implicated in a disease state. A drug molecule must possess certain geometry and physicochemical properties in order to have a sufficiently high binding affinity toward a given macromolecular target. As a result, the number of bioactive compounds is very small compared to a vast collection of candidate compounds. For example, the ZINC database of commercially available small molecule entities consists of 17,900,742 drug-like compounds collected from 243 vendors as of January 2016 [44]. Considering molecules yet to be synthesized, the chemical universe comprises an estimated novemdecillion (1060) of small organic compounds [45]. At the outset of drug discovery, this large number of candidates need to be downsized to hundreds or thousands of the most promising compounds. Experimental high-throughput screening is a conventional approach used by the pharmaceutical industry to identify bioactive molecules, however, it suffers from high costs and relatively low hit rates [46]. For instance, a recent study by the Tufts Center for the Study of Drug Development estimates that the development of a new prescription medicine typically continues for longer than a decade with the total costs of over 2.5 billion US dollars [47]. Not surprisingly, modern drug discovery is increasingly supported by computational modeling to reduce the overall costs, improve the efficiency and speed up the development time. As an example, a fast drug development is critical in combating the Ebola virus, therefore, computational approaches are expected to significantly contribute to Ebola research through protein structure modeling and large-scale docking of small molecule libraries against viral proteins [48].

One of the most widely used techniques for ligand virtual screening is structure-based molecular docking to model the binding pose of a ligand in the binding site of the receptor protein followed by the prediction of binding affinity and/or free energy [49]. In contrast to ligand-based approaches that require an initial set of bioactive compounds, structure-based docking requires only the 3D structure of the protein target. Moreover, these methods are well positioned to take advantage of the continuously growing structure databases, such as the Protein Data Bank (PDB) [50], providing opportunities to discover novel biopharmaceuticals. Because of the importance of ligand docking in modern drug development, a number of programs have been developed to date [51]. In general, using large compound databases increases the chances of finding bioactives, however, large-scale virtual screening typically requires a long computing time. In addition to the database size, computing time also increases with the increasing accuracy of the modeling of drug-protein interactions. Although sophisticated models outperform simple approaches, these algorithms often have high demands for computational resources. For example, docking accuracy can be improved by incorporating the plasticity of biomolecules, e.g., using pre-generated ensembles of the target protein structure [52]. Since ensemble-based docking requires conducting docking simulation for each target conformation, the computational complexity increases linearly with the number of conformers. Another approach to improve ligand docking incorporates the configurational entropy. This property can be approximated by clustering ligand binding poses generated by a docking program to calculate the conformational similarity between each pair of ligand modes, leading to $\mathcal{O}(n^2)$ complexity, where n is the total number of binding poses. Mining Minima provides a more accurate way to calculate entropy by integrating potential energies as a function of coordinates, however, at a significantly increased computational cost [53]. Finally, the simulation time can also affect the docking accuracy for those docking programs relying on stochastic methods to sample the free energy landscape, where longer simulations are more likely to reach the global minimum [54]

Undeniably, achieving a good balance between the docking accuracy and the computation time represents a major challenge in structure-based virtual screening. To address this problem, parallel computing is often used to accelerate docking simulations. For instance, AutoDock Vina [55] supports multi-threading on CPU using the Boost::thread library yielding significant speedups on multi-core processors compared to a serial version. Moreover, a CUDA implementation of MolDock [56] accelerates both the evolution search algorithm and its two-element scoring functions on GPU , whereas PLANTS [57] employs a systematic grid search with an accelerated scoring function on GPU using a high-level shading language. A few projects take the heterogeneous concept one step further by developing a hybrid docking framework that can be executed on different computer architectures. For example, non-bonded interactions in molecular dynamics kernels were parallelized for both GPU (using CUDA) and CPU (using OpenMP), and further extended to fully utilize distributed platforms through MPI protocols [58]. The docking engine BUDE [59] employs the OpenCL language to maintain a parallel implementation of the genetic search algorithm for CPU, Xeon Phi and GPU. Nonetheless, to the best of our knowledge, an efficient multiple-backend implementation of the docking kernel based on Metropolis Monte Carlo (MMC) has not been reported yet.

Recently, LA-SiGMA team developed GeauxDock, in which I leads the code design and performance practice. GeauxDock is a new molecular docking package to model drug-protein complexes using a mixed-resolution molecular representation and the MMC search engine [60]. GeauxDock uses non-hydrogen atoms for ligands, whereas proteins are described at the coarse-grained, sub-residual level. Such a mixed-resolution description not only helps tolerate structural deformations in the target binding sites caused by using protein models as docking targets, but also speeds up calculations by decreasing the number of interaction points on macromolecules. Furthermore, GeauxDock employs an ensemble-based approach to effectively model the flexibility of ligands and proteins. Ligand ensembles comprise up to 50 low-energy conformations generated at the pairwise root-mean-squared

distance (RMSD) greater than 1 , whereas non-redundant ensembles of 11 conformations are used for proteins. The latter are constructed to mimic the flexibility of drug binding regions in the target receptors. The descriptor-based force field implemented in GeauxDock includes nine energy terms carefully optimized to drive docking simulations toward native-like conformations using a multi-replica MMC sampling.

Although GeauxDock simulations typically converge in less than 1,000 MMC cycles on standard datasets, its large-scale virtual screening applications remain computationally challenging due to a large number of candidate molecules to be evaluated. On that account, this chapter of the thesis describes my efforts porting GeauxDock to multi-core CPUs and massively parallel accelerators, Xeon Phi and GPU. Computational models and performance patterns are analyzed in detail for different architectures. I also discuss various code characteristics as well as general and platform-specific optimization techniques used to turn GeauxDock into an ultra-fast docking tool for large-scale drug virtual screening.

2.3 PERFORMANCE TUNINGS

GeauxDock is designed for virtual screening applications, where a given protein target is screened against a large library of small organic compounds. A docking simulation of a single ligand is an independent computational task. Figure 2.1 shows four stages of virtual screening using GeauxDock. The procedure starts with reading the input data and creating a pool of tasks (Figure 2.1A). Protein and ligand files provide the initial coordinates of the target protein and library compounds. The parameter file specifies various parameters, such as coefficients to calculate energy terms, weight factors to linearly combine individual energy components, as well as the length of rotation and translation vectors to perturb ligand conformations during MMC simulations. Other files contain data to calculate a pseudo-pharmacophore using the Kernel Density Estimation (KDE), restraints on family-conserved anchor substructures using the Maximum Common Substructure (MCS), and a pocket-specific potential (PSP). The KDE component of the scoring function describes

the likelihood of target ligand atoms to be at certain positions with respect to template-bound ligand atoms, whereas the MCS term imposes RMSD restraints according to a chemical matching between the target ligand and template-bound ligands collected from the PDB [60, 61]. Further, PSP is a contact-based statistical potential derived from weakly homologous holo-templates identified by threading rather than all protein-ligand complexes present in the PDB [60, 62]. Once the required input data are read and pre-processed, a computing device is initialized and the data is copied to the accelerator (Figure 2.1B). Subsequently, docking calculations are performed for individual tasks (Figure 2.1C) and finally, the output files are generated on the host (Figure 2.1D).

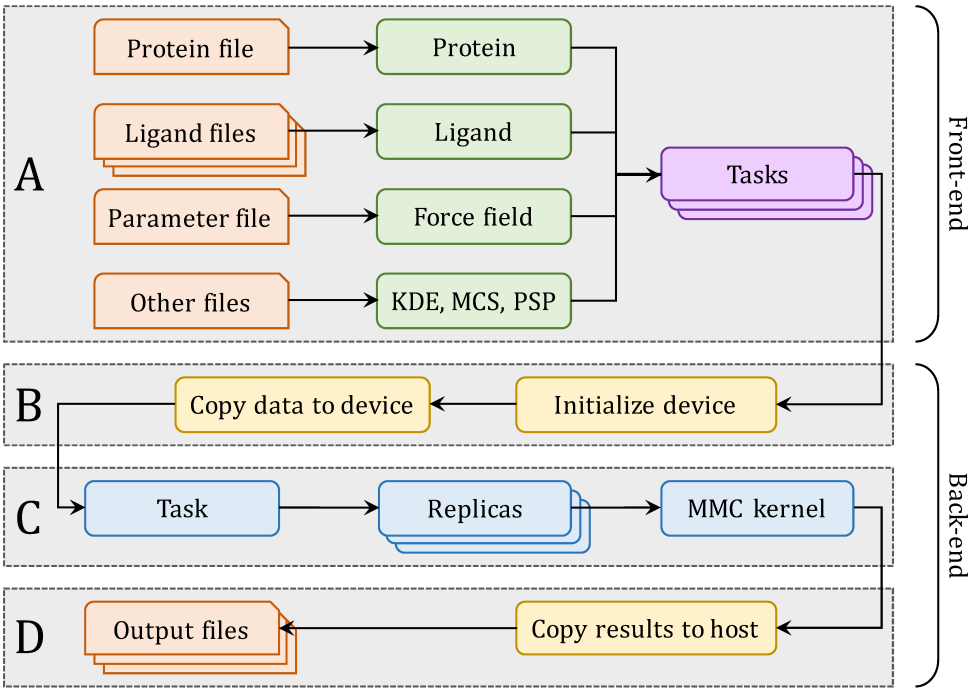

Figure 2.1: Workflow of virtual screening using GeauxDock. (A) The front-end reads input data and creates a pool of docking tasks. The back-end carries out three consecutive operations: (B) device initialization and data transfer, (C) docking calculations for individual tasks, and (D) saving output data.

Preliminary testing of this workflow reveals that the redundant loading and parsing of the same target protein when docking different ligands consumes up to 90% of the total I/O time (Table 1). As a consequence of these excessive I/O operations, the execution of MMC kernels on GPU makes for only 52% of the total simulation time. Furthermore, the

repetitive GPU memory allocation and de-allocation performed for each task takes almost as much time as running the MMC kernel. Although the code for Xeon Phi is expected to have similar issues, the compiler pragmas are placed inside the MMC kernel code, thus the entire offload procedure combines data transfer and core calculations. The memory management for the code offload is not required in the CPU implementation. To address the problem of the excessive I/O operations particularly for GPU-based platforms, the four-step workflow for GeauxDock is arranged into two parts. The front-end consists of data loading, pre-processing and creating a pool of tasks (Figure 2.1A), whereas the back-end fetches tasks, initializes a computing device, executes the docking kernel, and periodically saves the output data (Figure 2.1B-D). With this design, the memory allocation and de-allocation on GPU occur only once at the beginning and the end of the back-end process, respectively.

Docking simulations with GeauxDock can be conducted on three platforms, multi-core CPU, GPU and Xeon Phi. Therefore, the source code is modularized for an easy maintenance across different architectures (Figure 2.2). All three platforms share a common code for front-end computations, whereas back-end codes have two versions, one for CPU and Xeon Phi, and one for GPU. The C++ kernel employing OpenMP and Intel SIMD pragmas is shared between CPU and Xeon Phi. Using the "-Doffload" flag enables additional pragmas protected by the "#ifdef offload" macro, which instruct the compiler to generate object files for Xeon Phi instead of CPU. In contrast, the GPU version comprises a C++ launcher and a docking kernel implemented in CUDA. This design allows for maintaining a single front-end code and two versions of the back-end code. Compiling the source codes (Figure 2.2A) generates architecture-specific object files (Figure 2.2B), which are linked to create different versions of the binary (Figure 2.2C).

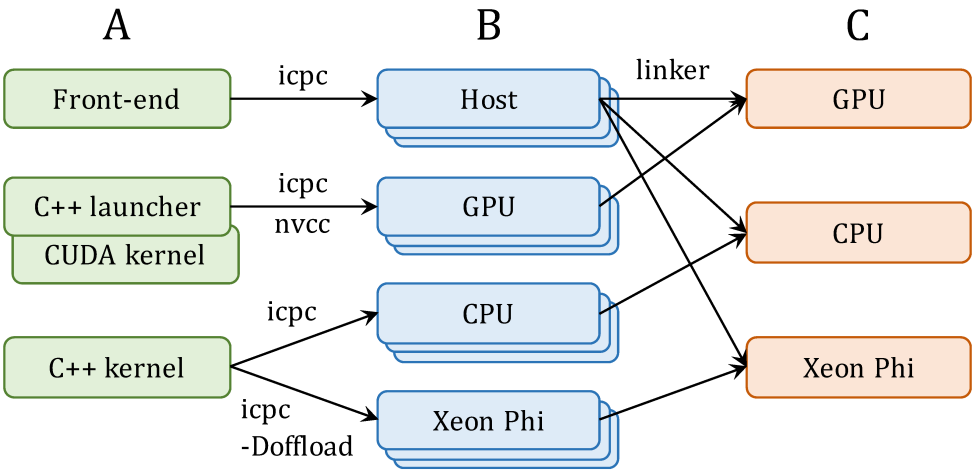

Figure 2.2: Implementation of GeauxDock. (A) The code repository is divided into three modules, a common front-end module for the CPU host and two back-end modules, one for GPU and one for CPU and Xeon Phi. (B) Compiling the source codes produces a series of architecture-specific object files. (C) Linking object files creates three binary versions for GPU, CPU and Xeon Phi.

2.3.1 PARALLELIZATION LEVELS

GeauxDock features an enormous task-level parallelism, where different library compounds docked against the target protein correspond to individual tasks. In addition, the docking kernel exploits coarse- and fine-grained parallelism. Docking calculations for a single task involve multiple protein and ligand conformations, where each unique combination of protein-ligand conformations is regarded as a replica of the system. Although replicas can be subjected to MMC simulations at different temperatures, only one temperature is currently used. For a given docking task, the corresponding ensembles of independent replicas are suitable for coarse-grained parallel computing. Moreover, a fine-grained parallelization takes place at the level of pairwise interactions between data points within each replica. These interactions are computed as three matrices, $protein_{ColumnVector} \times ligand_{RowVector}$ (PRT), $KDE_{ColumnVector} \times ligand_{RowVector}$ (KDE), and $MCS_{Matrix} \times ligand_{ColumnVector}$ (MCS). Here, a fairly large number of computations are subjected to fine-grained parallelization; the analysis of input data reveals up to 104 data points for a single replica, which is sufficient to saturate computing resources available on modern CPUs and accelerators.

Back-end calculations start when a task is fetched from the task pool. Figure 2.3 and

Table 2 explain mapping between the docking algorithm and computing resources. First, replicas within each task are mapped to coarse-grained resources, GPU streaming multiprocessors (SMs) as well as CPU and Xeon Phi cores (Figure 2.3A and Table 2, Coarse-grained parallelism). When multiple GPUs are available, replicas within a given task are evenly assigned to the attached GPU cards. Second, interaction-level calculations (Figure 2.3B) are mapped to fine-grained resources, where computing 2D matrices utilizes SIMD lanes on CPU and Xeon Phi, and CUDA threads on GPU (Figure 2.3C and Table 2, Fine-grained parallelism). Code 1 in S1 Codes illustrates loop operations on PRT, KDE, and MCS matrices involving a number of summation reductions. For instance, five energy terms calculated using the PRT matrix (E_{ele}^{soft}, E_{vdW}^{soft}, E_{HB}, E_{CP}, and E_{CP}^{PS}) are directly reduced from a 2D array to a scalar value. Another type of reduction is hierarchical, where a 2D array a[i][j] is first reduced to a 1D array b[i] along the j-dimension, and then to a scalar value along the i-dimension. This technique is applied to selected data across all three matrices, e.g., E_{HP} in the PRT matrix, E_{KDE} in the KDE matrix, and E_{MCS} in the MCS matrix. In order to implement hierarchical reductions on GPU, I made adjacent GPU threads efficiently exchange data by scheduling the i-dimension as the outer loop, and the j-dimension as the inner loop. Specifically, the outer (inner) loop iterates over ligandRowVector (proteinColumnVector) for the PRT matrix, ligandRowVector (KDEColumnVector) for the KDE matrix, and rows of MCS_{Matrix} (columns of MCS_{Matrix}) for the MCS matrix.

2D CUDA thread blocks are responsible for calculations on GPU (Figure 2.3A, green rounded boxes). The shape and size of CUDA thread blocks are flexible and can be tuned for the optimal performance. Given that the CUDA warp size is fixed at 32, the x-dimension of the CUDA thread block is best defined as a multiple of 32. Also, the maximum number of 1,024 threads per CUDA thread block restricts the y-dimension, for example, the size of the y-dimension cannot be greater than 32 when x-dimension is 32, because $32 \times 32 = 1024$. However, the shapes of 2D interaction matrices do not always perfectly match those of CUDA thread blocks. For instance, the x-dimension is always greater than the y-dimension

in PRT and KDE matrices, whereas a typical MCS matrix has the y-dimension greater than the x-dimension. Therefore, boundary conditions require a careful design of CUDA thread blocks to leave a certain number of idle threads for the thread management. This procedure is illustrated in Figure 2.4, where processing a small, 70-element data matrix (outlined in red) requires at least six cycles of a 4×4 CUDA thread block (each cycle is outlined in blue). With this setup, 70 parallel threads are fully utilized (gray cells), leaving 26 threads idle (white cells). Overall, the number of CUDA threads is fixed at the compiling time, but the optimal shape of the thread block is defined at the runtime, when the input data become available. Here, the objective is to find the best combination of x- and y-dimensions consuming the least amount of computing cycles to traverse the data matrix, where a computing cycle is defined as follows:

$$cycle = ceiling\left(\frac{data\ size_x}{cuda\ threads_x}\right) \times ceiling\left(\frac{data\ size_y}{cuda\ threads_y}\right) \tag{2.1}$$

In practice, only a handful of configurations are valid; I enumerate and evaluate these configurations to find the optimal solution. As an example, using Tesla K20Xm GPU with 1,024 threads per thread block, a typical configuration for PRT, KDE, MCS matrices is 128×8, 128×8, and 32×32, respectively.

Different from the GPU version, the back-end for CPU implemented in C++ with OpenMP pragmas assigns processor threads to carry out computations for individual replicas (Figure 2.3A, blue rounded boxes). In order to avoid thread migration and ensure the best cache locality, the environment variable "OMP_PROC_BIND" is set to "true." In addition, inner loops in data computations iterating over proteinColumnVector (PRT matrix), KDEColumnVector (KDE matrix), and columns of MCSMatrix (MCS matrix) are marked with vector pragmas to assist Intel compiler in generating an efficient, vectorized code. Note that the same CPU code can be used on Xeon Phi since almost all performance tuning techniques for CPU apply to this accelerator as well. The major difference is that the code for Xeon Phi is required to be offloaded to the accelerator, which is conceptually

similar to GPU programming. The offload is accomplished using compiler pragmas, i.e. "#pragma offload target (mic) in (data_in) out (data_out). However, the present pragma-based Xeon Phi programing model was designed to offload a block of code to only one device. The current implementation of GeauxDock works only with a single Xeon Phi card. Although replicas could be distributed manually across multiple accelerators, one should keep in mind that at least 240 replicas are required to effectively utilize Xeon Phi. Since docking tasks have no more than 550 replicas, splitting the workload among multiple Xeon Phi cards would inadvertently decrease the overall performance. In addition, any code modification targeting the Xeon Phi platform would complicate the code maintenance. In fact, workload sharing at the task level represents a more practical and scalable approach, which will be implemented in the future release of GeauxDock.

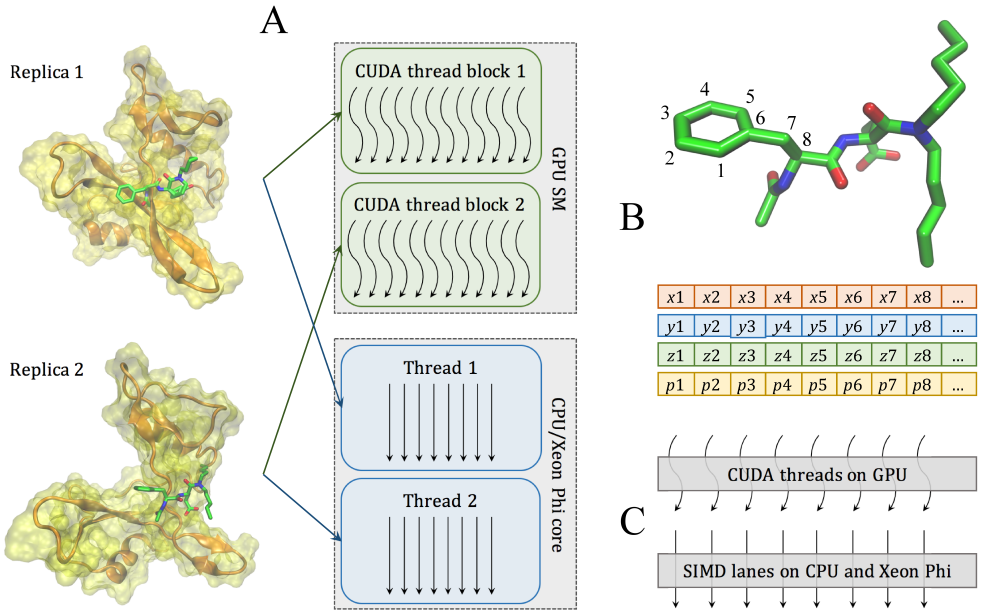

Figure 2.3: Two levels of parallelism in the docking kernel. (A) At the coarse-grained level, individual replicas are assigned to different CUDA thread blocks on GPU streaming multiprocessors (SMs) and different threads on CPU/Xeon Phi cores. (B) At the fine-gained level, data points for each replica are organized as Structure of Arrays containing Cartesian coordinates x, y, z, and parameters p associated with atoms, such as type, charge, and etc. Parameters for neighboring atoms are placed closely in memory to ensure the best execution efficiency. (C) Data points at the fine-gained level are accessed in parallel by CUDA threads on GPU and SIMD lanes on CPU and Xeon Phi.

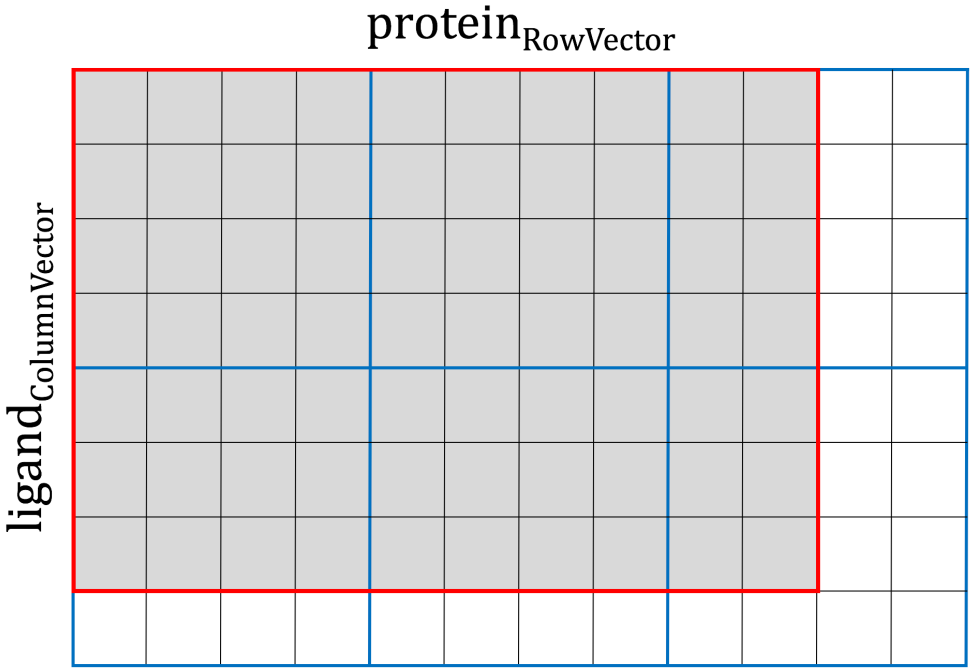

Figure 2.4: Example of parallel calculations for a data matrix. A small, 96-element matrix ligandColumnVector × proteinRowVector is outlined in red, whereas the 4 × 4 CUDA thread block iterating over the matrix is outlined in blue. Here, at least 6 cycles are required to process the data matrix utilizing a total of 70 parallel threads (gray cells), while the remaining 26 threads are idle (white cells). An optimal shape of CUDA thread blocks can be constructed dynamically to improve the computational performance by reducing the number of cycles required to traverse the data matrix.

2.3.2 DATA STRUCTURE

A docking task contains complex data, including read-only protein and ligand conformations, MMC simulation parameters, MCS, KDE and PSP force field parameters, as well as the dynamic configuration and output data from individual replicas. GeauxDock employs the Structure of Arrays (SoA) to store the data ensuring the best data locality. For example, the SoA for the ligand conformation shown as Code 2A in S1 Codes contains elements x[L], y[L], z[L], t[L], and c[L], representing x, y, z coordinates, the type, and electric charge for all ligand atoms, respectively. L defines the maximum number of ligand atoms and it is set at the compiling time. Figure 2.3B shows that the data associated with neighboring atoms are stored in consecutive memory addresses in order to maximize the efficiency of memory operations required for the fine-grained parallelization. With this

design, CUDA threads on GPU and SIMD lanes on CPU and Xeon Phi access these data in a stride-1 pattern as illustrated in Figure 2.3C. Data structures for protein conformations, MMC simulation parameters, and PSP, KDE and MCS force field parameters are created in a similar fashion. These data constitute the first-level SoA providing read-only information, and are used as building blocks to construct the multiple-replica simulation context.

To systematically assemble replicas from these raw data, I created a data structure called "ReplicaInfo," whose purpose is to assemble a replica from the raw data using indirect references to various arrays. The concept of ReplicaInfo is presented in Figure 2.5, where two example replicas, (L1, P1, T1) and (L1, P3, T2), are created using indexes to the same ligand conformation (L1), but different protein conformations (P1 and P3) and simulation temperatures (T1 and T2). ReplicaInfo was designed to yield a high computational efficiency of data exchange between replicas during parallel tempering MMC simulations [63], which requires swapping only a few indexes rather than the associated large data arrays. Further, the ReplicaInfo structure is used to store the temporary simulation status, including energy values and ligand orientations with respect to the target protein pocket. Simulation logs are saved in the "Simlog" data structure, whose entry can also be found in ReplicaInfo. I note that the ReplicaInfo can be modified during MMC simulations, while the associated data are read-only.

In addition to the first-level SoA, I designed the second-level SoA called the "Complex" (Code 2B in S1 Codes) providing the outermost container for the computation data. The elements of Complex are various data structures, including protein and ligand conformations, MMC simulation parameters, MCS, KDE and PSP force field parameters, ReplicaInfo, and the data size. Essentially, a single instance of Complex SoA and Simlog hold all data associated with a computation task. Because the memory for Complex and Simlog is allocated only once, when either the CPU/Xeon Phi or GPU version of GeauxDock is initiated, it must be large enough to hold data for any docking tasks from the task pool. Docking cal-

culations for the CCDC/Astex dataset require about 5 MB of memory for each Complex, whereas the entire Simlog would allocate about 1.5 GB of memory. In practice, only about 100 MB of Simlog data need to be transferred to the host and saved on disk.

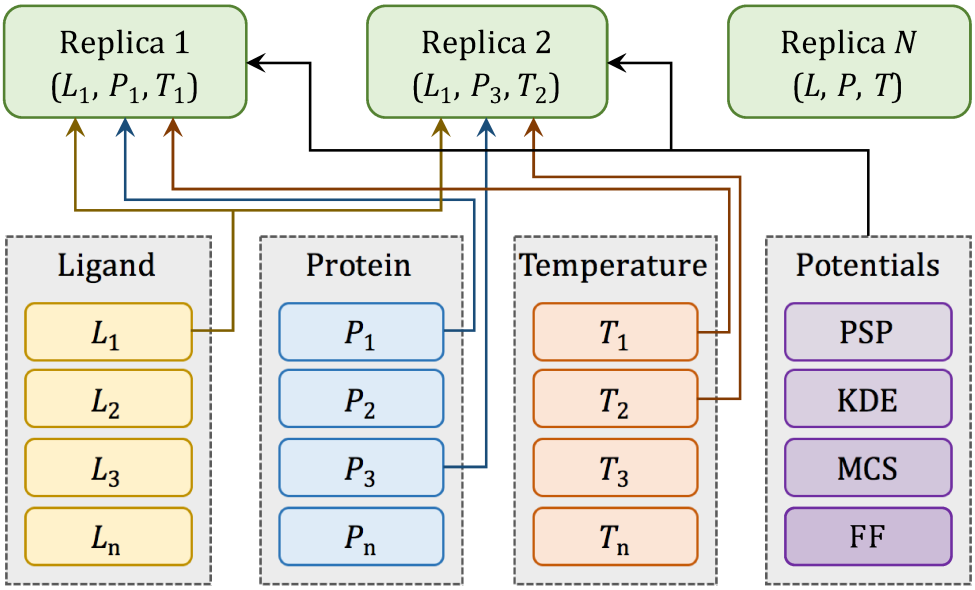

Figure 2.5: Data indexing for multi-replica Monte Carlo simulations. Individual replicas are multi-dimensional objects comprising different combinations of ligand (L) and protein (P) conformations, and temperatures (T), as well as the same set of PSP, KDE, MCS potentials and force field (FF) parameters. All these data are read-only, labeled with tags, and accessible through indexes as depicted by arrows.

2.3.3 DATA REARRANGEMENT

Irregular code patterns caused by dynamic data may significantly affect the performance. The docking kernel code contains conditional branches and indirect memory references, for example, calculating a branch path depends on the distance between a ligand atom and a protein point, which is changing in the course of MMC simulations. Although it is difficult to speed up the code containing these dependencies, I improved the code regularity for certain cases. For instance, incrementally sorting KDE data elements by the atomic type t helps improve the regularity of the conditional code “if (lig-¿t[index] == kde-¿t[index])”

in a loop iterating over hundreds of KDE data points. Another example is the indirect

memory reference, such as "d = array[ligand-¿t[index]][protein-¿t[index]]." Here, sorting ligand and protein objects by t greatly improves the locality of accessing array elements. Altogether, data rearrangement enhances the performance of GeauxDock by 9.6%, 12.2% and 8.2%, on CPU, Xeon Phi and GPU, respectively.

2.3.4 STRENGTH REDUCTION

In order to further speed up calculations within the docking scoring function, the strength reduction technique is applied to reduce its computation complexity. Original mathematical formulas for various energy terms in the MMC kernel are divided into pre-processing and computation groups. The pre-processing combined with data transformation is conducted within the front-end of GeauxDock. An example is shown as Code 3 in S1 Codes, where the indirect memory reference "prtconf.r[index]" is removed from the original kernel (Code 3A) and included in the pre-processing stage (Code 3B), leading not only to a better memory locality, but also to fewer instructions in the optimized kernel. Another technique used to accelerate computations within the docking kernel is the reduction of the arithmetic intensity. For instance, Code 4A in S1 Codes shows a part of the original kernel computing the soft van der Waals potential, which includes 6 loads, 9 multiplications, 3 division and 5 power functions. To speed up the MMC kernel, some calculations are either moved to the pre-processing step or executed between certain blocks of the code and then reused when calculating the potential. As the result, the optimized code shown as Code 4B in S1 Codes has only 2 loads, 6 multiplications, 3 divisions and no power functions.

2.3.5 GPU-SPECIFIC TUNING

The power of accelerators can be fully utilized only when time is primarily spent on computations rather than data communication. GeauxDock is implemented based on this principle by moving compute-intensive MMC simulations to Xeon Phi and GPU. Code 5 in S1 Codes shows the MMC conformational sampling in ligand docking. First, a new

configuration of a ligand is generated by randomly perturbing the present configuration. Next, the energy of the new configuration is calculated and compared to the energy of the old configuration using the Metropolis algorithm [64]; the new configuration is accepted with a certain probability to be used in the next iteration, otherwise it is rejected. Even though some components of the docking kernel, such as evaluating the Metropolis criterion, are less suitable for the parallelization on GPU and Xeon Phi, this approach yields a better overall performance than offloading parts of the docking kernel. For instance, offloading only energy calculations could potentially generate an excessive communication between the host and the accelerator. In that case, advanced optimization techniques such as the asynchronous kernel execution and data copying between multiple tasks would have to be applied for a better performance. However, because extra communication is avoided in the MMC kernel, the code requires no further optimization of data transfer.

For GPU, the memory is carefully managed within the GeauxDock code with heavily reused variables, such as interaction distances, placed in registers. Moreover, the shared memory is used for those frequently reused data, such as ligand coordinates and energy parameters, which may have an irregular access pattern. Large arrays with the stride-1 parallel access pattern are defined as SoA, sorted for improved regularity, and saved in the global memory. Importantly, level 1 data cache on Tesla K20Xm GPU does not buffer the global memory traffic by default. The docking kernel has a good reuse pattern for PRT and KDE matrices, therefore, inserting _ldg intrinsic enables the level 1 data cache mechanisms to enhance memory operations. This technique improves the GPU performance by 4% for PRT and KDE matrices. In contrast, the cache optimization cannot be applied to computations for the MCS matrix, which have no global data reuse at all.

Since the docking kernel invokes reduction operations, partial results in each CUDA thread need to be added to a scalar value. Here, a simple implementation stores temporary data in the shared memory, where the amount of the required memory scales linearly with the number of CUDA threads. In the early version of GeauxDock, the capacity of the

shared memory limited the maximum number of CUDA threads per thread block to 768. Since using more CUDA threads per block generally delivers a better performance on Tesla K20Xm GPU, the current docking kernel uses __shfl and __shfl_xor intrinsic instructions for reduction operations. This technique enables a direct data exchange between CUDA threads without consuming the shared memory. Not only is the new reduction code 3 × faster, but it also allows to use 1,024 CUDA threads per block improving the overall performance by 40%. Finally, many elementary functions, exp, log, sin, cos, etc., are frequently used in the docking kernel. The CUDA math library offers accelerated versions of these math functions [20], which are enabled by the "-use_fast_math" compiler flag. This tuning yields a 30% performance boost, however, the fast math intrinsic for GPU is not guaranteed to be fully compatible with the IEEE floating point standard. Nonetheless, a careful comparison of the results against the CPU code shows that the error rate is smaller than 0.0001

2.4 RESULTS

The performance of MMC kernels in GeauxDock is evaluated on several computing platforms using diverse input data. I conducted benchmarking calculations using four Linux computers listed in Table 3, including a mainstream PC desktop, a PC desktop with the latest consumer grade GPU, a heterogeneous HPC cluster node with both GPU and Xeon Phi accelerators, and an HPC cluster node with two GPU cards. I set the optimization level to "-O3" with the following additional flags for the Intel compiler: "-fno-fnalias -ansi-alias -fargument-noalias" (to safely remove pointer aliases), "-ipo" (to enable inter-procedural optimization), "-vec-threshold0" (to enable vectorization whenever possible), and "-fma" (to enable the fused-multiplication-add code generation). Architectural events listed in Table 4 were recorded by hardware counters using the Performance Application Programming Interface (PAPI) library version 5.4.0 [65]. In addition, I implemented timers directly in the code in order to measure the execution time of an arbitrary segment of the

code. I noticed that time measurements have minor fluctuations of nearly 5%, therefore, all timings are reported as the average over 8 independent runs.

Benchmarking calculations are carried out for a single target protein, the pp60(c-src) SH2 domain complexed with ace-malonyl Tyr-Glu-(N,N-dipentyl amine) (PDB-ID: 1a07) [66] and a set of 204 drug compounds selected from the CCDC/Astex dataset [67]. 1a07 represents a typical docking target with 344 protein effective points and an ensemble of 11 protein conformations. Depending on the number of rotatable bonds, up to 50 conformations are generated for ligands, thus the ensemble-based docking employs up to 550 replicas (11×50) of individual systems. In addition to this default protocol, I test the code scalability using a varying number of replicas at multiple temperatures. Other parameters affecting the computational complexity are the number of non-hydrogen ligand atoms and the number of points to compute the evolution-based components of the GeauxDock force field, KDE and MCS. Although both KDE and MCS scoring terms are used to calculate various restraints derived from homology rather than physical interactions, these points are iterable from the computing point of view. Therefore, KDE and MCS interacting points are equivalent to ligand atoms and protein effective points in the physics-based components of the GeauxDock force field.

The distributions of the number of replicas, ligand atoms, as well as KDE and MCS points are shown in Figure 2.6. GeauxDock employs multiple replicas to account for the flexibility of protein-ligand complexes, where each replica contains a unique combination of protein and ligand conformations. The highest peak in Figure 2.6A at around 550 replicas corresponds to highly flexible compounds with multiple rotatable bonds, whereas the smaller peak at around 11 replicas represents those rigid complexes having only a single conformer. Given that the hydrogen atoms are omitted when counting atoms, the range between 6 and 62 heavy atoms presented in Figure 2.6B agrees well with the qualifying range for drug molecules according to the extended version of Lipinski's rule-of-five [68]. Because KDE points and rows in MCSMatrix are calculated using template-bound ligands detected

by the eFindSite algorithm [61], their distributions (Figures 2.6C and 2.6D, respectively) depend on the number and size of ligands extracted from holo-templates.

Another important simulation parameter is the number of MMC cycles. I found that 1,000 MMC cycles is sufficient for production runs to converge. Since these calculations require 4.8 to 61 minutes on various platforms, the average wall time for the docking kernel is 1.4 seconds on the fastest machine (platform D2, Table 3) and 18 seconds on the slowest computer (platform D1, Table 3). Because the number of replicas (up to 550) is multiplied by the number of temperatures (up to 240) in our benchmarks, and several versions of the docking code needed to be tested, the time required to complete simulations could be hundreds times longer than that for production runs. Therefore, shorter simulations with 100 MMC cycles are used for benchmarking purposes.

2.4.1 PERFORMANCE WITH AN AMPLE COARSE-GRAINED PARALLELISM

The execution time for docking kernels includes not only computations but also time required for the data transfer to and from accelerator devices. Moreover, the kernel performance can be affected by the ensemble size (the number of replicas), because those docking systems containing rigid ligands provide insufficient coarse-grained parallelism to fully utilize computing resources. On that account, I first need to determine the ideal performance as well as a performance penalty caused by the meager coarse-grained parallelism. To address this problem, I conducted a series of simulations providing a sufficient number of replicas to deliver an ample coarse-grained parallelism. Specifically, I used 400 replicas for a dual CPU with 20 cores and 20 threads, 2,400 replicas for Xeon Phi with 60 cores and 240 threads, and 280 replicas for GPU with 14 streaming multiprocessors and 14 CUDA thread blocks.

The performance of docking kernels on CPU is assessed using the C1 computing system (Table 3). I first evaluate the serial performance by enabling only 1 thread on a

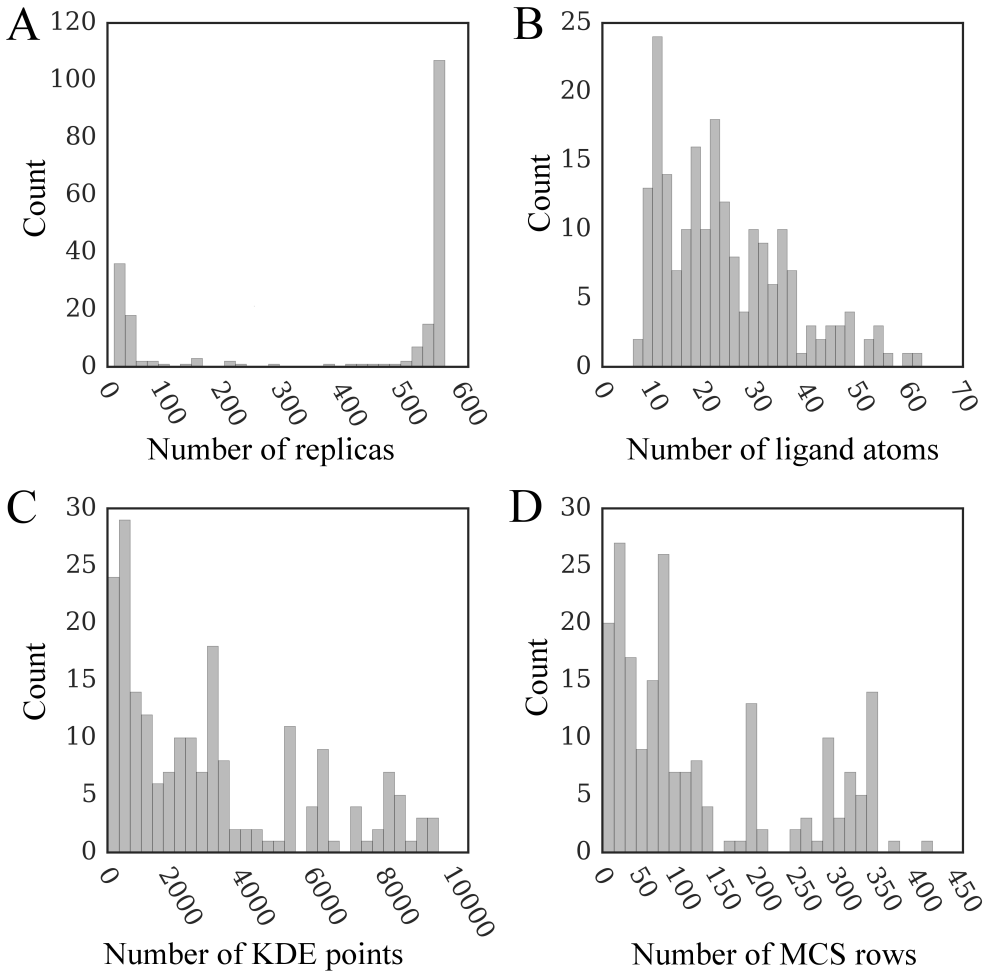

Figure 2.6: Distribution of various parameters affecting docking time for the dataset of 204 CCDC/Astex compounds. The number of (A) replicas, (B) ligand non-hydrogen atoms, (C) KDE points, and (D) rows in the MCS matrix. KDE (Kernel Density Estimation) and MCS (Maximum Common Substructure) points are used to calculate evolution-based components of the docking force field.

single processor core. Using the total number of CPU cycles according to the PAPI event PAPI_TOT_CYCLES (Table 4) and the computing time measured by either the PAPI timer or our timer, the average dynamic CPU clock rate is 3.58 ± 0.02 GHz. Figure 2.7 shows several characteristics assessing the overall computational performance of the docking code. In most cases, the number of level 1 data cache misses per 103 instructions is less than 7 (Figure 2.7A), which is lower compared to a broad distribution of 5-30 misses reported for thoroughly tuned SPEC CPU2006 benchmark kernels [69] tested on the same CPU microarchitecture. Similarly, the number of branch mis-predictions per 103 instructions

for the SPEC CPU2006 kernels is between 1 and 10 [69], therefore, the docking code is superior with no more than 2 branch mis-predictions (Figure 2.7B). Moreover, GeauxDock achieves an average instruction throughput rate of about 2, which is notably higher than 1.43 instructions per cycle reported for the most efficient SPEC CPU2006 kernel [69]. This comparison with the SPEC CPU2006 benchmark suite demonstrates that the serial, CPU version of the docking kernel in GeauxDock is indeed highly optimized.

Next, using the optimized serial CPU code as a baseline, I measure the performance of the parallel versions of GeauxDock on a dual multi-core CPU, Xeon Phi and GPU using the C1 computing system (Table 3). Enabling 20 threads on a dual CPU triggers the dynamic frequency scaling and decreases the average CPU clock rate to 3.07 ± 0.11 GHz. Figure 2.8A shows that the average speedup of multi-threaded GeauxDock over its serial version is 17.22 ± 0.06, which actually corresponds to the maximum theoretical speedup accounting for the lower clock rate ($20 \times \frac{3.07GHz}{3.58GHz}$).

Further, compared to the serial code, the parallel docking kernel runs from 22 $\times$ to 56 $\times$ faster on Xeon Phi 7120P (Figure 2.8B) and 10 $\times$ to 38 $\times$ faster on Tesla K20Xm GPU (Figure 2.8C). Despite these impressive speedups, the irregular portions of the docking code are handled differently by various devices because of their architectural characteristics causing significant variations across the dataset. As I mentioned in the introduction section when discussing hardware design, the simpler computing units of Xeon Phi and GPU are more susceptible to dynamic branches compared to sophisticated CPU cores.

2.4.2 PERFORMANCE ON REAL DATA

Next, I test the parallel performance of each platform against realistic workloads. Figures 2.8D and 2.8F show that multi-threaded CPU and GPU versions of the docking kernel generally maintain their high performance on real data. In contrast, the performance of Xeon Phi is significantly affected by the lack of an ample coarse-grained parallelism (Figure 2.8E). Although the co-processor is twice as fast as a dual CPU in 71.1% of the cases (a

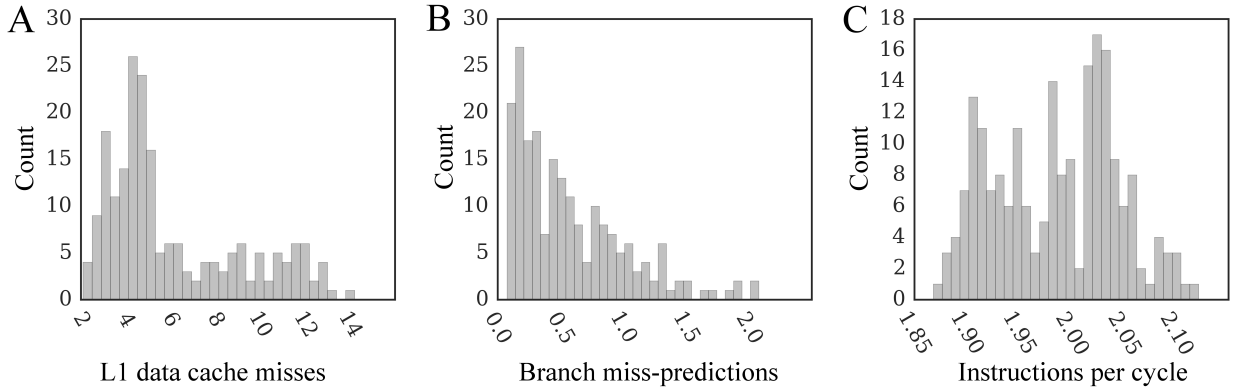

Figure 2.7: Performance characteristics for a single-threaded docking kernel on CPU. The number of (A) level 1 data cache misses per 10^3 instructions, (B) branch miss-predictions per 10^3 instructions, and (C) instructions per cycle.

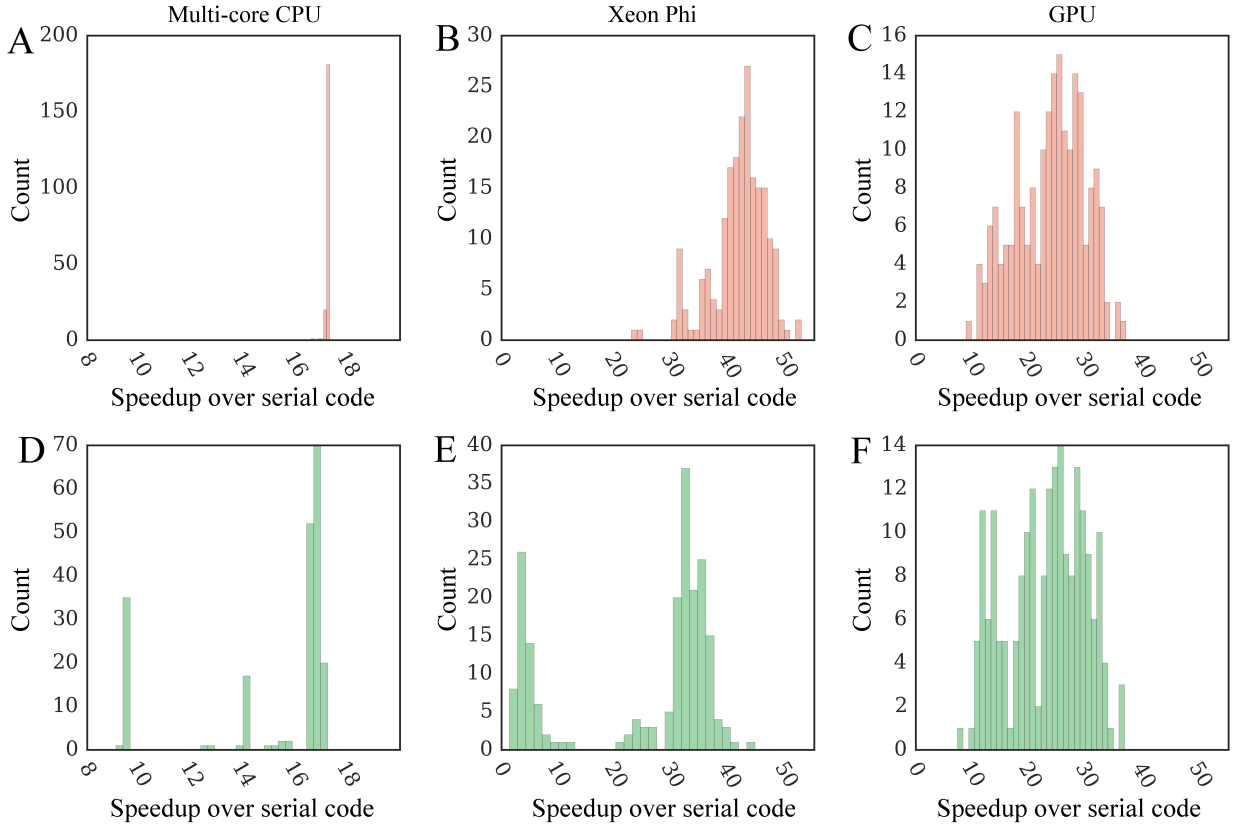

Figure 2.8: The distribution of speedups of parallel GeauxDock over the serial CPU version for the dataset of 204 CCDC/Astex compounds. Benchmarking calculations are conducted using (A-C, red) modified input data providing an ample coarse-grained parallelism and (D-F, green) unmodified input data. Three kernel implementations are tested for (A, D) multi-core CPU, (B, E) Xeon Phi, and (C, F) GPU.

speedup of 17× and more), Xeon Phi performs about twice as slow as a dual CPU for the remaining docking systems. This double peak pattern matches the bimodal distribution of the number of replicas shown in Figure 2.6A, demonstrating that the computational throughput of Xeon Phi is significantly affected by those workloads providing insufficient coarse-grained parallelism.

To further investigate the effect of the number of replicas on the parallel performance, I compiled a separate testing dataset comprising a single conformation of the target protein 1a07 and a rigid ligand adamantanone (PDB-ID: 5cpp) [70]. This docking system is replicated n times at different temperatures to strictly control the number of replicas in docking simulations. The docking time for multi-core CPU, Xeon Phi and GPU kernels are presented in Figure 2.9. Figures 2.9A and 2.9C show sets of horizontally parallel lines with even vertical distances, whose width corresponds to the number of CPU cores and GPU streaming multiprocessors, respectively. Here, replicas are processed in parallel by independent computing units with the execution time equal to the number of replicas divided by the core count. The width of horizontal lines for Xeon Phi shown in Figure 2.9B is 240 because of the hardware multi-threading (60 cores × 4 threads per core). Clearly, it is beneficial to place 4 threads on a single core in order to fully utilize the hardware. Moreover, the kernel time for the first few data points at the beginning of each horizontal line is somewhat shorter demonstrating that the co-processor performance is affected by the global resource contention.

2.4.3 A RELIABLE MODEL FOR THE PERFORMANCE

To further understand the performance characteristics, I analyze various components of the docking kernel including the time spent on computing PRT, KDE, and MCS interaction matrices. KDE and MCS data are used to calculate evolution-based components of the docking force field, whereas the PRT matrix is used to calculate physics-based potentials. The time spent on computing the remaining operations is measured using a modified

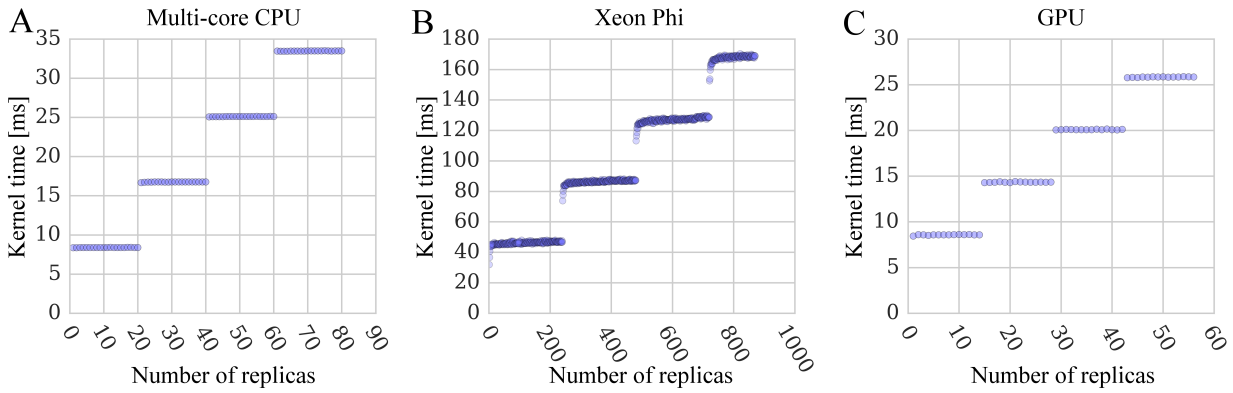

Figure 2.9: Performance scaling of docking kernels with different numbers of system replicas. Benchmarking calculations are performed using (A) multi-core CPU, (B) Xeon Phi, and (C) GPU. The width of horizontal lines is 20 replicas for a dual 10-core CPU, 240 for a 60-core Xeon Phi with 4-way multi-threading, and 14 for a 14-multiprocessor GPU.

kernel, in which PRT, KDE, and MCS calculations are disabled. Figure 2.10 shows time contributions from these four components. Computing PRT contributes to 64.4%, 60.4%, and 32.1% of the total execution time on CPU, Xeon Phi, and GPU, respectively (Figures 2.10A-C). The percentage of the kernel time for KDE is 33.9% on CPU, 28.2% on Xeon Phi, and 46.3% on GPU (Figures 2.10D-F), whereas for MCS, it is 2.7% on CPU, 5.1% on Xeon Phi, and 10.4% on GPU (Figures 2.10G-I). The remaining operations make up about 10% of the total kernel time on Xeon Phi and GPU. In contrast, these computations require almost no time on CPU because the sophisticated processor cores handle sequential workloads (e.g., updating ligand coordinates, generating random numbers, calculating Metropolis acceptance criterion, etc.) as efficiently as highly parallel workloads. Further, the CPU code has no data transfer between the host and the accelerator, which is required only for Xeon Phi and GPU.

Next, I analyze the correlation between the computing time and the static data size. In addition to the original docking code, I examine the performance impact of dynamic branches by forcing the calculation of all operations; this modified implementation is referred to as a "regulated" code. Figure 2.11 shows the correlation between the execution time and the data size for the original program in blue and the regulated code in red. Figures 2.11A-F demonstrate that the time required to calculate the PRT (KDE) matrix

strongly correlates with its size; the coefficient of determination, R^2, for the original code shown in blue is 0.996 (0.938) for CPU, 0.996 (0.987) for Xeon Phi, and 0.952 (0.981) for GPU. This correlation is somewhat weaker for the MCS matrix with the R^2 of 0.957, 0.720 and 0.793 for CPU, Xeon Phi and GPU, respectively. Forcing the execution of the entire code by eliminating dynamic branches has two major effects on the kernel performance. First, it improves the correlation between the computing time and the data size, for instance, the R^2 for the KDE matrix shown in red in Figures 2.11D-F is 0.999 for CPU and Xeon Phi, and 0.983 for GPU. Second, the regulated code is slower, however, the relative increase of the execution time is clearly architecture-dependent. In general, CPU skips executing most of the instructions downstream of branches because their conditional outcome can be accurately predicted, which yields a better performance (Figures 2.11A and 2.11D). The performance of GPU (Figures 2.11C and 2.11F) is unaffected by branches indicating that this accelerator always performs the predicated execution. Interestingly, the branch behavior of Xeon Phi falls between CPU and GPU. For the PRT matrix (Figure 2.11B), Xeon Phi performs the predicated execution similar to GPU, whereas the branch prediction clearly helps reduce the execution time on Xeon Phi for the KDE matrix when the KDE elements are sorted (Figure 2.11E). Nonetheless, the performance improvement for Xeon Phi is not as large as that for CPU because its computing cores are simpler and the wider SIMD vectors are generally less suitable for irregular data.

The original code improves the performance of computing PRT and KDE, however, it negatively impacts the calculation of the MCS. This effect can be attributed to the irregularity and shape of the MCS data structure containing a dense ligandColumnVector, but a sparse MCSMatrix. Note that since proteinColumnVector (Figures 2.11A-C) and KDEColumnVector (Figures 2.11D-F) data structures are 1D arrays, there is a branch pattern between different elements, which can be further improved by data sorting. This pattern is lost in the sparse $MCSMatrix \times ligandColumnVector$ causing a significant branch prediction penalty and longer execution times for CPU and Xeon Phi (Figures 2.11G

and 2.11H). On the GPU platform, I analyzed two versions of the generated Streaming ASSembly (SASS) code. The original SASS code always performs predicated execution, while the regulated SASS code uses non-predicated instructions without testing branch conditions. For that reason, the regulated docking code performs better for the irregular MCS data.

As mentioned above, the correlation between the computing time and the size of the MCS matrix also tends to be weaker than that for PRT and KDE matrices. For instance, the R^2 for the original (regulated) code shown in blue (red) in Figures 2.11G-I is 0.957 (0.946) for CPU, 0.720 (0.744) for Xeon Phi, and 0.793 (0.749) for GPU. This effect can be explained by the fact that the MCS data matrix is limited by the number of ligand atoms, which is between 6 and 62 for the CCDC/Astex dataset (Figure 2.6B). Consequently, the MCS matrix is not wide enough to efficiently utilize vector lanes on CPU (8 elements) and on Xeon Phi (16 elements) as well as the x-dimension of 2D CUDA thread blocks on GPU (32 elements); see Table 2. Consider a ratio of the data size and the number of cycles (Equation 2.2, 2.3:

$$ratio = \frac{data\ size_x}{cycles} \tag{2.2}$$

with the number of cycles required to traverse the x-dimension of the MCS matrix given by:

$$cycles = ceiling\left(\frac{data\ size_x}{vector\ width_x}\right) \tag{2.3}$$

For PRT and KDE matrices, whose data size is much larger than the vector width, the ratio in Equation 2 is close to the vector width yielding a strong linear correlation between the computing time and data size. In contrast, performance fluctuations caused by idle cycles created by the underutilized vector lanes (Equation 2.3) slightly decrease the correlation for the MCS matrix.

2.4.4 COMPARATIVE BENCHMARKS OF HETEROGENEOUS PROCESSORS

Finally, I perform comparative benchmarks of all computing platforms listed in Table 3 using the 1a07 target protein and the dataset of 204 CCDC/Astex ligands. In these simulations, I use the original GeauxDock code and the real data with respect to the number of protein and ligand conformations. Timing reports include the total execution time of the docking kernel for 204 tasks and the simulation wall time averaged over 8 independent docking runs for each task. GeauxDock is specifically designed for virtual screening applications, therefore, it reads the target protein input data only once for a given set of docking ligands. Indeed, GeauxDock spends from 95.4% (GeForce GTX 980) to 99.7% (Xeon E5-2680 v2) of the total time executing docking kernels, while loading and pre-processing input data take only about 10 seconds on average (Table 5). The reference time required to complete docking calculations for the entire dataset is 61.31 minutes using a multi-threaded CPU version running on Core i7-2600 multi-core CPU (platform D1, Table 3). Figure 2.12 shows that high-performance servers and hardware accelerators yield significant speedups over a mainstream PC desktop. GeForce GTX 980 is the fastest computing device in our tests, which achieves a 12.6 × speedup and dramatically reduces the wall time to only 4.84 minutes. Xeon Phi gives a a 6.8 × speedup corresponding to the wall time of 9.00 minutes, whereas the performance of a single Tesla K20Xm card with 11.14 minutes of wall time is about 23% worse than Xeon Phi. It is noteworthy that I obtained almost a perfect scaling on multiple GPU cards; using a pair of K20Xm GPUs increases the performance by 98%, compared with a single K20Xm GPU. A dual Xeon E5-2680 CPU needs 16.99 minutes to complete docking calculations, which is about 3.6 × faster than the baseline i7-2600 CPU running at a higher clock rate.

One should keep in mind that not only the theoretical peak performance, but also the cost and the energy consumption vary greatly for the testing platforms, particularly between consumer and server grade hardware (Table 5). For instance, a single Core i7

2600 is 12 × less expensive and requires 59% less energy than a dual Xeon E5-2680 CPU, whereas GeForce GTX 980 is more than 5 requires 27% less energy than Tesla K20Xm. For that reason, in addition to evaluating a pure computational performance, I analyze the performance with respect to the energy consumption and hardware cost. GeForce GTX 980 systematically outperforms other computing platforms, for example, it gives a benefit of 6.5 × per dollar and 7.3× per watt compared to the reference D1 platform (Figure 2.12). This remarkable performance results from mapping massively parallel computations and data structure to the GPU architecture. According to vendor specifications, GeForce GTX 980 has a higher core utilization and better energy efficiency than the previous generation Tesla K20Xm. Its streaming multiprocessors have two-thirds of the number of scalar processors of Tesla K20Xm, yet the number of registers and the shared memory size are the same. Therefore, extra efforts were devoted to tune the CUDA docking kernel in order to take advantage of the abundant resources per scalar processor on GeForce GTX 980. The performance per dollar of K20Xm GPU is comparable to a server grade Xeon E5-2680 CPU and Xeon Phi 7120P, but it is 2 × lower than a consumer grade Core i7 processor. Due to advances in the semiconductor technology constantly improving the energy efficiency, the performance per watt of a server grade hardware (Xeon E5 CPU, Xeon Phi and K20Xm) is about twice as high as that for an inexpensive, yet two years older Core i7 processor.

2.4.5 CASE STUDY

To demonstrate how GeauxDock samples the conformational space when searching for native conformations, in Figure 2.13, I present docking trajectories for several representative examples. In addition to the target complex 1a07 used in the profiling and benchmarking of parallel GeauxDock, I performed docking simulations of glutathione to glutathione S-transferase (PDB-ID: 1aqw) [71], and a non-peptidyl, active site-directed inhibitor LY178550 to human-thrombin (PDB-ID: 1d4p) [72]. Docking ligands were initialized at random orientations within target binding pockets to mimic a real application,

where the native conformations are unknown. Solid lines in Figure 2.13A show the trajectories of the pseudo-energy E1, E2 and E3 for 1a07, 1aqw and 1d4p, respectively. In all cases, the MMC sampling reached low-energy states with the fastest convergence for E3. On the other hand, pseudo-energy variations for E1 and E2 are smaller compared to E3, suggesting that the underlying energy surfaces for 1aqw and 1d4p are smoother.

In general, the convergence of molecular docking simulations is complicated by the fact that a large fraction of the search space may be sterically forbidden [54] and sophisticated scoring functions are often too sensitive to conformational changes in the binding regions [73]. To further investigate docking trajectories, I calculated the Contact Mode Score (CMS) for each accepted MMC step during the docking process of 1a07. CMS is a contact-based measure to assess the native-likeness of ligand binding poses, ranging from 1 for the exact native conformation down to about 0 for random configurations [60]. Encouragingly, the dashed black line in Figure 2.13A shows that the CMS increased as the pseudo-energy decreased owing to the fact that both quantities are strongly inversely correlated (Figure 2.13B). Altogether, these results demonstrate that the scoring function in GeauxDock effectively drives docking simulations toward native-like conformations.

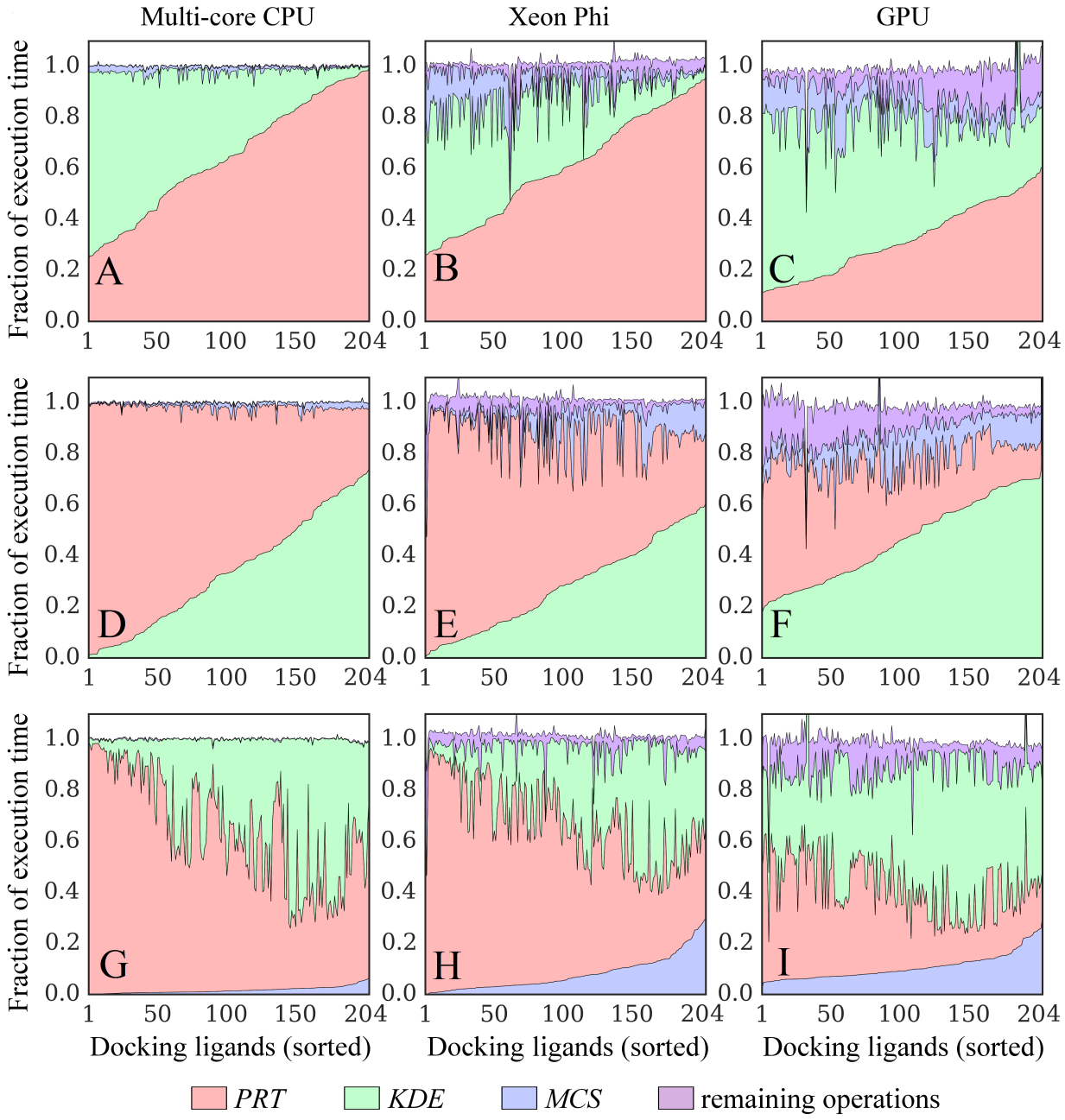

Figure 2.10: Time breakdowns for docking kernels running on different platforms. Kernel implementations for (A, D, G) multi-core CPU, (B, E, H) Xeon Phi, and (C, F, I) GPU are tested. Three major operations compute the following interaction matrices: proteinColumnVector × ligandRowVector (PRT, green), KDEColumnVector × ligandRowVector (KDE, red), and MCSMatrix × ligandColumnVector (MCS, blue). Purple areas correspond to the remaining operations. KDE (Kernel Density Estimation) and MCS (Maximum Common Substructure) points are used to calculate evolution-based components of the docking force field, whereas the PRT matrix is used to calculate the majority of physics-based potentials. Results collected for the dataset of 204 CCDC/Astex compounds are sorted on the x-axis with respect to increasing time of computing (A, B, C) PRT, (D, E, F) KDE, and (G, H, I) MCS matrices.

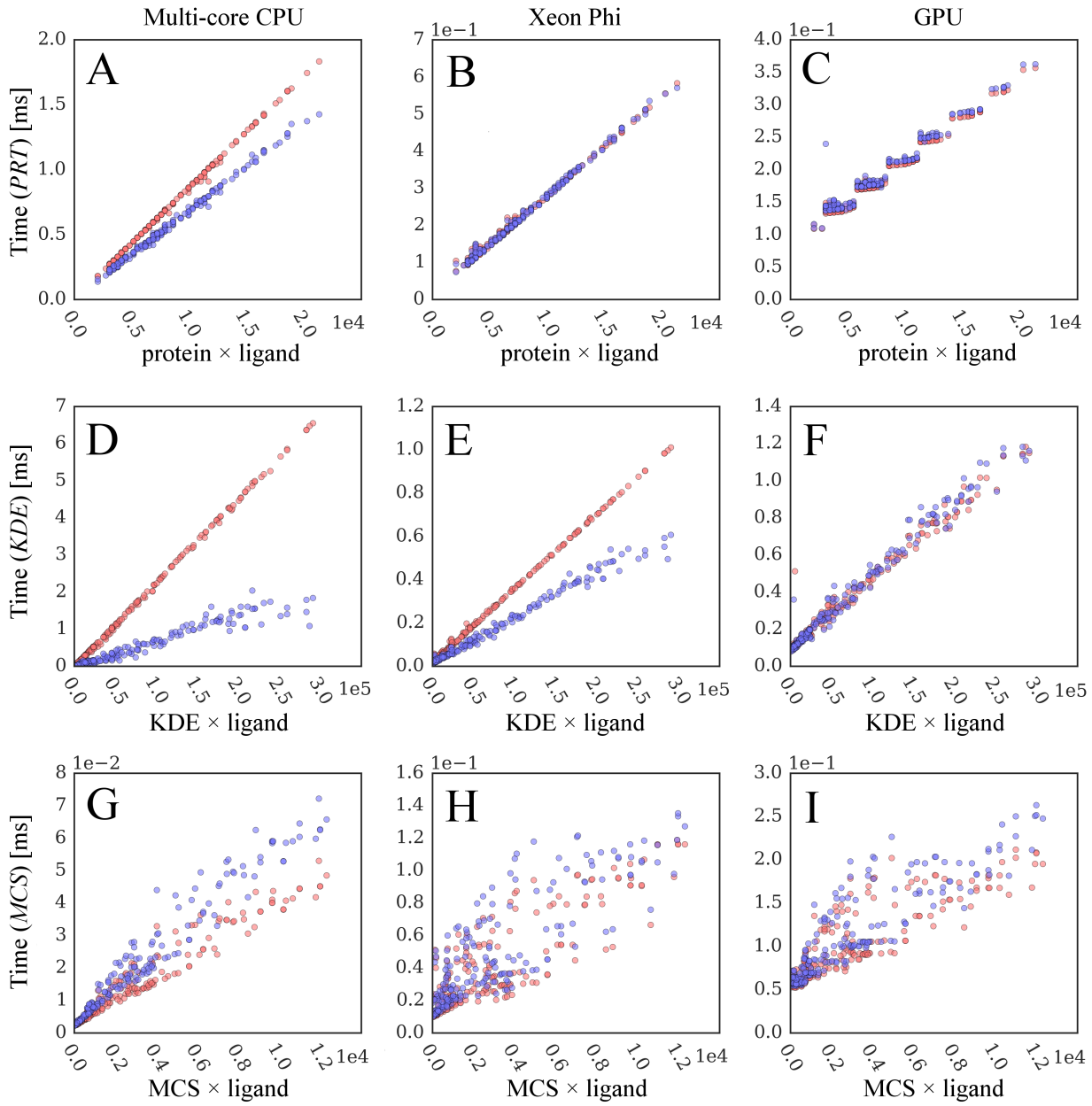

Figure 2.11: Correlation between computing time and static data size. Blue points are collected from original GeauxDock, whereas red points correspond to a modified docking code, where dynamic branches are turned off forcing the execution of all instructions. Three major operations compute (A-C) proteinColumnVector × ligandRowVector (PRT), (D-F) KDEColumnVector × ligandRowVector (KDE), and (G-I) MCSMatrix × ligandColumn-Vector (MCS) matrices. Three kernel implementations are tested for (A, D, G) multi-core CPU, (B, E, H) Xeon Phi, and (C, F, I) GPU.

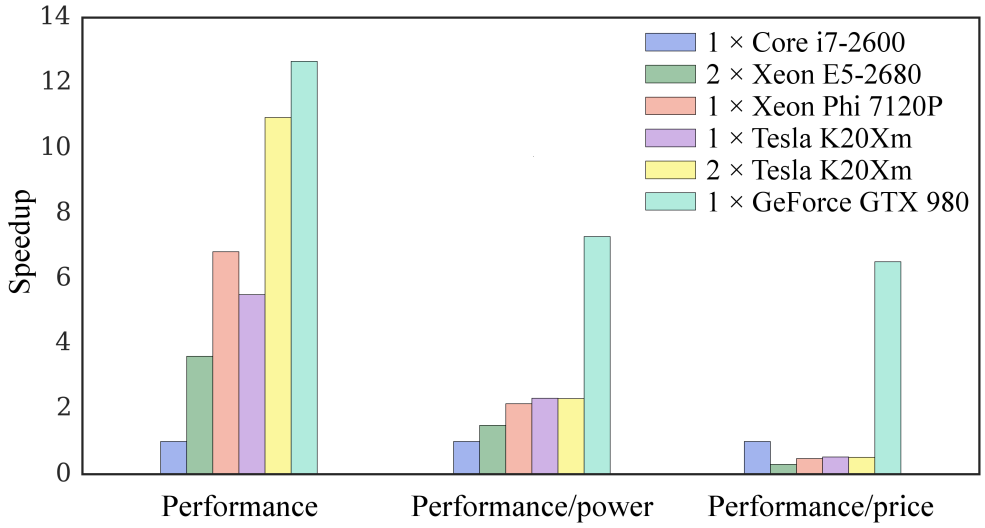

Figure 2.12: Benchmarks of GeauxDock against the dataset of 204 CCDC/Astex compounds using 6 platforms. Three measures are included, a pure computational performance, the performance divided by the energy consumption, and the performance divided by the hardware cost. Measurements for different platforms are normalized by the performance of Core i7-2600 CPU.

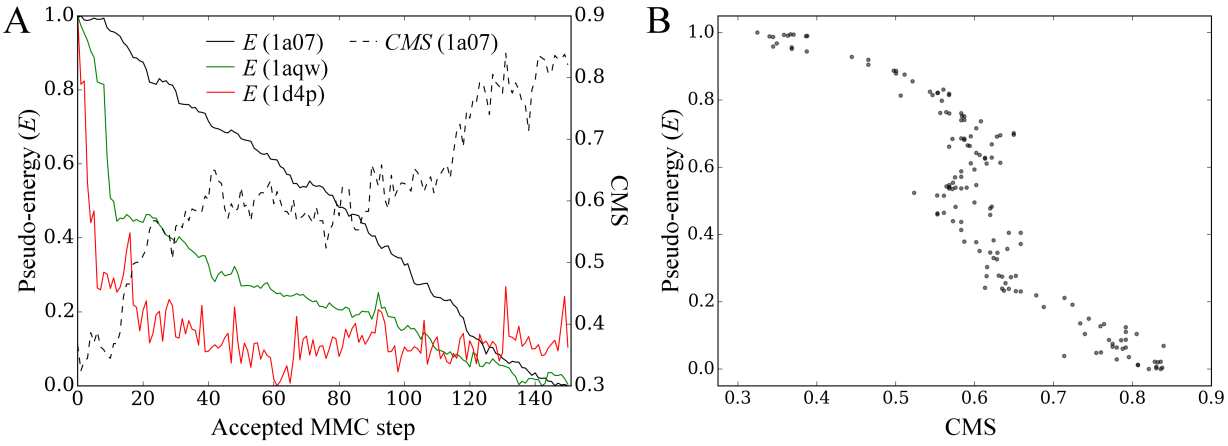

Figure 2.13: Examples of docking calculations using GeauxDock. Three cases are presented, a peptide ligand and C-src tyrosine kinase (PDB-ID: 1a07, black), glutathione and glutathione S-transferase (PDB-ID: 1aqw, green), as well as LY178550 and human-thrombin (PDB-ID: 1d4p, red). (A) Solid lines show the pseudo-energy plotted as a function of the accepted Metropolis Monte Carlo (MMC) step; a trajectory of the Contact Mode Score (CMS) is plotted for 1a07 (dashed black line). (B) Scatter plot of the CMS and pseudo-energy for 1a07.

Chapter 3
Performance Modeling

Heterogeneous processors perform differently on different data input data. Take GeauxDock computing Astex data set as an example. For this case, a Dual socket Xeon E5 CPU is generally the slowest. A Tesla K20Xm GPU offers 1.3 × speed-up on average, and a Xeon Phi 7120P yields 1.9× improvement on average (Figure 2.12). However, the average speedups on a bunch of tasks do not reliably reflect the speedup on a particular individual task due to the significant fluctuations. Figure 3.1 shows that each of the three kinds of processors could perform significantly better or worse depends on different inputs.

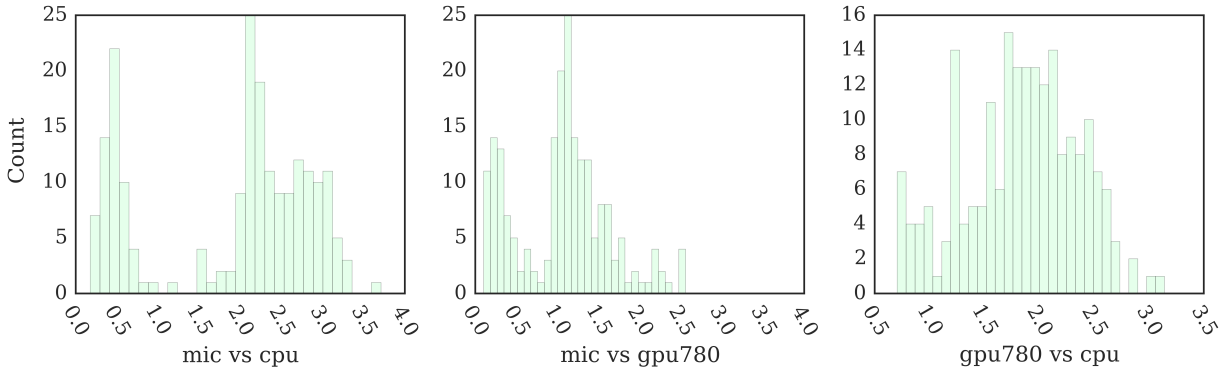

Figure 3.1: Distribution of relative performance of GeauxDock for the dataset of 204 CCDC/Astex compounds. Three kernel implementations are tested for multi-core CPU, Xeon Phi and GPU. Relative performance between there three platfroms are plotted. (A) Xeon Phi vs. multi-core CPU, (B) Xeon Phi vs. GPU, and (C) GPU vs. CPU

Understanding such performance trends helps utilize heterogeneous processors. With performance prediction, it will be possible for an optimizing scheduler to match tasks to processors in a heterogeneous system so that strength and weakness of different processors can be leveraged. In this chapter of my thesis, I will be using regression models to achieve this goal. Precise time consumption of a given task will be predicted before running.

3.1 COARSE GRAINED MODEL

In section 2.4, I study some performance characteristics of Geauxdock by running it with synthetic input data. I use a single conformation and replicated it n times to strictly control the number of replicas in the docking simulation. Figure 3.3A-C shows the performance versus the number of replicas on a multi-core CPU, Xeon Phi and GPU, respectively. It reflects the resource utilization. Utilizations are very high when the number of replicas is a multiple of 20, 240 and 14, respectively, for each of three platforms, CPU, Xeon Phi and GPU. Otherwise, on the CPU (Figure 3.3A) and GPU (Figure 3.3C), one suffers linear performance drop. Xeon Phi co-processor follow this overall trend, but the performance is higher when the utilization is really low (Figure 3.3B). This is attributed to resource contention as I had explained in Section 2.4.2.

The performance pattern in Figure 3.3 can be viewed from another angle, where the Y-axis (Figure 3.2) represents per-complex computing. This representation shows a better understandable and predictable pattern. CPU and GPU time patterns in Figure 3.2A,C are sets of horizontal lines. The pattern for Xeon Phi (Figure 3.2B) is slightly irregular, but is still predictable.

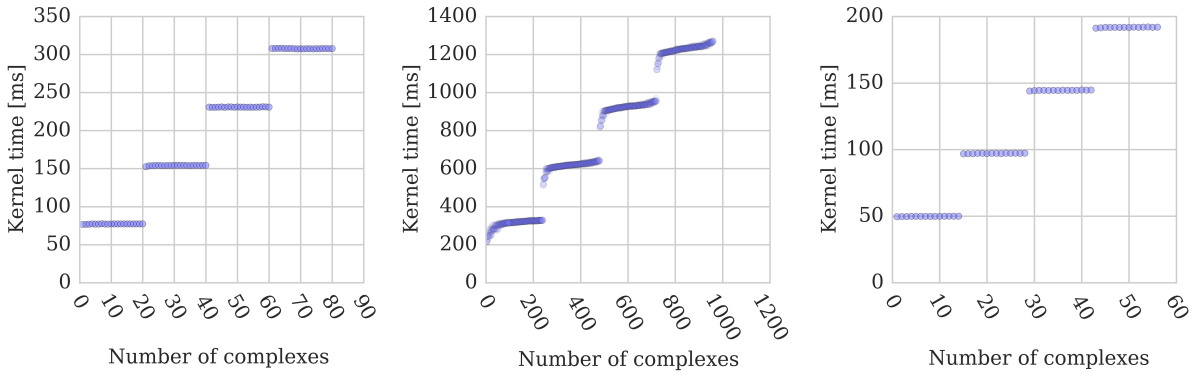

Figure 3.2: Coarse gained performance scaling, with computing time as the Y axis.

Comparing Figure 3.3 and Figure 3.2, the former has many flat lines. More importantly, the values are non-decreasing. Thus the pattern in Figure 3.3 is mathematically simpler, and could be something that a machine learning algorithm can learn with higher confidence,

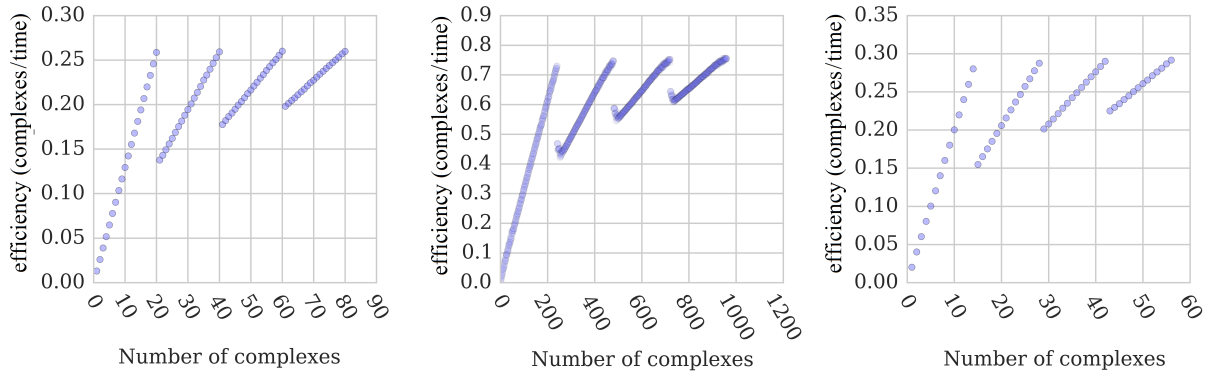

Figure 3.3: Coarse gained performance scaling, with computing time divided by the number of complexes as the Y axis.

and I will be trying to predict this pattern. The problem of performance prediction is defined as follows.

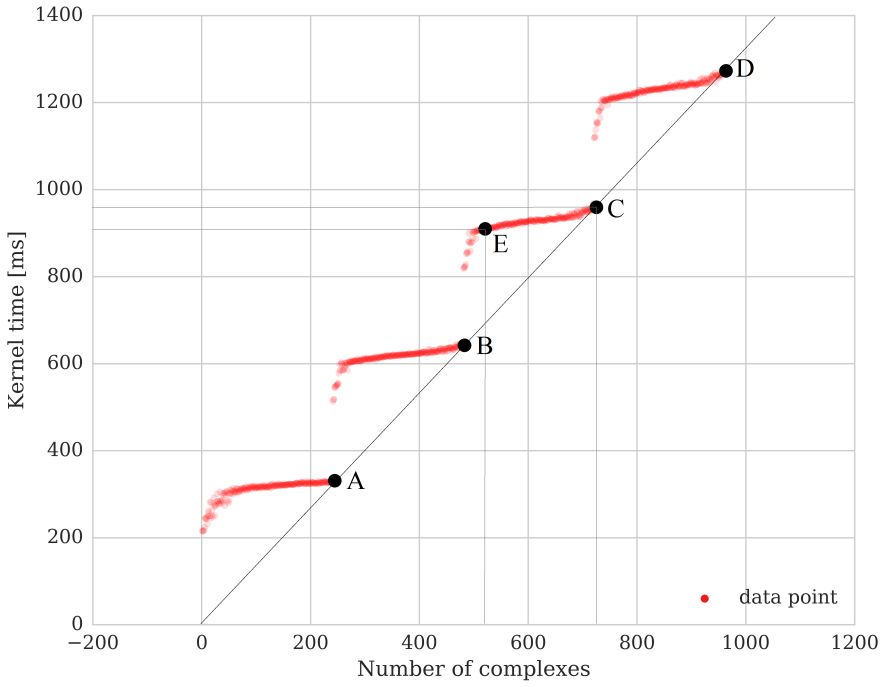

Figure 3.4: Defining the performance prediction problem on the coarse level

Giving a data point $E \in \mathbb{R}^2$, the value on the X axis is defined as E_x and is available. The value on the Y axis, denoted as E_y, is to be resolved. Here E_x represents the number of complexes and E_y represents the execution time. To make this prediction useful, I define a

reference point C, which is the right end of the cyclic pattern in Figure 3.4. The processor is underutilized at point E and fully utilized at point C. In other words, at point C, the processor is on the highest efficiency. The performance at data point C can be learned using linear regression, which I will discuss in Section 3.2. To calculate the performance at data point C, The coefficient E_{eff} is defined in Equation 3.1 to describe the execution efficiency on data point E as follows:

$$E_{eff} = \frac{\frac{E_{workload}}{E_{time}}}{\frac{C_{workload}}{C_{time}}} = \frac{E_x}{C_x} \times \frac{C_y}{E_y}. \tag{3.1}$$

It is easy to calculate E_{eff} for CPUs and GPUs, considering the regular patterns in Figure 3.2 A,C. E_y and C_y should be is equal all the time. In the meantime, the relationship of E_x and C_x can be expressed using modulo operators. The mathematical expression is shown in Equation 3.2

$$\begin{aligned} n &= 20(onCPU), or 14(onGPU) \\ C_x &= E_x(\mod n) \times n \\ C_y &\equiv E_y \\ E_{eff} = \frac{E_x}{C_x} \times \frac{C_y}{E_y} &= \frac{E_x}{E_x(\mod n) \times n} \end{aligned} \tag{3.2}$$

To resolve the performance pattern for Xeon Phi, (Figure 3.2 B), isotonic regression (IR) is used. IR fits a non-decreasing function to data. It corresponds to the following quadratic programming (QP) problem (Equation 3.3), where $x \in \mathbb{R}^n$, and $a \in \mathbb{R}^n$ is the vector to be fitted. I carried out experiments using `sklearn.isotonic.IsotonicRegression` provided in the Python *scikit-learn* package. The training data set ($a \in \mathbb{R}^n$) is the set of 960 samples for replicated synthetic data (Figure 3.2B). The testing data set is the unmodified 204 data samples presented in Section 2.4.4. Figure 3.5 demonstrates that isotonic

regression captures the pattern well. The training error is small, since the prediction (green points) closely follow the trend of the input data (red points). The test data (blue points) reflect the patterns of the training data. As these points generally overlap, it demonstrates that the isotonic regression algorithm works well for predicting the performance on Xeon Phi.

$$\begin{aligned} &\min \sum_{i=1}^{n} w_i (x_i - a_i)^2 \\ &\text{subject to } x_i \geq x_j \quad \text{for all } (i,j) \in E \end{aligned} \tag{3.3}$$

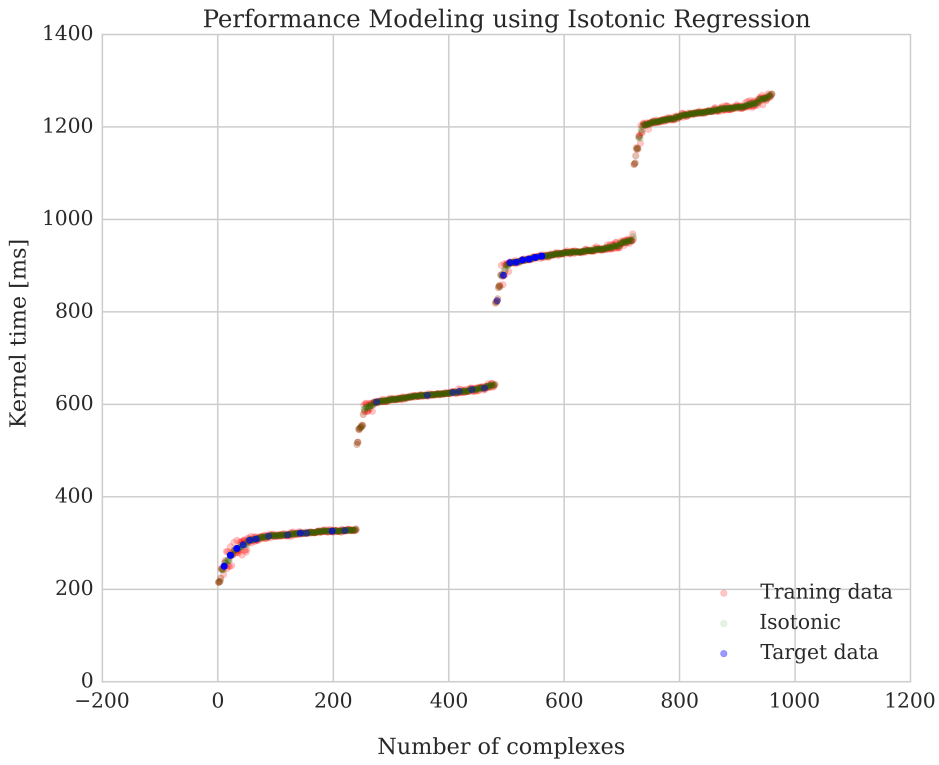

Figure 3.5: Using isotonic regression to fit the Xeon Phi performance patterns.

$$
\begin{aligned}
C_x &= E_x(\mod n) \times n \\
C_y &= IsoRegressor(C_x) \\
E_y &= IsoRegressor(E_x)
\end{aligned}
\tag{3.4}
$$

$$
E_{eff} = \frac{E_x}{C_x} \times \frac{C_y}{E_y} = \frac{E_x}{E_x(\mod n) \times n} \times \frac{IsoRegressor(E_x(\mod n) \times n)}{IsoRegressor(E_x)}
$$

3.2 FINE GRAINED MODEL

In this section, I will be resolving the performance model on the fine grain level. To simplify the analysis, I feed the machine learning model with data that always has sufficient coarse grained parallelism, i.e., data points like A, B, C and D in Figure 3.4

To review the computation in Monte Carlo kernels, three matrices are computed in the Monte Carlo kernel: $protein_{ColumnVector} \times ligand_{RowVector}$ (P), $KDE_{ColumnVector} \times ligand_{RowVector}$ (K), and $MCS_{Matrix} \times ligand_{ColumnVector}$ (M). All of the computations are subject to fine-grained parallelization. In Section 2.4.3, I conduct measurements trying to isolate the computation time for P, K and M. The corresponding time is denoted as T_P, T_K, T_M. I also calculate the remaining time $T_R = T_{walltime} - T_P - T_K - T_M$ Figure 2.11 shows that T_P, T_K and T_M positively correlate with the size of the matrix. Though the matrix is sparse, and I have not considered the valid data points or the data pattern, the data presents a fairly strong linear model. The R^2 for the unmodified $P(K,M)$ code shown in blue in Figure 2.11G-I is 0.996(0.938,0.957) for CPU, 0.996(0.987,0.720) for Xeon phi, and 0.952(0.981,0.792) for GPU. It is useful to fit the execution time with the matrix size using linear models. However, the matrix size can be regulated using architectural features, and therefore improve the linear correlation.

The performance of computing P on GPU (Figure2.11C) shows a pattern similar to what I have observed in coarse grained parallel performance scaling pattern (Figure 3.3). It implies that GPU's fine grained parallel execution resource is not always fully occupied.

Since I implement parallel execution on both the row axis and column axis of the matrix, the actual computation cycle should be regulated by the amount of parallelism in the execution (Equations 2.2 and 2.3) Also, compiler loop transformations such as loop unrolling could reduce the execution cycles on vector processors. To consider these facts, I am trying to fit the data with different vector length on both X and Y axes. ($size\prime$ in Equation 3.5).

$$\begin{aligned} size_x\prime &= size_x \,(\mod tile_x) \\ size_y\prime &= size_y \,(\mod tile_y) \\ size\prime &= size_x\prime \times size_y\prime \end{aligned} \tag{3.5}$$

To demonstrate how the parameter $tile_x$ and $tile_y$ affect the expected linear pattern of computing time versus tile size, I regulated the GPU performance measurements (Figure 2.11C,F,I). The results are visualized using the scatter plot in Figures 3.6, 3.7 and 3.8. These figures demonstrate that some patterns result in stronger linear correlations.

Then, using heat map to plot the R^2 score versus different tile size, the quantified scores of linear correlation is shown in Figures 3.9, 3.10, 3.11, 3.12, 3.13, 3.14, 3.15, 3.16, and 3.17. For the task of P matrix computing (Figures 3.9, 3.10, and 3.11), the tiling effect is insignificant. GPU $tile_y$ shows a better score at size 8, which agrees with the GPU thread level parallelism on the Y axis. The K matrix heat maps (Figures 3.12, 3.13, and 3.14) are more interesting. The best shape of GPU tile is 128 $\times$ 8, perfectly matching the shape of GPU thread. For CPU and Xeon Phi, the optimal tile shapes are both 64 $\times$ 4. This implies the compiler may have adopted the same loop transfers for folding both the x and y dimensions of the matrix, and generated vectored code. The M matrix heat map (Figures 3.15, 3.16, and 3.17) show the optimal value for $tile_x$ is 64 for all processors. This is because the size of x dimension is indeed smaller than 64. The code is also successfully vectorized, so any of the heterogeneous processors could compute a row in a cycle. There is no distinguished value for $tile_y$. Any value range form 1 to 32 is equally good. Finally

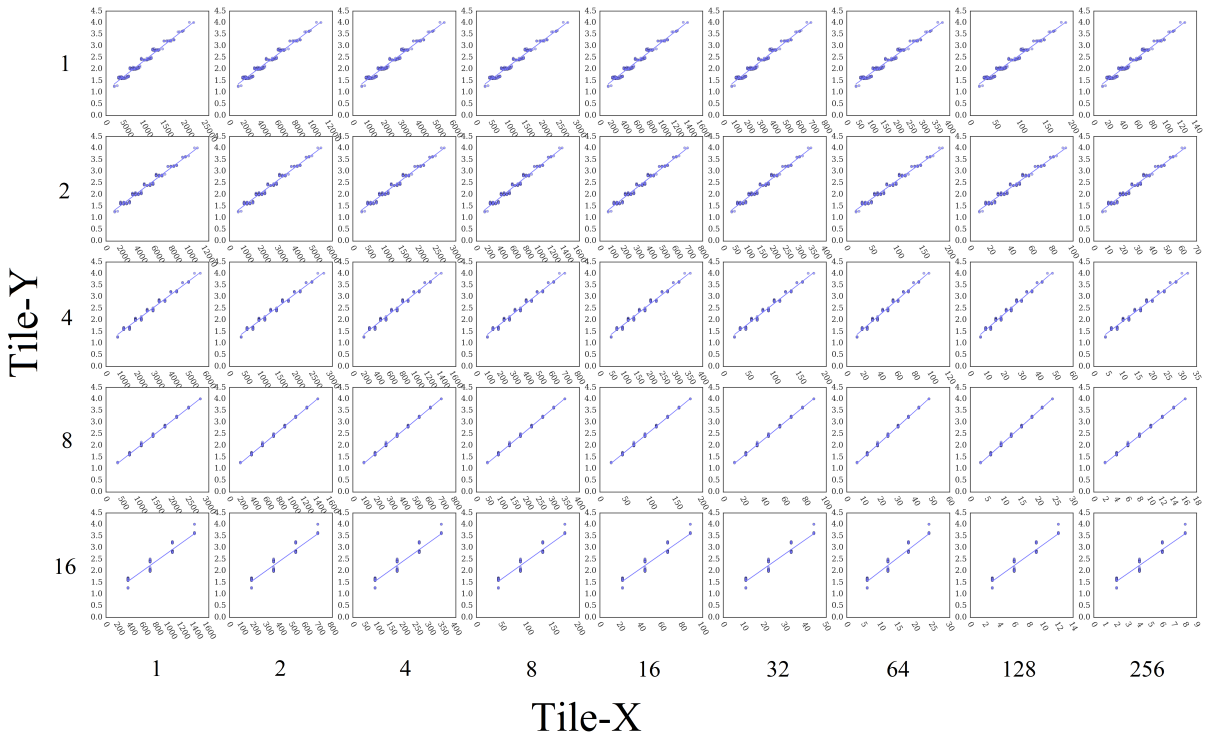

Figure 3.6: The patterns of GPU performance on P matrix versus different tile sizes the best tile sizes are concluded in Table 3.1

	CPU $tile_x$	CPU $tile_y$	Phi $tile_x$	Phi $tile_y$	GPU $tile_x$	GPU $tile_y$
PRT	1	1	1	1	1	8
KDE	64	4	64	4	128	8
MCS	64	1	64	1	64	1

Table 3.1: Optimal values of $tile_y$ and $tile_x$, that minimize R^2 for the linear fitting.

It is encouraging to use a general linear regression model (Equation 3.6) to resolve the compute time of a data point C, denoted as C_y. (see Figure 3.4). This data point has sufficient coarse grained replicas (Section 3.1). The values $size_{P'}$, $size_{K'}$ and $size_{M'}$ are the regulated data sizes (Equation 3.5) using the optimal tile parameters (Table 3.1). The data is trained using 3 fold cross-validation. Table 3.2 shows the R^2 across three processor. The quality of fitting is improved after applying the tiling regulation. The GPU platform shows the most significant improvement because it offers the highest amount of fine grained parallelism. The fitted parameters are listed in Table 3.3

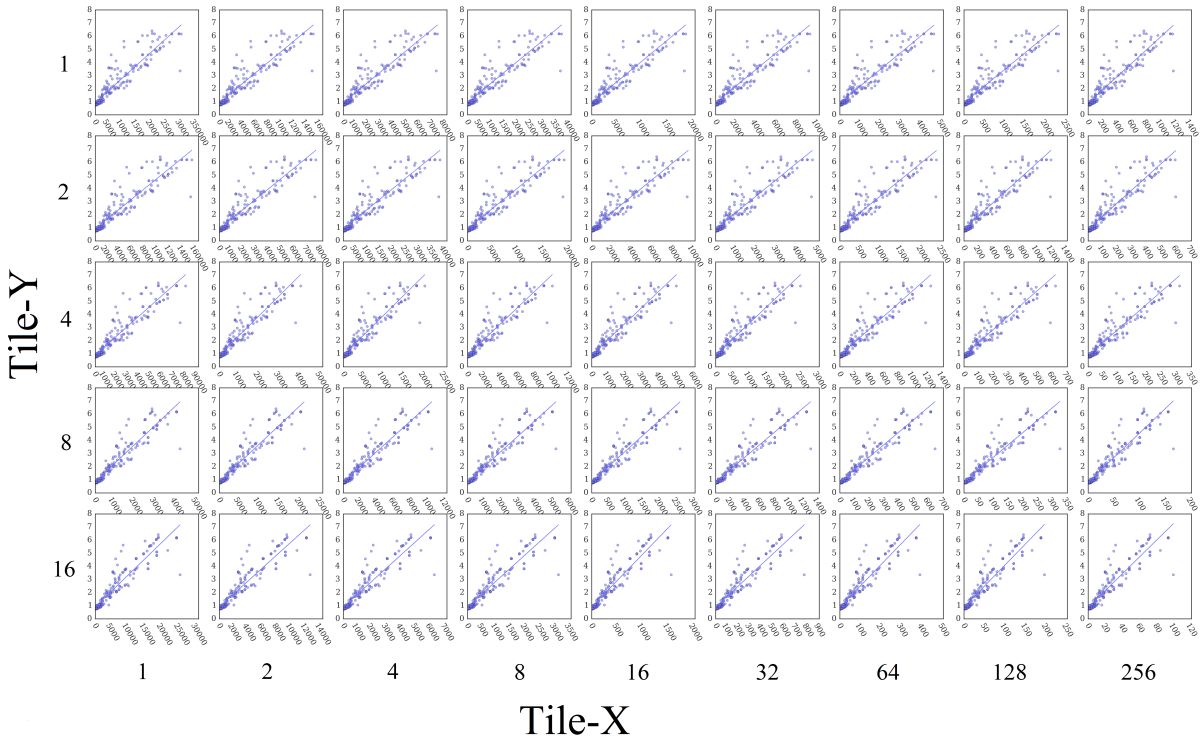

Figure 3.7: The patterns of GPU performance on K matrix versus different tile sizes

$$C_y = w_1 \times size_{P'} + w_2 \times size_{K'} + w_3 \times size_{M'} + c \tag{3.6}$$

	CPU	Xeon Phi	GPU
without using regulation	0.9638	0.9763	0.9163
with optimal regulation	0.9691	0.9817	0.9660

Table 3.2: Comparing the general liner regression fitting score R^2 before and after applying the tiling regulation.

	w_1	w_2	w_3	c
CPU	0.000764	0.01249	-0.000956	-0.481161
Xeon Phi	0.000239	0.001799	0.0018422	0.257347
GPU	0.001331	0.015900	0.010910	1.026985

Table 3.3: Fitted parameters of the general linear regression model (Equation 3.6)

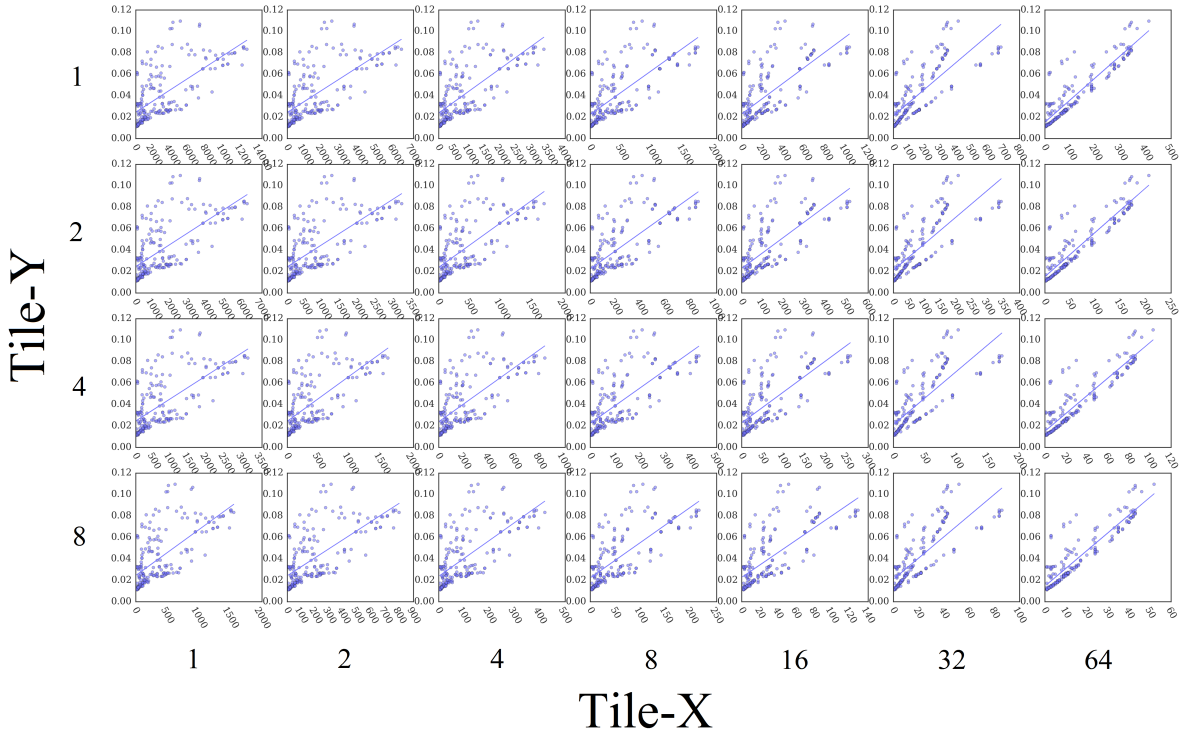

Figure 3.8: The patterns of GPU performance on M matrix versus different tile sizes

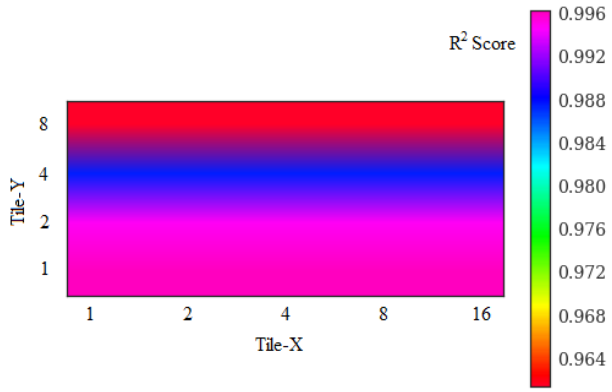

Figure 3.9: Visualizing the R^2 scores under different $tile_x$ (x-axis) and $tile_y$ (y-axis) Computing P matrix on CPU.

3.3 RESULTS

Now I combine the coarse grained performance model (Section 3.1) with the fine grained performance model (Section 3.2). Specifically, the linear model (Equation 3.6) predicts the computing time C_y for data set with sufficient coarse-grained parallelism. This data point

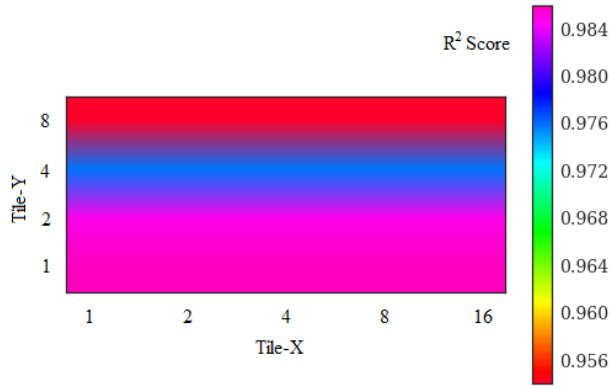

Figure 3.10: Visualizing the R^2 scores under different $tile_x$ (x-axis) and $tile_y$ (y-axis) Computing P matrix on MIC.

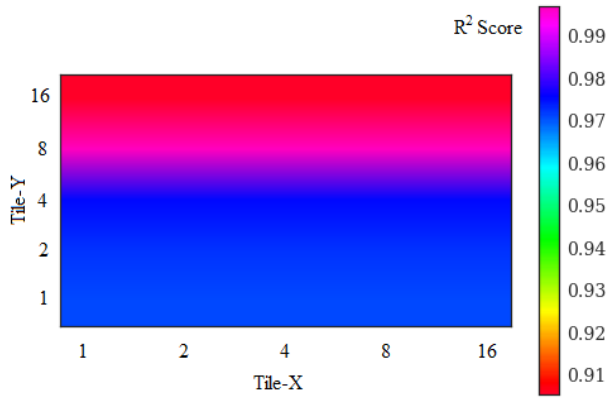

Figure 3.11: Visualizing the R^2 scores under different $tile_x$ (x-axis) and $tile_y$ (y-axis) Computing P matrix on GPU.

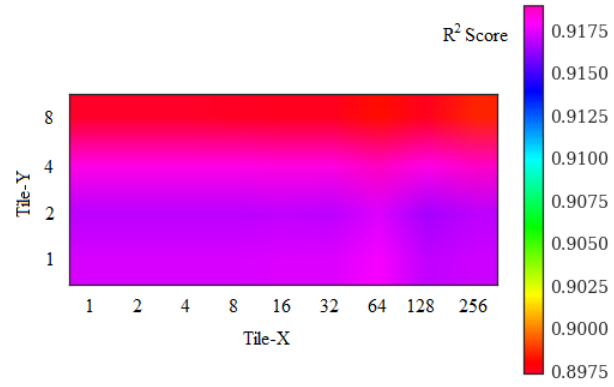

Figure 3.12: Visualizing the R^2 scores under different $tile_x$ (x-axis) and $tile_y$ (y-axis) Computing K matrix on CPU.

refers to the rightmost point (C for example) of one of the horizontal bars in Figure 3.4.

The coarse grained model (Equation 3.2 or 3.3) provides the slow down coefficient E_{eff},

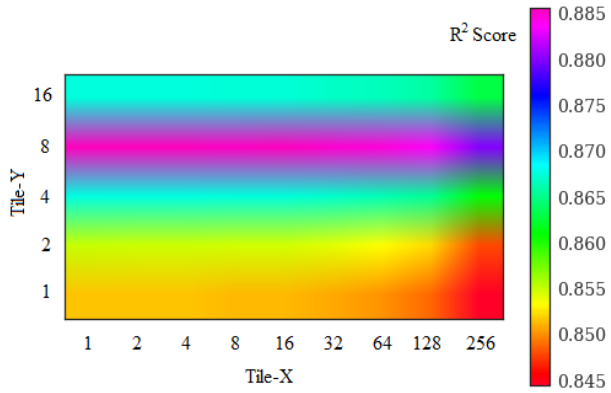

Figure 3.13: Visualizing the R^2 scores under different $tile_x$ (x-axis) and $tile_y$ (y-axis) Computing K matrix on MIC.

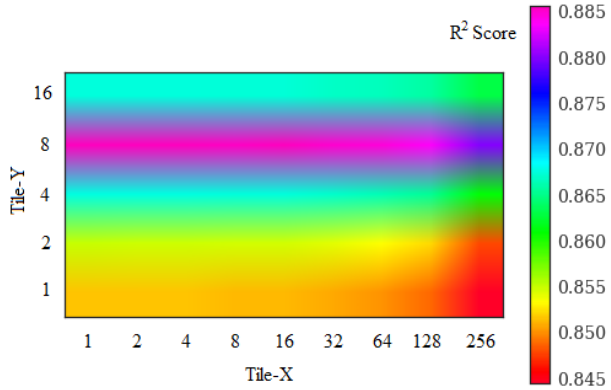

Figure 3.14: Visualizing the R^2 scores under different $tile_x$ (x-axis) and $tile_y$ (y-axis) Computing K matrix on GPU.

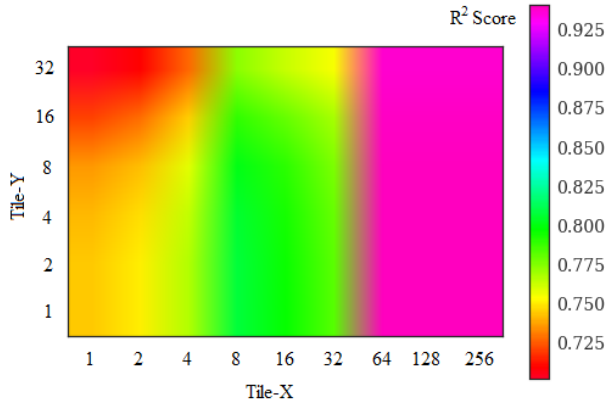

Figure 3.15: Visualizing the R^2 scores under different $tile_x$ (x-axis) and $tile_y$ (y-axis) Computing M matrix on CPU.

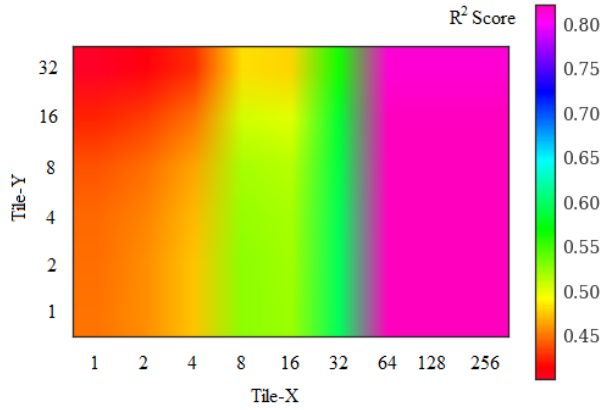

Figure 3.16: Visualizing the R^2 scores under different $tile_x$ (x-axis) and $tile_y$ (y-axis) Computing M matrix on MIC.

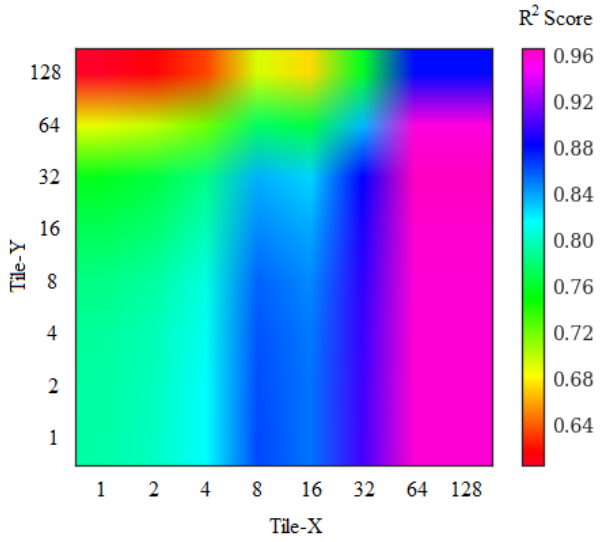

Figure 3.17: Visualizing the R^2 scores under different $tile_x$ (x-axis) and $tile_y$ (y-axis) Computing M matrix on GPU.

for data point E, where E may not have sufficient coarse grained parallelism as data point C. The compute time E_y is available using Equation 3.7

$$E_y = \frac{C_y}{E_{eff}} \tag{3.7}$$

Using Equation 3.7, the performance prediction results from coarse grained performance model (Section 3.1) and fine grained performance model (Section 3.2) are combined. The

predicted valued is compared against the actual value. The R^2 core is shown in Table 3.4

R^2	CPU	Xeon Phi	GPU
	0.974	0.994	0.980

Table 3.4: The R^2 score of the comprehensive performance prediction

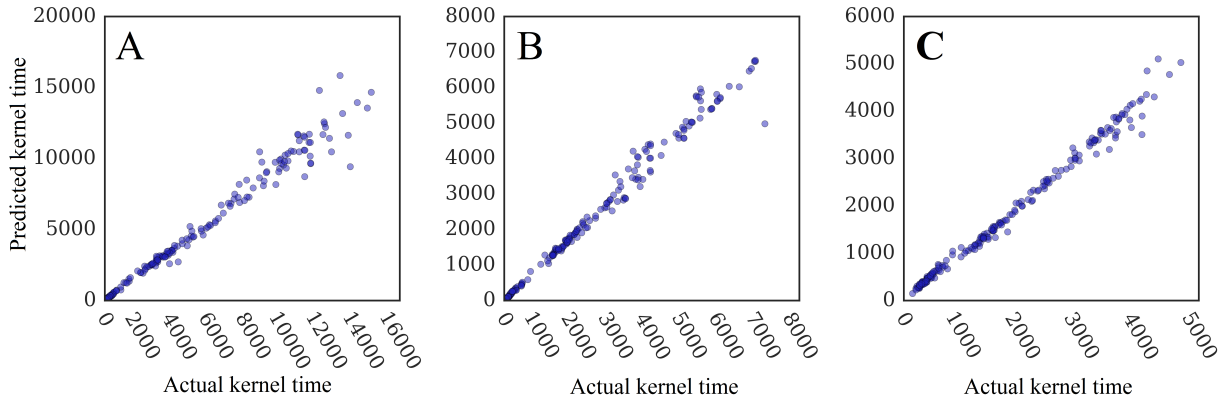

Figure 3.18: Correlation between the predicted and actual execution time for (A) multi-core CPU, (B) Xeon Phi, and (C) GPU.

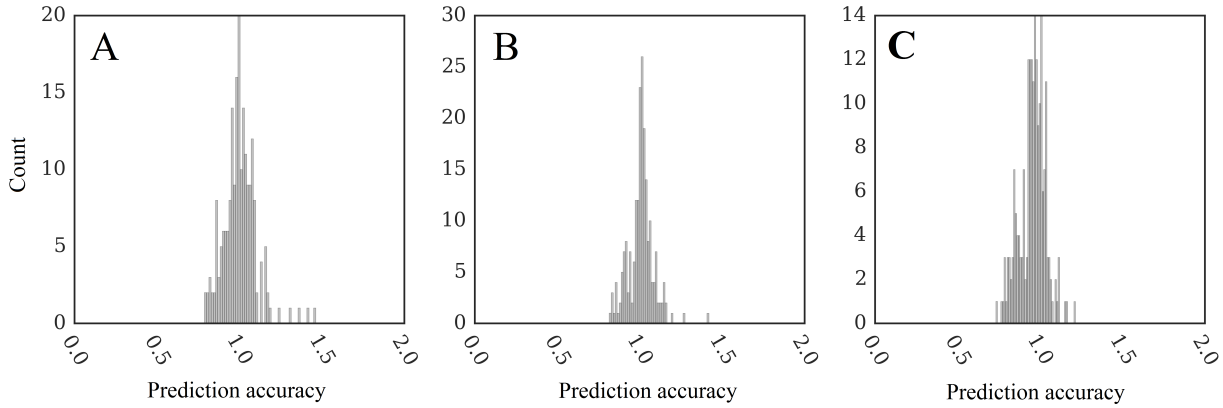

Figure 3.19: The histogram plots show the error of the execution time prediction versus the actual value. The three sub-figures are for (A) multi-core CPU, (B) Xeon Phi, and (C) GPU.

Chapter 4
Task Scheduling

4.1 BACKGROUND

Scheduling is one of the essential aspects of achieving high levels of performance and low energy consumption on heterogeneous systems. The case for efficient approaches to scheduling a set of independent tasks on a set of heterogeneous worker processors is indeed a common issue in high performance computing. The problem of independent task scheduling is defined as the following optimization problem in Equation 4.1.

$$\begin{aligned} \min_{x} \quad & makespan = \max_{1 \le i \le W} \sum_{j=1}^{T} (x_{ij} \times t_{ij}) \\ \text{s.t.} \quad & \sum_{j=1}^{W} x_{ij} \;=\; 1 \qquad\qquad \forall x \in \{0,1\}^{W \times T} \end{aligned} \tag{4.1}$$

A pool of T independent tasks is scheduled to run on W worker processors. The execution time of assigning task i to worker j is t_{ij}. An assignment is label by $x_{ij} = 1$ in the binary matrix $X \in \{0,1\}^{W \times T}$. If x_{ij} is 0, it means task i was not assigned to worker j. The execution time of worker j is $\sum_{j=1}^{T}(x_{ij} \times t_{ij})$. The makespan is defined as the maximum execution time among all workers, and should be minimized.

Scheduling problems can be classified by the properties of the task and the properties of the workers. The simplest case of homogeneous tasks for homogeneous worker scheduling has been discussed in few classic algorithms. *List scheduling* assigns the task to the machine whose load is the lowest. Scheduling with the *Longest Processing Time* rule (LPT) sorts the tasks in decreasing order of execution time before the assignment and effectively improves load balancing. *Work stealing* [74] allows migrating workload between workers when starvation occurs. Projects implementing homogeneous scheduling includes the Cilk [75] parallel programming language for CPUs. StarSs [10] is a library and runtime system that partitions a large computational workload for multiple symmetric computational re-

sources. Multiple CPUs, GPUs, and Sony's Cell co-processors are supported, but not all at the same time.

The seconds important problem is that of scheduling homogeneous tasks onto heterogeneous workers and must account for the differences between workers. Since an average performance metric can reliably represent performance patterns, from the efficiency point of view, it does not matter whether a processor chooses one task over another. Therefore, scheduling algorithms for this problem focus on deriving more workload-balanced (or, balanced) partitions. Examples of this model include StarPU [11]. A queue of tasks are assigned to CPUs and GPUs from both end of the queue. Anthill [12] implements similar features using queues and events.

Scheduling for heterogeneous tasks onto a set of heterogeneous workers has received attention in the distributed computing research community. Various algorithms have been proposed in this area. Linear programming (LP) [76] give precise optimal solution on small size input data, but rarely scales to more than 20 tasks. Search algorithms explore the solution space, trying all values of all parameters. Branch-and Bound search (BB) [77] terminates the search when it concludes that the currently explored path is sub-optimal. The performance of BB is still too slow for realistic large data sets. Monte Carlo Simulated Annealing (SA) and Genetic Algorithms (GA) are used to randomly sample the solution space. Tabu Search [78] and A^* [79] are similar iterative search algorithms.

Several heuristics have been proposed in the literature to offer lower cost scheduling solutions. Opportunistic Load Balancing (OLB) assigns tasks in arbitrary order to the next available machine. This is essentially list scheduling and does not consider either the task heterogeneity or the worker heterogeneity. Minimum execution time (MET), also known as Limited Best Assignment (LBA) or User Directed Assignment(UDA), assigns tasks to the fastest machine regardless of the machine's availability. Minimal Completion Time (MCT) assigns tasks to machine with the minimum expected completion time (CT). By definition, the completion time is the summation of the execution time and the queueing time. MET

and MCT concentrate on two aspects of the scheduling problem: efficiency matching and load balancing. Efficiency matching means a task is mapped to the fastest machine. Load balancing means minimizing the machine idle time. MET and MCT algorithms stand on opposite sides of the optimization goal. Switching algorithm [80] combines both MET and MCT, and makes the switch between them periodically. Min-min [81] heuristic uses MCT as the performance metric. It gives high priority to the task that can be completed at the earliest. This heuristics begin with the pool of unmapped tasks. Before issuing a new task, it scans through the task pool and compute the MCTs of tasks in the pool. The task with the lowest CT is assigned to the corresponding worker. This procedure repeats until the task pool is empty. Max-min [81] heuristic is very similar to Min-min and the metric used is also MCT. It also searches all waiting takes for the minimal CT. However, the task with the highest score is selected. Suffrage [82] heuristic also scans all candidate tasks in the pool before assigning a new task. Its scoring function is the suffrage value, which is defined as the difference between the best CT and the second best CT. The intuition is that a larger suffrage value implies a higher relative performance.

A survey paper [83] concludes that OLB, MET, Max-min, SA, and Tabu Search do not produce good schedules in general. Min-min, GA, and A* are good, however, the differences are usually within 5%. A* produces better or worse results than Min-min for different cases. The GA implementation is seeded with the results from Min-min, thus is always slightly better than Min-min. The drawback of GA is the speed. In their experiments, scheduling 512 tasks on 16 works requires 1 second computation time using Min-min, 100 seconds using GA, and 1,200 seconds using A*.

Most of the previously proposed heuristic algorithms, with the exception of OLB and work sealing, decouple the scheduling and execution aspects. The schedule table is computed ahead of the execution of the tasks. The runtime system strictly follows the arrangement in the scheduling table, and cannot apply any modifications. This implies that the execution times of tasks must be perfectly predicted. Otherwise, the schedule might be

sub-optimal. The complexities of the existing heuristic algorithms are still very high. The Min-min, Max-min and Suffrage algorithm fall into the category of the two-round search paradigm. To schedule a pool of T tasks on W workers, every newly issued task requires the scheduler to scan through the entire combination, which has the complexity of $\mathcal{O}(WT)$. The complexity of the whole schedule is $\mathcal{O}(WT^2)$. The two-round search algorithms will be too slow when the number of tasks is huge.

4.2 RANKED OPPORTUNISTIC BALANCING (ROB)

One important component of my algorithm is *Ranked Opportunistic Balancing* (ROB). This scheduling happens in a self-organized manner [84]. Both the work sharing model and the competition method are deployed. The worker actively fetches a new task when it becomes free. The newly fetched task must be suitable for the worker, and least little computation effort. I utilize the static queueing method for this. The tasks are sorted by the performance suitability scores. The exact scoring function and ranking method will be discussed in Section 4.4. The queue is implemented with linked list data structure (Figure 4.1A). Each node in the list represents a task. At the head of the list is the task with the highest score. When the head node is removed, the next node with the second highest score becomes the new head of the list. Different from traditional linked list data structure, where a node has only one previous pointer and one next pointer, the ROB linked list could hold multiple chains. Figure 4.1A demonstrates a set of tasks A, B, C and D are arranged in two linked lists. This can also be viewed in Figure 4.1B, where the tasks are organized as two queues. Each heterogeneous worker watches only the head node of its associated task queue(s). The run time complexity is of this scheme is extremely low.

One important issue in truly dynamic scheduling is the requirement of maintaining a synchronized global status. When a task is fetched by a worker, another worker must be able to see that this task is no longer available. This is implemented using a centralized master server, which maintains the status of the task queues. The workers send request

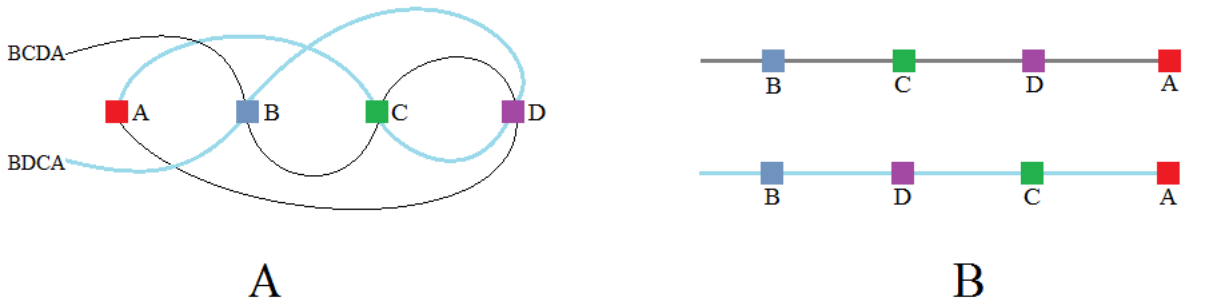

Figure 4.1: Arranging tasks in priority queues in the ROB algorithm.

messages to the server, and receive response messages which contain task information. In practice, the overhead of sending and receiving messages is low. For the typical scenarios where a task lasts for a few hundred milliseconds, the centralized management protocol will never become the performance bottleneck.

4.3 MULTI-SUBJECT RANKING (MR)

The scoring functions of the popular *Min-min* and *Suffrage* algorithms are also problematic. Min-min algorithm searches for a task that delivers the minimal CT. This intuition is trying to balance the CT at every step. A smaller CT does not reliably reflect the performance suitability. Suffrage algorithm tries to calculate the relative performance. However, the absolute difference between CTs is not an ideal metric.

Therefore, I propose *Multi-subject Ranking* (MR) heuristics to calculate the scoring function. The fundamental idea is based on the ratio between two ETs. (Equation 4.2)

$$S_{ij} = \frac{ET_i}{ET_j} \tag{4.2}$$

In the multiple worker heterogeneous environment, however, it is not straight forward to find a pair of data for fair comparison. For example if there were 3 heterogeneous workers, the ETs of the jobs are a set of triples. To calculate the performance suitability

score for worker 1, should we adopt $\frac{ET_2}{ET_1}$ or $\frac{ET_3}{ET_1}$? One solution is to use a synthetic value as the baseline. The choices include: **(1)** ET_{min}: minimal value of the task's ETs, **(2)** ET_{max}: maximum value of the task's ETs, or **(3)** ET_{avg}: average value of the task's ETs. Therefore, there are 3 variation of MR heuristics. (Equations 4.3)

$$\begin{aligned} \textbf{MRmin heuristic} : S_i &= \frac{ET_{min}}{ET_i} \\ \textbf{MRmax heuristic} : S_i &= \frac{ET_{max}}{ET_i} \\ \textbf{MRavg heuristic} : S_i &= \frac{ET_{avg}}{ET_i} \end{aligned} \tag{4.3}$$

4.4 MULTI-SUBJECT RELATIVE RANKING (MRR)

Intuitively, MR_{min} and MR_{max} heuristics do not seem to be fair. For example, the score of worker 1 executing task A is $S_A = \frac{ET_2}{ET_1}$, and the score of worker 1 executing task B is $S_B = \frac{ET_3}{ET_1}$. If $S_A > S_B$, can we conclude that task A is surely a better match? Is is really fair when A and B are compared against two references. To resolve this issue, I propose *Multi-subject Relative Ranking* (MRR). MRR is a relative scoring system. Let there be w types of heterogeneous workers. One worker holds a vector of $w-1$ scores. Each score is calculated (see Equation 4.2) by comparing this performance of this worker with another worker. Because every task now has $w \times (w-1)$ scores, the tasks are ranked into $w \times (w-1)$ queues. Each queue gives the fair order of the comparing the relative fitness between two workers. Now every worker will need to watch $w-1$ priority queues, and thus has $w-1$ tasks candidates. Which one should it choose? To answer this question, I propose 4 heuristics.

MRR heuristic 1: longest time heuristic: The candidate with the longest execution time wins. This is for the promotion of larger tasks.

MRR heuristic 2: highest local rank heuristic: I call the $(w-1)$ lists the local queue. Each local queue tries to promote a candidate, which also has ranks in other local queues.

The summation of all the ranks in the local queues defines the overall fitness, and the task with the largest of these values is selected. As illustrated in Figure 4.2, worker B has three candidates in colors blue, green and red. In the scope of three local queues (B_A, B_C, and B_D), the red task has the overall lowest rank. Thus worker B should select the red task.

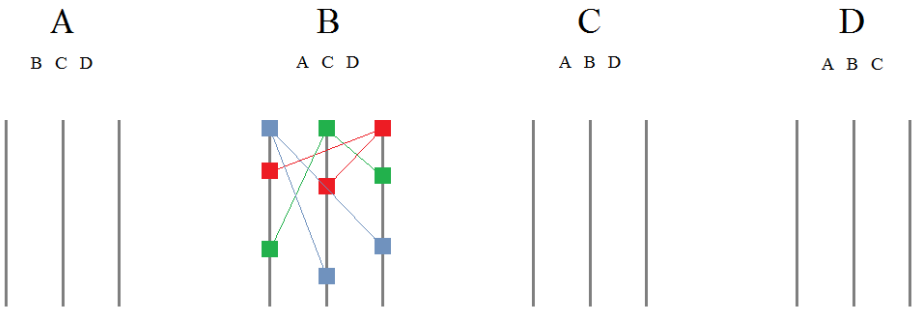

Figure 4.2: MRR heuristic 2: the node in red color hold the highest overall local rank, and is therefore selected.

MRR heuristic 3: lowest remote rank heuristic: If the task is suitable for only one type of worker, it should be unsuitable for the others. The design is demonstrated in Figure 4.3. The system has 4 workers A, B, C and D. Worker B is now ready to choose a task from candidates in color blue, green and red. There candidates hold the highest performance matching score for B versus A, B versus C, and B versus D. To evaluate the fitness of the blue task on worker A, we look into A's local queue. Based on the intuition that the highest rank in A's local queue may determine how soon the blue task is going to start executing on worker A, we choose the highest as the fitness score.

Similarly, the remote fitness score of the green task and red task are calculated from C's local queue and D's local queue. In this example, the red task has the lowest remote fitness score. This implies the red task is the *most unsuitable* for other workers, and is therefore selected.

MRR heuristic 4: improved lowest remote rank heuristic:

The MRR heuristic 3 can be slightly improved from more precisely calculating the remote fitness scores. In Figure 4.3, worker A has three local queues, the blue task places

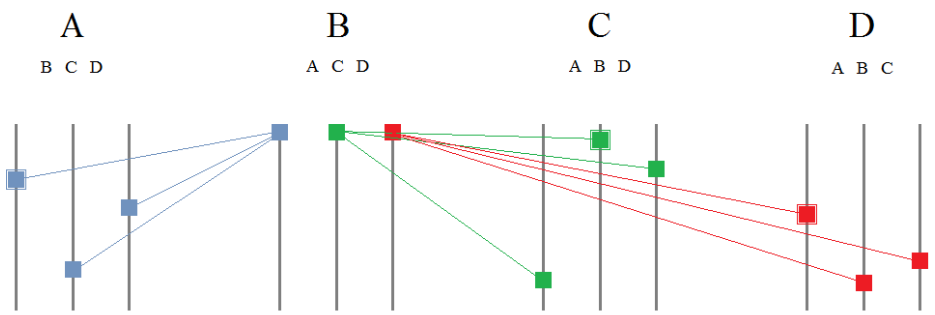

Figure 4.3: MRR heuristic 3 and 4: the node in red color holds the lowest remote rank, and is therefore selected.

the highest on queue A_b. If the task were consumed by worker A, the queueing time is in proportion to the rank on queue A_b. Previously, only the rank is used to derive the score. To improve the precision of the model, I have added the throughput metric. The scoring function is the rank divided by the historic throughput of the queue. A smaller score means a higher suitability. Everything else in this heuristics remains the same with MRR heuristic 3. Again, the most remotely unsuitable task should be selected. The historic throughput can be implemented and measured using counters.

The MRR heuristics are quite moderate in terms of complexity. Let the number of tasks be T. There are w types of heterogeneous worker, and total number of worker is W. T is huge in real world scenarios. W could be large, however, w is typically small. The computational complexity can be split into data preparation time and scheduling run time. For data preparation, creating the task queue is $\mathcal{O}(TlogT)$. Preparing all tasks queues demands the complexity of $\mathcal{O}(w^2TlogT)$. At runtime, when the scheduler issues a task, it compares $w-1$ values on MRR heuristics 1 and 2, and $w \times (w-1)$ values on MRR heuristics 3 and 4. Thus, the execution time complexities are $\mathcal{O}(wT)$ and $\mathcal{O}(w^2T)$, for these two cases respectively. Because T is usually a large number, $logT$ is much greater than 1. The worst case complexity of MRR heuristics is $\mathcal{O}(w^2TlogT)$. The MR heuristics are even faster with the complexity of $\mathcal{O}(wTlogT)$. In comparison, the complexity of two-round searching paradigm is $\mathcal{O}(WT^2)$ (section 4.1).

4.5 AUTOMATIC SMALL TASKS REARRANGING (ASTR)

The MRR heuristic is designed for the sole purpose of optimizing the execution efficiency. The load balancing is largely resolved by ROB. However, ROB is not sufficient enough. If a few big tasks were scheduled in the end of the queue, load balancing may become an issue. Inspired by LPT homogeneous scheduling algorithm, in addition to the ROB and MR/MRR algorithms, I decided to a few small tasks onto the end of the task queues. Since this procedure is automatic, I call it the *Automatic Small Tasks Rearranging* (ASTR) heuristic.

The size of a task is defined by its maximum ET. If the size is below a threshold, I consider it "small." I first sort in decreasing order the task pool by the size. Then the task pool is partitioned into two pieces: large task pool and small task pool. The large pool must be consumed before the scheduler starts to issue tasks from the small task pool. The scheduling algorithm for both pools is ROB + MR/MRR.

Finding the optimal partition is a challenge. If the small task pool does not hold enough tasks, it may not resolve the load balancing issue. On the other hand, too much rearrangement degrades the ranking quality. The relationship of the scheduling quality - the makespan and rearrangement ratio should be a "U" shape curve. Automatic iterative tuning helps to find the minimal point on this curve. The algorithm is fast. First of all, the design space is one dimensional. Second, for each point in the design space, the optimizer calls ROB + MR/MRR algorithm to calculate the score, which has been proved to be very fast (Section 4.4). An iterative tuning framework has been implemented. The auto-tuner slightly increases the rearrangement ratio, and terminates the tuning in one of the following situations: (1) No better solution was found for a given number of tries; (2) The upper bound of rearranging ratio is reached.

4.6 RESULTS

Some of the previously published algorithms were implemented, including OLB, MET, MCT, Min-min, Max-min, Suffrage, BB search, SA Monte Carlo search and GA search. The GA search is seeded with random values. The newly proposed heuristic algorithms, ROB + MR/MRR + ASTR are also implemented. In addition, I implemented a Parallel Tempering Monte Carlo (PT) search [63, 85]. PT has been proven to very useful in find global minimal/maximal value in rough energy landscape. The simulation programs are implemented in C++ without explicit parallelism. They are compiled using GCC 5.4.0 with -O2 flag, and run on a Linux X86-64 host with an Intel Xeon E3-1225 v5 CPU.

The simulation programs reads the following input files. (1) The realistic time table file: It consists of a $w \times T$ metrics, and represents the actual computing time of running T tasks on W different types of heterogeneous workers. (2) The predicted time table file: It is similar to the realistic time table, but the values are slightly mismatched due to the errors on performance predictions. (3) The worker number file: It contains a w width vector, where each vector element represents the number of workers. I use three data sets to prepare the time tables. The first data set is a 204×3 matrix, which is collected from the experiments of running CCDC/Astex dataset on GeauxDock with multicore CPU, GPU and Xeon Phi 2.4.4. The second data set is a 176787×3 matrix. This is collected from running EDUD [86] data set on GeauxDock with the same platforms. The third data set is randomly generated following the method of previous publications [87, 88, 83, 89]. The size of the time table matrix is 512×16, representing the times of computing 512 jobs on 16 workers. The elements of the matrices are randomly generated following the equation $e = r1 \times r2$. Where r1 is a common value across all columns in the same row. $r1 = uniform\ rand(1, r1_{max})$. $r2$ is an independent random number. $r2 = uniform\ rand(1, r2_{max})$. The heterogeneities are varied by changing the values of $r1_{max}$ and $r2_{max}$. The values for the experiments are set to be $r1_{max}$ equals 100 or 3000, $r2_{max}$ equals 10 or 1000. Hence, four different 512×16 matrices are created. For each matrix, there are three variations. The "random copy"

keeps the data untouched. The "sorted copy" arranges every row so that the values are incrementally sorted. The "partial-sorted copy" only incrementally sort even columns of every row, and keeps the odd columns untouched.

In the first phase of the benchmarks, I will ignore the performance prediction issues. The same matrix will be fed to the simulation program as both the estimated execution time and the actual execution time.

I first conduct experiments to show the overall characteristics of the scheduler. The simulation program reads the 204×3 time table and a machine vector "1, 1, 1." This is to simulate scheduling 204 tasks on 3 heterogeneous workers. The scheduling is visualized in Figure 4.4. The x axis denotes time. The 3-bar sets represent the execution status on 3 workers. Each color block is a task. The figure shows 5 algorithms: MCT, Max-min, Min-min, Suffrage, OLB + LPT, and my algorithm (ROB + MR/MRR + ASTR). For those algorithms not included in this figure, MET suffers from severe load balancing issue, and the makespan is about $2 \times$ worse than the others. BB search cannot finish the computation due to the scalability issue. SA search and PT search deliver very close results compared with my algorithms. Among the five algorithms in this figure, OLB + LPT algorithms is the simplest. It does not optimize for performance heterogeneity at all, but the load balancing is perfectly handled. Small tasks are placed at the end, and the workers finish executing at almost the same time. I consider OLB + LPT performance as the baseline. Max-min algorithm tries to address the load balancing issue by selecting the largest task in every step. While this objective has been achieved, however, the performance is even worse than OLB + LPT. The results show Max-min is a false optimization for matching heterogeneous tasks, and should be discarded. MCT scheduling dispatches the tasks in sequence order, although it is better OLB + LPT, there are space to improve. The min-min heuristic and suffrage heuristics show better results over MCT, but either made a good load balance. Comparatively, Min-min is slightly better on the makespan, but the execution efficiencies on each worker are certainly better than that of the suffrage algorithm. This

suggests the suffrage's effort of heterogeneous matching may be no better than Min-min. Finally, my ROB + MRR + ASTR algorithm presents a significant advantage over Min-min heuristic. The makespan is shorter. At the same time, the load balance is perfectly addressed. My algorithm presents 13.82% improvement of OLB + LPT algorithm. and 5.92% improvement over Min-min algorithm.

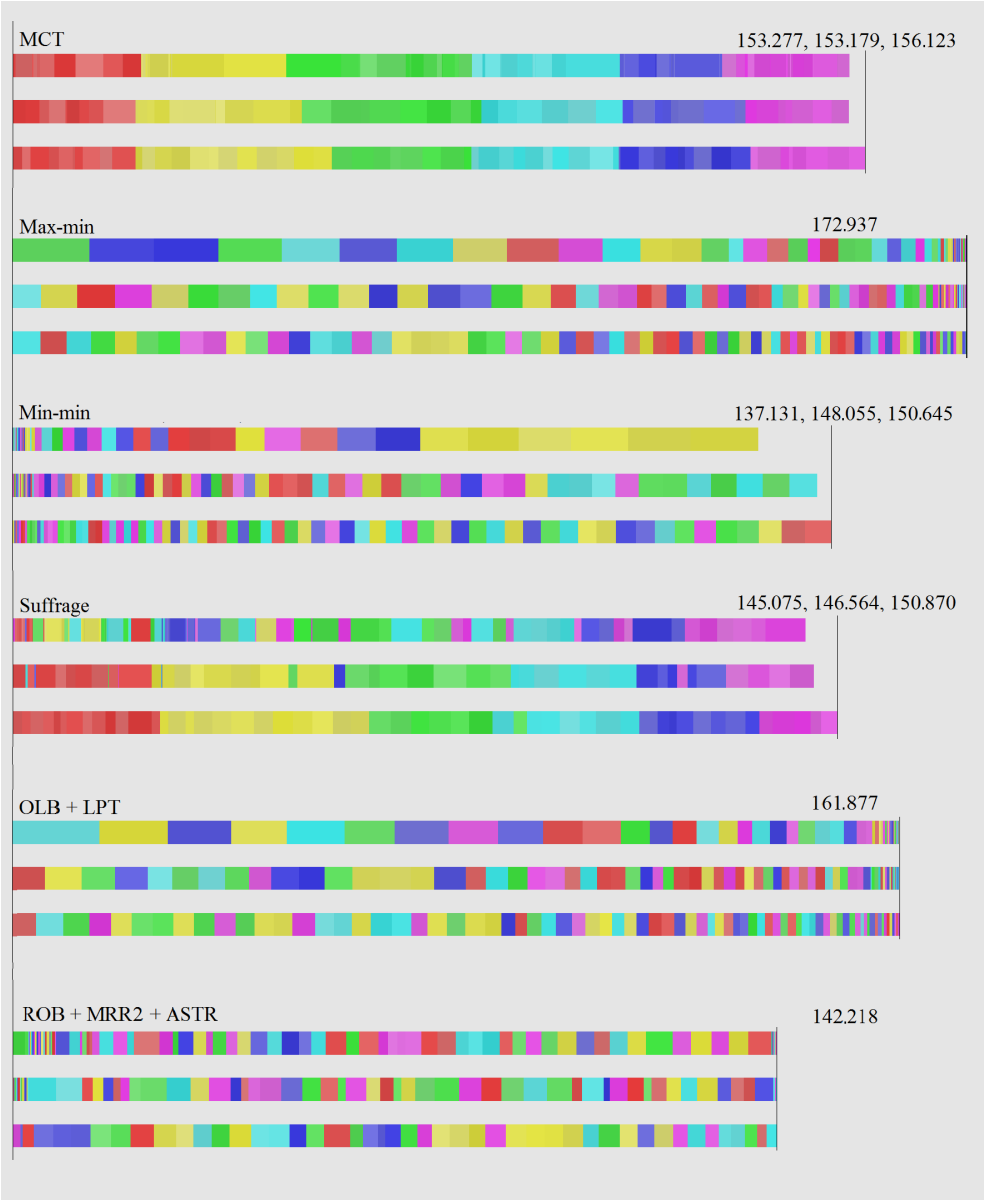

Figure 4.4: The scheduling results of using different algorithms. The experiment is carried on 204 tasks on 3 workers. Each of the three horizontal bar is the time slot of of a worker. Each colored clock is a task.

The quality of scheduling results and characteristics of the algorithms are further stud-

ied. I use the same input data as the last experiment. There are 7 variations of heuristics for the scoring: MRmin, MRmax, MRavg, MRR1, MRR2, MRR3, MRR4. For each variation, the ASTR optimization is turned both off and on. Since the results are close for all algorithms, plot in linear scale cannot make distinguishments. The time for each worker, and the makespan time is shown in Table 4.1. For those with ASTR optimizations, the time of the workers are very close, so these data are eliminated for better clarity. The simulation of SA and PT scheduling takes 10,000,000 Monte Carlo cycles, and about 20 minutes of runtime. The results of SA and PT are almost identical, and are a significant leap over Suffrage and Min-min algorithms. In deep, the performance of Suffrage and Min-min is worse than any of the other algorithms. My algorithm is able to deliver competitive results. Interestingly, the result of the ROB + MRR2 + ASTR scheduling is actually slightly better than SA and PT. My algorithms performs strongly even without the ASTR optimization. This implies the heterogeneous matching is handled really well. ASTR further optimize the result. Comparing the 7 different heuristics of my new algorithms, MRmin, MRavg, MRR2 and MRR4 lead a small gap over the other 4 heuristics. They are likely to be better matching heuristics than others. MRR4 show better result than MRR3, and this implies that the consideration of the work queue throughputs helps improve the scoring function.

Next I use the same 204×3 time table with the "10, 10, 10" a machine vector. This simulates running 204 tasks on 30 workers. The average task per work is 6.8. The purpose of this benchmarking is to observe the load balancing characteristics. Table 4.2 shows that without ASTR, heuristic algorithms cannot match with the Monte Carlo search algorithms when the number of tasks per workers is low. However, after the ASTR is added, the results are really close. Precisely, the combination of ROB + MRavg + ASTR heuristics is 2.44% close to the best result. Comparing two Monte Carlo search algorithms, SA delivers better results than PT for this test case. Old heuristic algorithms suffrage and Min-min are indeed worse than the baseline OLB + LPT due to severe load balancing issues. Interestingly, two

Algorithms	CPU queeu	Xeon Phi equue	GPU queue	makespan
OLB + LPT	161.814	161.782	161.877	161.877
Suffrage	145.075	146.564	150.870	150.870
Min-min	137.131	148.055	150.645	150.645
SA	142.277	142.249	142.240	142.277
PT	142.273	142.290	142.291	142.291
ROB + MRmin	140.963	142.826	142.939	142.939
ROB + MRmin + ASTR				142.939
ROB + MRmax	144.361	144.375	144.376	144.693
ROB + MRmax + ASTR				144.376
ROB + MRavg	141.353	142.120	142.549	142.549
ROB + MRavg + ASTR				142.549
ROB + MRR1	143.965	144.693	143.682	144.693
ROB + MRR1 + ASTR				144.376
ROB + MRR2	143.451	142.951	1429.15	143.451
ROB + MRR2 + ASTR				142.218
ROB + MRR3	144.568	144.299	143.682	144.568
ROB + MRR3 + ASTR				144.329
ROB + MRR4	144.053	143.536	143.178	144.053
ROB + MRR4 + ASTR				143.196

Table 4.1: The makespans of scheduling 204 tasks on 3 workers.

of my algorithms, ROB + MRmin + ASTR and ROB + MRavg + ASTR did not perform as good as they were in the previous benchmark. This should contribute to the random effect by having smaller number of tasks per worker.

The third experiment uses a large task pool. The size of the time table is 176787×3, and the machine vector is "10, 10, 10." This experiment simulates running $176,787$ tasks on 30 workers. For this size of input data, Monte Carlo search algorithm can hard converge. Heuristic algorithms are the only practical solutions. The baseline OLB + LPT algorithm is still very fast, because its computational complexity is as low as $\mathcal{O}(TlogT)$. The suffrage and Mini-min algorithms spend more than 4 hours computation time. My MR heuristics are almost as fast as the OLB + LPT algorithm, and could deliver the result in less than 0.1 seconds. The MRR heuristics are slower than MR, because they need to prepare $w \times (w-1)$ work queues instead of w work queues. However the run time of the MRR schedulers is still under 0.3 seconds. Quality wise, my algorithms provide 13.6% lower makespan than

Algorithms	makespan
OLB + LPT	17.071
Suffrage	18.021
Min-min	18.242
SA	15.027
PT	15.261
ROB + MRmin	19.253
ROB + MRmin + ASTR	16.489
ROB + MRmax	17.383
ROB + MRmax + ASTR	15.513
ROB + MRavg	16.890
ROB + MRavg + ASTR	15.395
ROB + MRR1	17.338
ROB + MRR1 + ASTR	15.513
ROB + MRR2	18.111
ROB + MRR2 + ASTR	15.541
ROB + MRR3	17.265
ROB + MRR3 + ASTR	15.513
ROB + MRR4	17.262
ROB + MRR4 + ASTR	15.579

Table 4.2: The makespans of scheduling 204 tasks on 30 workers.

OLB + LPT, and is at least 3.06% better than suffrage and Min-min algorithms. Load balancing is unimportant for this test case. The scheduler understands that the average number of tasks per worker is a few thousand, so the ASTR feature is automatically turned off.

Algorithm	makespan	execution time (seconds)
OLB + LPT	11657	0.055
Suffrage	10708	21660
Min-min	10571	14552
ROB + Smin	10258	0.087
ROB + Smax	10342	0.087
ROB + Savg	10257	0.087
ROB + MRR1	10342	0.253
ROB + MRR2	10307	0.262
ROB + MRR3	10363	0.257
ROB + MRR4	10264	0.261

Table 4.3: The makespans of scheduling 176787 tasks on 30 workers.

Finally, I test the scheduling algorithms on synthetic data set of 12 512×16 time

tables. The machine vector is "1, 1, 1, ... 1." Table 4.4 shows the normalized makespans. The relative performance of the algorithms vary between data sets. However, it is clear that variations of ROB + MR/MRR + ASTR algorithms outperform any other algorithms. MRR heuristic is the best of all. It delivers the top performance in 6 out of 12 test cases, In other cases, the result of MRR2 is very close to the best result. The performance of Min-min algorithm is competitive too. Sufferage, as well as the other old heuristic algorithms are significantly slower.

$r1_{max}$	100	100	100	100	100	100
$r2_{max}$	1000	1000	1000	10	10	10
sorting columns	no	partial	yes	no	partial	yes
OLB + LPF	7.676	4.420	1.696	3.220	2.480	1.309
MET	1.304	4.125	5.498	1.471	5.843	7.669
MCT	1.199	1.247	1.342	1.191	1.252	1.169
Max-min	2.251	1.960	1.518	1.939	1.800	1.301
Min-min	1.064	1	1.051	1.059	1.060	1.058
Sufferage	1.622	1.511	1.324	1.424	1.402	1.164
ROB + MRmin + ASTR	1.010	1.003	1	1.054	1.029	1.027
ROB + MRmax + ASTR	1.053	1.047	1.061	1.054	1.025	1.024
ROB + MRavg + ASTR	1	1.029	1.016	1.013	1.034	1.020
ROB + MRR1 + ASTR	1.053	1.047	1.061	1.054	1.025	1.024
ROB + MRR2 + ASTR	1.008	1.043	1.003	1	1	1
ROB + MRR3 + ASTR	1.008	1.102	1	1.031	1.032	1.027
ROB + MRR4 + ASTR	1.036	1.095	1.060	1.063	1.045	1.055
$r1_{max}$	3000	3000	3000	3000	3000	3000
$r2_{max}$	1000	1000	1000	10	10	10
sorting columns	no	partial	full	no	partial	full
OLB + LPF	6.752	4.035	1.290	3.359	2.344	1.291
MET	1.412	4.992	7.608	1.239	5.840	7.608
MCT	1.250	1.310	1.171	1.223	1.215	1.171
Max-min	1.973	1.929	1.319	1.840	1.714	1.319
Min-min	1.022	1.047	1.040	1.057	1.032	1.040
Sufferage	1.509	1.436	1.130	1.391	1.333	1.130
ROB + MRmin + ASTR	1.057	1.076	1.015	1.012	1.029	1.015
ROB + MRmax + ASTR	1.043	1	1.021	1.008	1.016	1.021
ROB + MRavg + ASTR	1.024	1.226	1.011	1.013	1	1.011
ROB + MRR1 + ASTR	1.043	1	1.021	1.008	1.016	1.021
ROB + MRR2 + ASTR	1.014	1.150	1	1	1.002	1
ROB + MRR3 + ASTR	1.011	1.105	1.015	1.050	1.042	1.015
ROB + MRR4 + ASTR	1	1.047	1.055	1.061	1.053	1.055

Table 4.4: The normalized makespans of scheduling 512 tasks on 16 workers. 12 different time matrices are tested.

So far, the experiments have demonstrated the performance of various scheduling algorithms when they are feed with perfectly predicted data. In the real word, the execution time of a program is not available at scheduling time. The error of the performance estimation may significantly affect the quality of the schedules. In the last phase of this section, I'm going to exam this fact. I feed the scheduling simulator with two time tables. One represents the actual execution time of the tasks. The other represent the predicted run

time. Since the scheduling problem is extract from a real world problem, the performance predictions (Chapter 3) is also conducted from the real world. Other than that, I also add synthetic performance prediction data. This will give finer control on different aspects of the errors, and allows to examine how the schedulers respond to these errors. To generate the synthetic performance prediction data, I take the original performance data, and add uniformly distributed noise. The bias error is only applied to one of the three types of heterogeneous workers. If the variation and bias in the noise are zeros, the generated data and the original data are identical. This is called perfect prediction (See Equation 4.4).

$$Pred = Orig * (Noise_{uniform\ distribution}(1 - variation,\ 1 + variation) + bias) \qquad (4.4)$$

Two data sets are used. The first is 204×3 time tables plus the "1, 1, 1" machine vector, for the case of running 204 task on 3 workers. The second test case utilize 176787 time tables and the "10, 10, 10" machine vector, which stands for running 176787 tasks on 30 workers. The results are listed in Table 4.5 and Table 4.6. The reference performance numbers are those generated by OLB + LPT, which is always the slowest.

The results of the of suffrage and Min-min algorithms are shown in Table 4.5. They offer equally 8.64% speedup over OLB + LPT algorithm, on the realistic case (the line of "practical predictions"). However, are 6.36% slower compared with the best of my schedule algorithms, the OLB + MRR4 + ASTR. Running the synthetic performance prediction data, if 0.2 variations are added, the performance of suffrage algorithm drop 1.18%, and the performance of Min-min algorithm drops 2.78%. If I further add the bias of "-0.1," 2.21% and 5.96% slow down are observed respectively. My new heuristic algorithm suffers from this problem too. However, the degradations are clearly smaller than Min-min algorithm, but not as good as the suffrage algorithm. However, on the practical performance prediction data, my algorithms, especially the MRR2 and MRR4 suffer the

lowest performance impact. A clear trend can be observed by comparing the last two lines of Table 4.5. When the bias as added, the performance impacts of my algorithms are constantly smaller than suffrage and Min-min algorithms. This is because my algorithm utilized the run time dynamic scheduling scheme. If one worker is faster than expected,it will automatically fetch more tasks. The loads balancing are not impacted. On the other size, the static scheduling such as suffrage and Min-min suffers severely from load balance problem. A few workers will finish their tasks sooner and keep idle for the rest of time.

input data	OLB+LPT	suffrage	Min-min	MRmin	MRavg	MRR2	MRR4
perfect predictions	161.877	150.870	150.645	142.939	142.549	142.218	143.196
practical predictions	165.266	152.141	152.141	144.191	144.333	143.240	143.031
variation 0.2, bias 0	166.513	152.662	154.835	146.199	147.817	146.876	146.224
variation 0.2, bias -0.1	165.107	154.211	159.619	147.597	147.940	147.309	147.787

Table 4.5: The makespans of scheduling 204 tasks on 3 workers. The estimated performances of the tasks are not always equal to the actual performance. As a result, the qualities of the schedulings degrade.

Table 4.6 shows the makespans of scheduling 176787 tasks on 30 workers. The scheduling time for such a problem is longer than 4 hours for suffrage and Min-min heuristics. I skipped those impractical algorithms. The results of my 4 algorithms match closely. However, the MRmin heuristic shows advantage on almost all test cases. Meanwhile, MRmin heuristic is very competitive in table 4.5. For any input data, MRmin heuristic algorithm could archive no less than 12% speedup. Again, in table 4.6, the additional bias errors does not impact the performance. It actually speedup a little bit. This is because the "-0.1" bias means the worker is faster than the expectation. In fact, the constant bias does not affect the task ranking. The matching mechanism in my algorithm still runs perfectly.

input data	OLB+LPT	MRmin	MRavg	MRR2	MRR4
oracle predictions	11657	10259	10258	10307	10265
practical predictions	11593	10301	10310	10321	10330
variation 0.2, bias 0	11670	10493	10527	10552	10575
variation 0.2, bias -0.1	11663	10492	10523	10545	10571

Table 4.6: The makespans of scheduling 176787 tasks on 30 workers. The tasks estimated performances are not always equal to the actual performance. Thereby the qualities of the schedules degrade.

Chapter 5
Optimizing for Power and Energy

5.1 BACKGROUND

Power and energy have become the major bottlenecks for developing high performance computer systems (see Chapter 1.1). As a result, the effectiveness of a computer system should not be solely quantified by the performance metrics, such as the makespan time of executing a set of tasks. The power consumption and energy consumption should also be considered as first-order metrics. In this chapter of my thesis, I'll revisit the scheduling problem for heterogeneous tasks and heterogeneous processors, with a new perspective from power and energy efficiencies.

To begin with the study, the power characteristics of the processors must be modeled. The power statistics of modern processors depends on the workload. When no workload is present, the static power dominates the total power consumption. The number is typically low. An increase in workload adds dynamic power, and causes the processor to burn more power until the upper bound (i.e., maximum power) is reached. Thermal Design Power (TDP) measures the maximum heat dissipation rate for typical workload. TDP is available from the vendor specification, and could be used to estimate the maximum power if the latter is hard to obtain. In practice, measuring the power has two possible methods. Analog power meters are always reliable; however, it suffers from high expense and low flexibility. A more practical method is using software regression models.

worker	idle power (Watt)	loaded power (Watt)
Xeon E5 2680 v2 CPU × 2	40	230
Xeon Phi 7120P × 1	103	185
Tesla K20Xm GPU × 1	19	110

Table 5.1: Typical power consumptions of heterogeneous processors running GeauxDock.

Running Average Power Limit (RAPL) [90] provides a set of counters for energy and

power consumption information on Intel CPUs. RAPL uses a software power model. It estimates energy usage by using hardware performance counters and I/O models. Intel claims the update rate of RAPL Machine Specific Registers (MSRs) is once every millisecond. RAPL driver has been implemented in the Linux kernel, and could be accessed using the *perf* [91] tool. Accessing RAPL MSRs needs super-user privilege. As a result, the power metrics of Xeon E5 2680 v2 CPU installed on a public supercomputer is not directly available. The closest matching CPUs are a pair of Xeon E5-2620 v3s, which are installed on my workstation. Using the *perf* tool, I obtained their power metrics. The idle power of a CPU is 20W, and its peak power is 90W when running GeauxDock, which closely matches its 85W TDP. Therefore, I use 20W as the idle power for Xeon E5 2680 v2, and use the TDP number (115W) to estimate the typical loaded power. See Table 5.1.

The power statistics of the Xeon Phi co-processor can be measured using 3 different software tools. RAPL interface will be supported on the second generation of Xeon Phi "KnightLanding." It will provide the identical interface as its counterpart on Intel CPUs. For older architecture, MPSS libraries and MPSS utilities [92] are the best measurement tools, and they are shipped with the driver. Alternatively, PAPI provides the micpower API since version 5.3.2 [93]. My experiment is carried with running GeauxDock on Xeon Phi, while running the MPSS utility, micpower, to collect the power samples on an interval of 100 milliseconds. The time sequence is plotted in Figure 5.1. The idle power and typical loaded power us set to be 103W and 185W. See Table 5.1.

For NVIDIA GPUs, the power metrics is accessible via NVIDIA Management Library (NVML) library [94]. NVIDIA claims the error rate of the power readings is no larger than 5%, and the read period is in the microsecond scale. Utilizing the NVML library, I implemented a utility that samples the GPU power statistics in 10 millisecond intervals. The utility is coupled to launch with the GPU version of GeauxDock. Figure 5.2 shows the time sequence data. The data show that the typical loaded power of the GPU is 110W, and the idle power is 19W (Table 5.1).

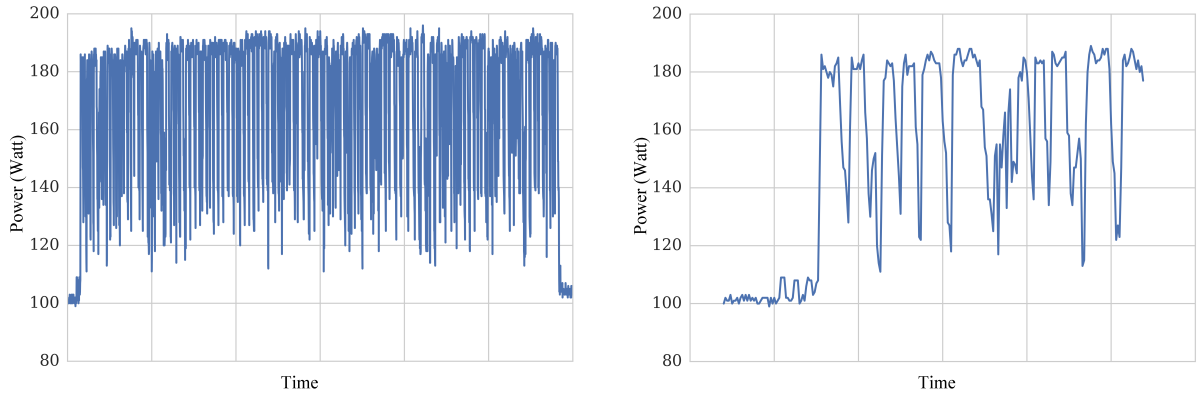

Figure 5.1: The power statistics of Xeon Phi 7120P running GeauxDock. The right figure zooms into the beginning of the procedure.

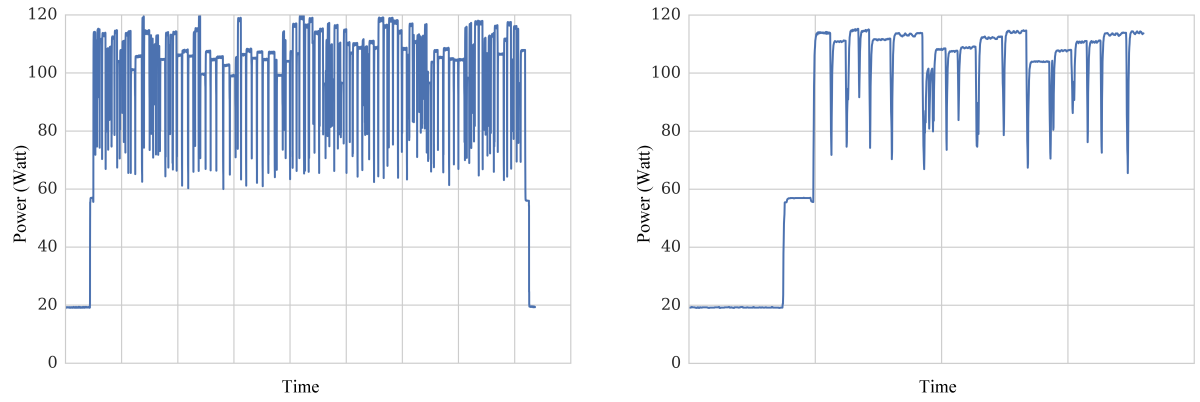

Figure 5.2: The power statistics of Tesla K20m GPU running GeauxDock. The right figure zooms into the beginning of the procedure.

5.2 POWER-CONSTRAINED TASK SCHEDULING

Optimized task scheduling problem without power constrains has been discussed in Chapter 4. The fundamental idea is to make all workers busy all the time. Meanwhile, when a worker fetches a new task, it should favor those with higher suitability scores. In addition to this design, I have added the power capping feature, which makes sure the total power consumption is under a certain threshold. Power capping ratio is used to quantify the level of power capping. It means the ratio between the current threshold and the maximum power of the system. If the power capping rate equals to 1, then the system is not power capped at this configuration.

The power and energy status of the system is modeled as follows. The execution time of mapping task T on worker W is t_{tw}. The workers have two power states: loaded power Pl_{tw} and idle power Pi_{tw}. The dynamic power is Pd_{tw} which equals to $Pl_{tw} - Pi_{tw}$. The total energy consumption of the computer system is $E = \sum_w (Pi_{tw} * makespan) + \sum_{w,t} (Pd_{tw} * t_{tw})$. The idle powers are available from the second column of Table 5.1. To simply the experiments, I use the typical loaded powers for Pl_{tw}, which are shown in the third column of Table 5.1.

The power-capping task scheduling algorithm inherits the scheduling principle from ROB + MR/MRR + ASTR algorithm (Chapter 4.2). The tasks are arranged in many priority queues. In each queue, the tasks are arranged for a specific type of heterogeneous worker by sorting the performance suitability scores in decreasing order. Similar to the implementation of ROB + MR/MRR + ASTR algorithm, the power capping task scheduling system also utilizes the server-client model. The workers actively send requests to a centralized scheduling server. Once the feedback message is received, the worker reads the information and start computing the task. Otherwise, if no message is received, the worker keeps idle and consumes the minimal amount of power. The server maintains the system status, and issues tasks to the workers by replying their requests. In the non-power capping scheduling, the requests are responded immediately. However, in power-capping scheduling, the server intentionally holds some requests. In such a way, the scheduler could turn down those less energy efficient workers. Thus the system achieves higher energy efficiency.

Detailed behaviors of the scheduler server are described in Algorithm 1. The key component of the algorithm is to calculate the set of workers that are valid fora scheduling for a certain power constraint. This is described in the procedure "find_valid_workers." Afterwards, using the workers as inputs, the scheduler applies ROB + MR/MRR + ASTR algorithm (Chapter 4) to find their associated tasks. Among these tasks, only one task is to be selected and issued. The selection procedure is guided by the energy efficiency scoring

Algorithm 1 Power-constrained heterogeneous scheduling algorithm

```
1: procedure FIND_VALID_WORKERS()                          ▷ find valid worker candidates
2:     P ← the current power consumption of the system
3:     W ← the set of idle workers
4:     W_1 ← deduplicate W by worker types
5:     W_2 ← filter W_1 by condition (P + w_1 < P_capping)          ▷ power capping
6:     return W_2
7: procedure SCHEDULING(T, W)                  ▷ Schedule task set T on worker set W
8:     while T ≠ ∅ do
9:         W_c ← find_valid_workers()
10:        if W_c ≠ ∅ then
11:            T_c ← find task candidates for W_c, using ROB + MR/MRR + ASTR algorithm
12:            t ← select a task from T_c, by the largest energy efficiency score
13:            issue task t to the corresponding worker w
```

function SE. Similar to the performance efficiency scoring function S 4.4, SE has many variations. Previous experiments has proved $MRmin$, $MRavg$, $MRR2$ and $MRR4$ are effective heuristics for S. Since the heuristics for optimizing energy metrics are no different from that of optimizing performance, these four heuristics should also work well for SE. Ultimately, I choose SE_{min} (Equation 5.1) as the energy scoring function because of the simplicity. A higher score of SE_{min} implies that scheduling this task has better energy efficiency, and therefore, should be selected.

$$SE_{min\ i} = \frac{Energy_{min}}{Energy_i} \tag{5.1}$$

The computational complexity of power capping heterogeneous scheduler is also low. Recall that in Chapter 4.4, I defined a few quantities. The number of tasks is T, the number of heterogeneous worker types is w, and total number of workers is W. The complexity of ROB + MRmin + ASTR algorithm is $\mathcal{O}(wTlogT)$. The power capping scheduling algorithm extends the ROB + MRmin + ASTR algorithm by adding extra computations for selecting the most energy efficient workers at run time. The amount of computations per task is in proportion to the number of workers. So, there is an additional $\mathcal{O}(WT)$

complexity. The overall computational complexity is $\mathcal{O}(wTlogT + WT)$.

5.3 RESULTS

The energy capping scheduling algorithm is implemented in my scheduling simulator (Chapter 4.6). I use SE_{min} (Equation 5.1) as the energy efficiency scoring function, and S_{min} (Equation 5.1) as the performance scoring function. Experiments are carried out on the same computer as the earlier scheduling experiments, a Linux X86-64 host with an Intel Xeon E3-1225 v5 CPU. The data for scheduling is the heterogeneous implementations of Geuxdock running EDUD [86] data set, which is a 176787×3 matrix. The performance prediction of the tasks are generated using regression models (Chapter 3).

In the first experiment, I simulated scheduling 176787 tasks on 30 workers consist of 10 dual-CPUs, 10 Xeon Phis, and 10 GPUs. The power capping scheduler spends 0.91 seconds for the computation. Compared with the 0.087 seconds computation time of the non-power capping scheduling algorithm (Table 4.3), the power capping scheduling algorithm is slower but still very practical.

A range of power capping ratios are tested. Figure 5.3 plots the relative speedups of the makespans and the energy consumptions. The figure shows that the makespan is monotonically increasing, which means more aggressive power capping always slows down the computation. The energy consumptions curve follows a U shape. It drops in the beginning, then increases. In the most aggressive power capping case (where the power capping ratio is 0.45), the energy consumption is even higher than the no capping case. Overall, the 0.65 power capping ratio is a sweet spot, at which the heterogeneous computing system saves 35.0% power as well as 31.5% energy, on the cost of 19.56% performance slowdown. Interestingly, 0.65 power capping ratio is the turning point of energy efficiency metrics. To understand this trend, I checked the utilization of each worker. It turns out that the CPU workers are the least energy efficient ones, and the scheduler will always turn off CPUs whenever possible. As far as the number of idle CPUs increase, the overall

energy efficiency of the system also increases. However, when all the CPUs are turned off, the scheduler starts to turn off Xeon Phis or GPUs. The idle power of the workers will become increasingly dominant. As a result, the power efficiency drops. According to Table 5.1, when the scheduler turn down all the CPUs, the power capping ratio is $\frac{(40+185+110)\times 10}{(230+185+110)\times 10} = 0.638$. This is very close to the number 0.65 in my test cases. In conclusion, this experiment verifies that power capping heterogeneous scheduling algorithm is able to select the best energy efficient workers, and boost the power efficiency and energy efficiency.

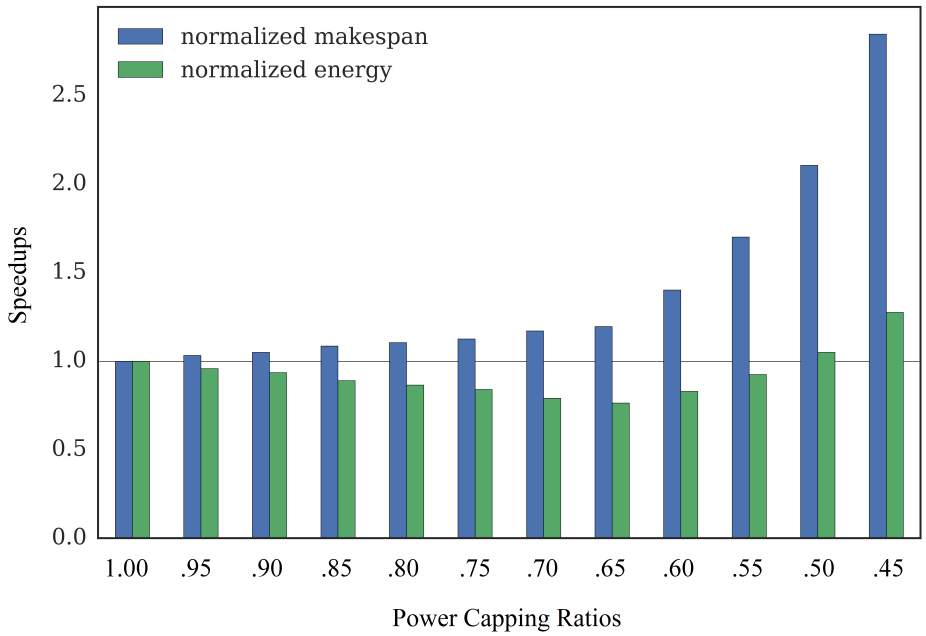

Figure 5.3: The impact of the performance and energy consumption by applying different power capping ratios.

Next, I compare the power efficiencies of different computing a system. The typical loaded power of a Xeon Phi is 185W, and the number for a GPU is 110W.(Table 5.1). An 8140W power budget could supply 44 Xeon Phis, or 74 GPUs, or 22 Xeon Phis plus 37 GPUs. The three hypothetical computing systems are configured accordingly: (1) a pure 44 Xeon Phis system; (2) a pure 77 GPUs system, and (3) a heterogeneous 22 Xeon Phis + 37 GPUs system. All of the systems are loaded with the 176787 heterogeneous tasks. I used

ROB + MRmin + ASTR scheduling algorithm without power capping. The metrics of this experiment can be either makespan time or total energy consumption. However, since all systems consume the same amount of power, these two metrics are in proportion with each other. Ultimately, I select the energy consumption metrics, and normalized the values with that of the best system. The results are shown in Figure 5.4. It is very clear that pure GPU system delivers the best power efficiency. The pure Xeon Phi system is 92.15% worse than the pure GPU system. This implies, on average, the GPUs deliver a stronger performance across all the tasks. If GPU and Xeon Phi are combined without utilizing a good scheduler, I'm expecting a linearly combined makespan time, which is $\frac{1.0+1.9215}{2} = 1.4607$. In contrast, my heterogeneous scheduler is able to achieve a 1.1644 normalized makespan, which is 25.45% better than the result of linear combination. Indeed, this speedup is achieved by taking care of the performance variations of different tasks, and mapping the tasks to the most suitable workers. Although the performance of Xeon Phi + GPU system is lower than pure GPU system. It didn't suggest that the philosophy of heterogeneous system is wrong. In contrast, this experiment suggests a performance potential of heterogeneous systems. When the performance heterogeneities are high, and the average performance between workers does not differ too much, a systematic combination of many heterogeneous workers could improve the overall power efficiency.

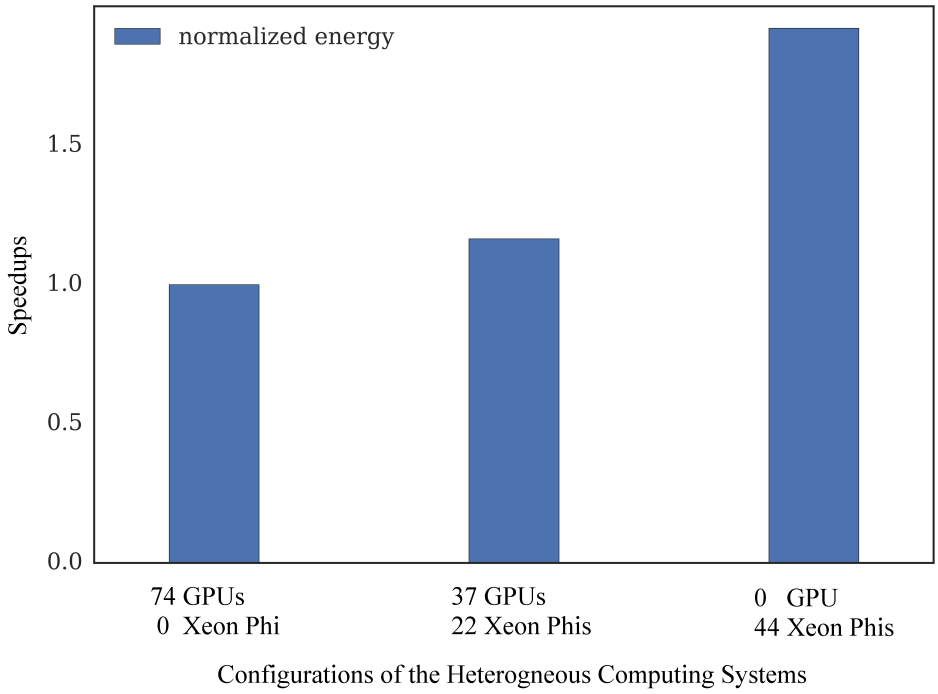

Figure 5.4: The comparison of three computer systems running the same tasks set. All of the computer systems consume the same amount of power of 8140W.

Chapter 6
Conclusion

Accelerated parallel computing using devices such as GPUs and Xeon Phis, along with traditional CPUs, have great promise in extending the cutting edge of high-performance computer systems. Such systems are referred to as heterogeneous systems. A significant performance improvement can be achieved when suitable workloads are handled by the accelerator. This thesis addressed four topics in the context of heterogeneous computing.

The first topic discussed is the acceleration and tuning techniques in the context of *GeauxDock,* a molecular docking package featuring a novel scoring function and Monte Carlo-based conformational space sampling. GeauxDock is designed for large-scale virtual screening applications using heterogeneous computer architectures. Because of its modular code framework, GeauxDock supports modern multi-core CPU, as well as Xeon Phi and GPU accelerators. I devoted considerable effort to minimize the data communication leading to at least 95% of the time spent on executing MMC kernels. Various tuning techniques have been applied to significantly accelerate the docking kernel based on the performance characteristics obtained by a meticulous code profiling using diverse input data. For instance, a systematic optimization of the serial CPU code brought about not only a 6.5 $\times$ speedup on a single computing core, but also a perfect scaling with the number of cores on modern shared-memory platforms equipped with multiple sockets of multi-core CPUs. Docking benchmarks conducted on many-core accelerators show that using Xeon Phi 7120P yields 1.9 $\times$ performance improvement over a dual-socket Xeon E5 CPU, whereas the fastest GPU, GeForce GTX 980, achieves a 3.5 $\times$ speedup over a dual CPU. It is important to note that in addition to hardware capabilities, a thorough code tuning for accelerator devices plays an important role in increasing the computational performance. In addition to the evaluation of a purely computational performance, I examined the energy consumptions and hardware costs. In conclusion, heterogeneous com-

puting platforms, especially those equipped with the latest GPU cards, offer significant advantages over traditional CPU-based systems. Using parallel codes optimized for modern heterogeneous HPC architectures can significantly accelerate structure-based virtual screening applications. GeauxDock is open-source and publicly available from our website at http://brylinski.cct.lsu.edu/geauxdock

The second topic analyzes the performance characteristics of heterogeneous processors. Two models are developed for the analysis of correlations between the performance and the input data. The coarse grained model addresses how replica scaling impacts core utilization. Both the modulo line model as well as the isotonic regression model are utilized for the coarse grained model. The fine-grained model deals with the heterogeneous performance speed-ups on the three major components of the compute kernel. The result is that I have figured out a set of parameters to regulate the feature set of each component. Then, linear regression is used to predict the weight of each component. Finally, these two models are combined to provide accurate performance predictions for heterogeneous tasks running on heterogeneous processors.

The third topic of this thesis studies the batch scheduling of running independent heterogeneous workloads on heterogeneous processors. The objective is to minimize the wall time of the computation, a.k.a. *makespan*. All heterogeneous workers must be fully utilized when power constraints do not exist. The system should achieve a good source matching for execution efficiency, as well as a good load balancing. In my research, I proposed a heuristic algorithm composed of four major components: Ranked Opportunistic Balancing (ROB), Multi-subject Ranking (MR), Multi-subject Relative Ranking (MRR), and Automatic Small Tasks Rearranging (ASTR). My algorithm consistently outperforms previously proposed algorithms achieving better schedules and with lower computational complexities. Additionally, it delivers more consistent results on imperfect performance predictions. Not only does my algorithm deliver the best results, but also it is the only practical method to resolve realistic large problems.

Finally, I study the power and energy efficiencies of heterogeneous systems. I collected the power characteristics of Xeon Phi and GPU. A power and energy model was then derived for the heterogeneous task scheduling problem. Also, I designed a new algorithm to resolve the scheduling problem under a certain power budget. It is proven to significantly improve the power efficiencies and energy efficiencies for heterogeneous computing systems.

References

[1] E. Strohmaier, J. Dongarra, H. Simon, M. Meuer, and H. Meuer, *TOP500 list, November 2015*, 2015.

[2] H. W. Meuer, E. Strohmaier, J. Dongarra, and H. D. Simon, *The TOP500: History, Trends, and Future Directions in High Performance Computing*. Chapman & Hall/CRC, 1st ed., 2014.

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

[4] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *Proceeding of the 38th annual international symposium on Computer architecture - ISCA '11*, p. 365, 2011.

[5] W. C. Feng, X. Feng, and R. Ge, "Green supercomputing comes of age," *IT Professional*, vol. 10, pp. 17–23, 2008.

[6] P. a. Merolla, J. V. Arthur, R. Alvarez-Icaza, a. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, a. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, pp. 668–673, aug 2014.

[7] G. Lacey, G. W. Taylor, and S. Areibi, "Deep learning on fpgas: Past, present, and future," *CoRR*, vol. abs/1602.04283, 2016.

[8] B. A. Draper, J. R. Beveridge, A. P. Bohm, C. Ross, and M. Chawathe, "Accelerated image processing on fpgas," *Trans. Img. Proc.*, vol. 12, pp. 1543–1551, Dec. 2003.

[9] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*. New York, NY, USA: Cambridge University Press, 10th ed., 2011.

[10] J. Planas, R. M. Badia, E. Ayguade, and J. Labarta, "Hierarchical Task-Based Programming With StarSs," *International Journal of High Performance Computing Applications*, vol. 23, pp. 284–299, jun 2009.

[11] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, pp. 187–198, feb 2011.

[12] G. Teodoro, R. Sachetto, O. Sertel, M. N. Gurcan, W. M. Jr, U. Catalyurek, and R. Ferreira, "Coordinating the Use of GPU and CPU for Improving Performance of Compute Intensive Applications,"

[13] S. Henry, D. Barthou, A. Denis, R. Namyst, and M.-c. Counilh, "SOCL : An OpenCL Implementation with Automatic Multi-Device Adaptation Support," no. August, 2013.

[14] K. Chandramohan and M. F. P. O'Boyle, "A compiler framework for automatically mapping data parallel programs to heterogeneous MPSoCs," *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems - CASES '14*, pp. 1–10, 2014.

[15] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather, "Active learning accelerated automatic heuristic construction for parallel program mapping," *Proceedings of the 23rd international conference on Parallel architectures and compilation - PACT '14*, pp. 481–482, 2014.

[16] S. AbuAsal, R. Tohid, and J. Ramanujam, "Lost in heterogeneity: Architectural selection based on code features," in *Proceedings of the 2Nd International Workshop on Hardware-Software Co-Design for High Performance Computing*, Co-HPC '15, (New York, NY, USA), pp. 6:1–6:6, ACM, 2015.

[17] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 5th ed., 2011.

[18] D. Shreiner, G. Sellers, J. M. Kessenich, and B. M. Licea-Kane, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley Professional, 8th ed., 2013.

[19] B. Bargen and P. Donnelly, *Inside DirectX: In-depth Techniques for Developing High-performance Multimedia Applications*. Redmond, WA, USA: Microsoft Press, 1998.

[20] Nvidia, "CUDA C Programming Guide," 2016.

[21] B. Nichols, D. Buttlar, and J. Farrell, *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.

[22] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, p. 298, 2008.

[23] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, p. 6, ACM, 2014.

[24] B. Schling, *The Boost C++ Libraries*. XML Press, 2011.

[25] O. A. R. Board, "OpenMP Application Programming Interface," no. March, 2015.

[26] M. P. I. Forum, "MPI : A Message-Passing Interface Standard," 2015.

[27] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: A system for programming graphics hardware in a c-like language," *ACM Trans. Graph.*, vol. 22, pp. 896–907, July 2003.

[28] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, "Software pipelined execution of stream programs on gpus," in *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, (Washington, DC, USA), pp. 200–209, IEEE Computer Society, 2009.

[29] M. McCool and S. D. Toit, *Metaprogramming GPUs with Sh*. AK Peters Ltd, 2004.

[30] M. D. McCool, K. Wadleigh, B. Henderson, and H.-Y. Lin, "Performance evaluation of gpus using the rapidmind development platform," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, (New York, NY, USA), ACM, 2006.

[31] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: Stream computing on graphics hardware," in *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, (New York, NY, USA), pp. 777–786, ACM, 2004.

[32] M. Papakipos, "The peakstream platform: High-productivity software development for multi-core processors," in *Proceedings of Windows Hardware Engineering Conference (WinHEC), Industry Papers*, 2007.

[33] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *IEEE Des. Test*, vol. 12, pp. 66–73, May 2010.

[34] S.-z. Ueng, M. Lathara, S. S. Baghsorkhi, and W.-m. W. Hwu, "CUDA-lite : Reducing GPU Programming Complexity,"

[35] T. D. Han, T. Edward, and S. R. Sr., "hiCUDA : A High-level Directive-based Language for GPU Programming," *Computer Engineering*, vol. 383, pp. 1–10, 2009.

[36] S. Lee, "OpenMPC : Extended OpenMP for Efficient Programming and Tuning on GPUs Rudolf Eigenmann," vol. 7, no. 1, 2012.

[37] R. Dolbeau, S. Bihan, and F. Bodin, "Hmpp: A hybrid multi-core parallel programming environment,"

[38] M. Wolfe, "Implementing the pgi accelerator model," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, (New York, NY, USA), pp. 43–50, ACM, 2010.

[39] OpenACC Working Group, "The OpenACC Application Programming Interface version 2.0," vol. 2.0a, 2013.

[40] G.-T. Bercea, C. Bertolli, S. F. Antao, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, and K. O'Brien, "Performance analysis of openmp on a gpu using a coral proxy application," in *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, PMBS '15, (New York, NY, USA), pp. 2:1–2:11, ACM, 2015.

[41] M. Amini, S. Guelton, J. O. Mcmahon, and F.-x. Pasquier, "Par4All : From Convex Array Regions to Heterogeneous Computing," pp. 1–2, 2012.

[42] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Transactions on Architecture and Code Optimization*, vol. 9, pp. 1–23, jan 2013.

[43] J. Jeffers and J. Reinders, *Intel Xeon Phi coprocessor high-performance programming*. Morgan Kaufmann, 2013.

[44] J. J. Irwin, T. Sterling, M. M. Mysinger, E. S. Bolstad, and R. G. Coleman, "ZINC: a free tool to discover chemistry for biology.," *Journal of chemical information and modeling*, vol. 52, no. 7, pp. 1757–68, 2012.

[45] J.-L. Reymond and M. Awale, "Exploring chemical space for drug discovery using the chemical universe database," *ACS chemical neuroscience*, vol. 3, no. 9, pp. 649–657, 2012.

[46] P. Ripphausen, B. Nisius, L. Peltason, and J. Bajorath, "Quo vadis, virtual screening? A comprehensive survey of prospective applications," *Journal of medicinal chemistry*, vol. 53, no. 24, pp. 8461–8467, 2010.

[47] D. E. Clark, "What has virtual screening ever done for drug discovery?," *Expert Opinion on Drug Discovery*, vol. 3, no. 8, pp. 841–51, 2008.

[48] P. D. Karp, B. Berger, D. Kovats, T. Lengauer, M. Linial, P. Sabeti, W. Hide, and B. Rost, "ISCB Ebola Award for Important Future Research on the Computational Biology of Ebola Virus," *PLoS Comput Biol*, vol. 11, no. 1, 2015.

[49] T. Cheng, Q. Li, Z. Zhou, Y. Wang, and S. H. Bryant, "Structure-based virtual screening for drug discovery: a problem-centric review.," *The AAPS journal*, vol. 14, pp. 133–41, mar 2012.

[50] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne, "The protein data bank," *Nucleic acids research*, vol. 28, no. 1, pp. 235–242, 2000.

[51] D. M. Kruger, G. Jessen, and H. Gohlke, "How Good Are State-of-the-Art Docking Tools in Predicting Ligand Binding Modes in ProteinProtein Interfaces?," *Journal of chemical information and modeling*, vol. 52, no. 11, pp. 2807–2811, 2012.

[52] M. Totrov and R. Abagyan, "Flexible ligand docking to multiple receptor conformations: a practical alternative," *Current opinion in structural biology*, vol. 18, no. 2, pp. 178–184, 2008.

[53] M. A. Lill, "Efficient incorporation of protein flexibility and dynamics into molecular docking simulations," *Biochemistry*, vol. 50, pp. 6157–6169, 2011.

[54] H. Merlitz and W. Wenzel, "Comparison of stochastic optimization methods for receptorligand docking," *Chemical physics letters*, vol. 362, no. August, pp. 271–277, 2002.

[55] O. Trott and A. Olson, "AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading," *Journal of Computational Chemistry*, vol. 31, no. 2, pp. 455–461, 2010.

[56] M. Simonsen, C. N. S. Pedersen, and M. H. Christensen, "GPU-accelerated high-accuracy molecular docking using guided differential evolution: real world applications," *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pp. 1803–1810, 2011.

[57] O. Korb, T. Stützle, and T. E. Exner, "Accelerating molecular docking calculations using graphics processing units," *Journal of chemical information and modeling*, vol. 51, pp. 865–76, apr 2011.

[58] G. D. Guerrero, H. E. Perez-Snchez, J. M. Cecilia, and J. M. Garcia, "Parallelization of Virtual Screening in Drug Discovery on Massively Parallel Architectures," *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pp. 588–595, feb 2012.

[59] A. M.-S. Simon N, P. James R, S. Richard B, and Avila Ibarra, "High performance in silico virtual drug screening on many-core processors," *International Journal of High Performance Computing Applications*, vol. 29, no. 2, pp. 119–134, 2014.

[60] Y. Ding, Y. Fang, W. P. Feinstein, J. Ramanujam, D. M. Koppelman, J. Moreno, M. Brylinski, and M. Jarrell, "GeauxDock: A novel approach for mixed-resolution ligand docking using a descriptor-based force field," *Journal of Computational Chemistry*, vol. 36, no. 27, pp. 2013–2026, 2015.

[61] M. Brylinski and W. P. Feinstein, "eFindSite: improved prediction of ligand binding sites in protein models using meta-threading, machine learning and auxiliary ligands," *Journal of computer-aided molecular design*, vol. 27, no. 6, pp. 551–567, 2013.

[62] M. Brylinski and J. Skolnick, "Q-Dock: Low-resolution flexible ligand docking with pocket-specific threading restraints," *Journal of computational chemistry*, vol. 29, no. 10, pp. 1574–1588, 2008.

[63] D. J. Earl and M. W. Deem, "Parallel tempering: theory, applications, and new perspectives," *Physical chemistry chemical physics : PCCP*, vol. 7, no. 23, pp. 3910–3916, 2005.

[64] H. G. Katzgraber, "Introduction to Monte Carlo Methods," *arXiv*, 2009.

[65] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI : A Portable Interface to Hardware Performance Counters," *Proceedings of the Department of Defense HPCMP Users Group Conference*, pp. 7–10, 1999.

[66] P. S. Charifson, L. M. Shewchuk, W. Rocque, C. W. Hummel, S. R. Jordan, C. Mohr, G. J. Pacofsky, M. R. Peel, M. Rodriguez, and D. D. Sternbach, "Peptide ligands of pp60c-src SH2 domains: a thermodynamic and structural study," *Biochemistry*, vol. 36, no. 21, pp. 6283–6293, 1997.

[67] M. J. Hartshorn, M. L. Verdonk, G. Chessari, S. C. Brewerton, W. T. M. Mooij, P. N. Mortenson, and C. W. Murray, "Diverse, high-quality test set for the validation of protein-ligand docking performance.," *Journal of medicinal chemistry*, vol. 50, pp. 726–41, feb 2007.

[68] A. K. Ghose, V. N. Viswanadhan, and J. J. Wendoloski, "A Knowledge-Based Approach in Designing Combinatorial or Medicinal Chemistry Libraries for Drug Discovery. 1. A Qualitative and Quantitative Characterization of Known Drug Databases," *Journal of Combinatorial Chemistry*, vol. 1, no. 1, pp. 55–68, 1999.

[69] E. Blem, J. Menon, and K. Sankaralingam, "A detailed analysis of contemporary arm and x86 architectures," *UW-Madison Technical Report*, 2013.

[70] R. Raag and T. L. Poulos, "The structural basis for substrate-induced changes in redox potential and spin equilibrium in cytochrome P-450CAM," *Biochemistry*, vol. 28, no. 2, pp. 917–922, 1989.

[71] L.-N. Pouchet, "PoCC," 2013.

[72] N. Y. Chirgadze, D. J. Sall, V. J. Klimkowski, D. K. Clawson, S. L. Briggs, R. Hermann, G. F. Smith, D. S. GiffordMoore, and J. Wery, "The crystal structure of human α-thrombin complexed with LY178550, a nonpeptidyl, active site-directed inhibitor," *Protein science*, vol. 6, no. 7, pp. 1412–1417, 1997.

[73] D. Rognan, "Beware of Machine Learning-Based Scoring Functions-On the Danger of Developing Black Boxes," *Journal of chemical information and modeling*, vol. 54, no. 10, pp. 2807–2815, 2014.

[74] T. T. Vu and B. Derbel, "Link-heterogeneous work stealing," *Proceedings - 14th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2014*, pp. 354–363, 2014.

[75] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," *ACM SIGPLAN Notices*, vol. 33, no. 5, pp. 212–223, 1998.

[76] S. Marcos and S. Marcos, "Task scheduling on multicores under energy and power constraints," pp. 9–12, 2013.

[77] K. Vivekanandarajah and S. K. Pilakkat, "Task mapping in heterogeneous MPSoCs for system level design," *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems, ICECCS*, pp. 56–65, 2008.

[78] F. Glover, M. Laguna, and R. Mart, "Tabu search," 1997.

[79] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.

[80] M. Maheswaran, S. Ali, H. Siegal, D. Hensgen, and R. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems," *Proceedings. Eighth Heterogeneous Computing Workshop (HCW'99)*, 1999.

[81] R. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. Lima, F. Mirabile, L. Moore, B. Rust, and H. Siegel, "Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet," *Proceedings Seventh Heterogeneous Computing Workshop (HCW'98)*, 1998.

[82] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems," *Journal of Parallel and Distributed Computing*, vol. 59, no. 2, pp. 107–131, 1999.

[83] M.-y. Wu and W. Shu, "A High-Performance Mapping Algorithm for Heterogeneous Computing Systems," *Simulation*, vol. 00, no. C, pp. 8–13, 2001.

[84] H. Schmeck, C. Müller-Schloer, E. Çakar, M. Mnif, and U. Richter, "Adaptivity and self-organization in organic computing systems," *ACM Trans. Auton. Adapt. Syst.*, vol. 5, pp. 10:1–10:32, Sept. 2010.

[85] Y. Fang, S. Feng, K. M. Tam, Z. Yun, J. Moreno, J. Ramanujam, and M. Jarrell, "Parallel tempering simulation of the three-dimensional Edwards-Anderson model with compact asynchronous multispin coding on GPU," *Computer Physics Communications*, vol. 185, no. 10, pp. 2467–2478, 2014.

[86] M. M. Mysinger, M. Carchia, J. J. Irwin, and B. K. Shoichet, "Directory of useful decoys, enhanced (dud-e): better ligands and decoys for better benchmarking," *Journal of medicinal chemistry*, vol. 55, no. 14, pp. 6582–6594, 2012.

[87] T. D. Braunt, H. J. Siegel, N. Beck, L. L. Boloni, and M. Maheswarans, "A Comparison Study of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Ileterogeneous Distributed Computing Systems," *Distributed Computing*, 2000.

[88] H. Izakian, A. Abraham, and V. Snášel, "Comparison of heuristics for scheduling independent tasks on heterogeneous distributed environments," *Proceedings of the 2009 International Joint Conference on Computational Sciences and Optimization, CSO 2009*, vol. 1, pp. 8–12, 2009.

[89] Rafsanjani and Bardsiri, "A New Heuristic Approach for Scheduling Independent Tasks on Grid Computing Systems.," *... Distributed Computing*, vol. 6, no. 4, pp. 371–376, 2013.

[90] S. Pandruvada, *Running Average Power Limit RAPL*, 2014.

[91] A. C. de Melo, "The new linuxperftools," in *Slides from Linux Kongress*, vol. 18, 2010.

[92] M. Barth and M. Schliephake, "Best Practice Guide Intel Xeon Phi v1 . 1," no. February, pp. 1–33, 2014.

[93] H. McCraw, J. Ralph, A. Danalis, and J. Dongarra, "Power monitoring with papi for extreme scale architectures and dataflow-based programming models," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 385–391, IEEE, 2014.

[94] NVIDIA, "Nvidia management library (nvml)," 2016.

Vita

Ye Fang was born in Nanchang and raised in Jinan, China. He obtained his bachelor's degree in electrical engineering from ShanDong University at Weihai, in June 2009. After that he has been studying at Louisiana State University, in pursuit of his Ph.D. degree in electrical engineering. In May 2016, he earned a masters degree in electrical engineering in Louisiana State University. He anticipates graduating with his Ph.D. degree in Spring 2017.