Computers in Biology and Medicine 43 (2013) 1011–1022

Contents lists available at SciVerse ScienceDirect

Computers in Biology and Medicine

journal homepage: www.elsevier.com/locate/cbm

GPU-based acceleration of an RNA tertiary structure prediction algorithm

Yongkweon Jeon [a], Eesuk Jung [b], Hyeyoung Min [c], Eui-Young Chung [b], Sungroh Yoon [a,d,*]

[a] Department of Electrical and Computer Engineering, Seoul National University, Seoul 151-744, Republic of Korea
[b] Department of Electrical and Electronic Engineering, Yonsei University, Seoul 120-749, Republic of Korea
[c] RNA Biopharmacy Laboratory, College of Pharmacy, Chung-Ang University, Seoul 156-756, Republic of Korea
[d] Bioinformatics Institute, Seoul National University, Seoul 151-747, Republic of Korea

ARTICLE INFO

Article history:
Received 3 November 2012
Accepted 14 May 2013

Keywords:
RNA
RNA structure prediction
Parallel algorithm
Multi-core CPU
GPGPU

ABSTRACT

Experimental techniques such as X-ray crystallography and nuclear magnetic resonance have been useful for the accurate determination of RNA tertiary structures. However, high-throughput structure determination using such methods often becomes difficult, due to the need for a large quantity of pure samples. Computational techniques for the prediction of RNA tertiary structures are thus becoming increasingly popular. Most of the existing prediction algorithms are computationally intensive, and there is a clear need for acceleration. In this paper, we propose a parallelization methodology for the fragment assembly of RNA (FARNA) algorithm, one of the most effective methods for computational prediction of RNA tertiary structure. The proposed parallelization scheme exploits multi-core CPUs and GPUs in harmony to maximize their utilization. We tested our approach with a number of RNA sequences and confirmed that it allows the time required for structure prediction to be significantly reduced. With respect to the baseline architecture equipped with a single CPU core, we achieved a speedup of up to approximately 24 × (roughly 4× by multi-core CPUs and 20× by GPUs). Compared with a quad-core CPU setup, the proposed approach delivers an additional 12× speedup by utilizing GPU devices. Given that most PCs these days have a multi-core CPU and a GPU card, our methodology will be very helpful for accelerating algorithms in a cost-effective manner.

1. Introduction

The role of RNA molecules in biology is amazingly diverse. They can carry and decode genetic information, work as part of protein-synthesizing machine, catalyze chemical reactions, and regulate gene expression. Such a variety is possible because RNAs can vary their tertiary structures they adopt in different conditions [1]. RNA tertiary structure determination is therefore important for understanding RNA functions and interactions. Experimental techniques such as X-ray crystallography and nuclear magnetic resonance (NMR) have been useful for the accurate determination of RNA tertiary structures, but high-throughput structure determination using such methods often becomes difficult, due to the need for a large quantity of high-purity samples [2]. Furthermore, there are a number of functionally important RNA states whose structures cannot be directly determined by high-resolution techniques [3,4]. To understand the structure–function relationships for these RNAs, we need accurate tertiary structure modeling [4].

The field of RNA structure modeling and prediction is thus receiving growing attention. There exist various approaches [5–10], and their common goal is to provide an accurate structural model of RNA and prediction methods useful for designing and verifying biological hypotheses [4]. The prediction of native-like RNA structures typically needs algorithms with high computational complexity and involves a huge number of precise floating-point number calculations [2]. The computation takes longer for a longer RNA sequence, and the prediction of a non-trivial RNA sequence can easily become prohibitively time consuming. Genomic databases are growing fast, and the need for a rapid prediction tool is becoming increasingly clear.

In this paper, we present a GPU-based parallelization scheme of the fragment assembly RNA (FARNA) algorithm [6] in the Rosetta software suite [11]. FARNA is one of the most powerful computational methods for modeling native-like RNA structural motifs with high resolution [1] and can recapitulate many non-Watson-Crick base pairs seen in native structures, which are crucial for accurate modeling. Ideally, prediction algorithms could be run on a supercomputer or a cluster of servers. Such a powerful computing environment is not available to every researcher, however, and affordable microprocessor-based PCs have been the workhorse for executing prediction algorithms in many cases. Another notable trend in computer architecture is the use of graphics processing units (GPUs) for general-purpose computing. These days, GPUs have hundreds of processing units embedded within them, and the growth in raw performance of GPUs has been outpacing

* Corresponding author at: Department of Electrical and Computer Engineering, Seoul National University, Seoul 151-744, Republic of Korea. Tel.: +82 2 880 1401; fax: +82 2 871 5974.
E-mail address: sryoon@snu.ac.kr (S. Yoon).

0010-4825/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.compbiomed.2013.05.007

that of CPUs. The many-core architectures such as GPUs are significantly more efficient in terms of arithmetic operations per units of energy or per transistor [12] and have been used for accelerating molecular modeling [13,14]. To the best of our knowledge, this paper is the first attempt to parallelize FARNA using GPUs.

2. Background

2.1. RNA structure prediction

In order to understand the function an RNA and its interaction with other molecules, it is critical to identify the tertiary structure of an RNA. However, the accurate determination of the three-dimensional structure and folding kinetics of RNAs remains challenging experimentally. It is often difficult to use X-ray crystallography or nuclear magnetic resonance for high-throughput structure determination, due to the need for the preparation of large quantities of RNA samples with high purity and technical limitations [2]. Consequently, the computational prediction of RNA tertiary structures and folding is becoming increasingly popular and important. Table 1 lists the existing computational methods for RNA tertiary prediction. In contrast to the advances made in algorithms for predicting protein folding, the computational prediction of RNA structures is still in its infancy.

There has been a great deal of interest in the modeling of 3D protein structures, and the methods developed for proteins may be useful in the context of RNA as well [15]. However, there exist key differences between RNA and protein structures, which must be considered when developing an algorithm for RNA structure prediction. Most obviously, an RNA sequence consists of only four types of bases, whereas a protein sequence can have 20 types of amino acids. Secondly, RNAs are highly negatively charged and can create strong intermolecular and/or intramolecular electrostatic interactions within and/or outside the molecules. In addition, the formation of the secondary and tertiary structures is much more clearly separated in the time domain for RNAs. Lastly, significant topological constraints can be put on the RNA molecule by the intimate intertwining of the two parts of the RNA strand.

2.2. Fragmented assembly of ribonucleic acid

The fragment assembly of RNA (FARNA) algorithm [6] was developed to predict RNA tertiary structure based on the minimum energy required to form RNA structure with great stability. FARNA was inspired by the low-resolution protein structure prediction algorithm included in the Rosetta suite [11]. The Rosetta software (https://www.rosettacommons.org) has been widely used for macromolecular modeling and includes tools for structure inference, design, and modeling of nucleic acids and proteins. For accurate modeling, it is critical to consider non-Watson–Crick pairs such as a wobble base pair and a Hoogsteen base pair [1], and FARNA cannot only reproduce canonical Watson–Crick pairs accurately but also recapitulate many of the non-Watson-Crick pairs seen in the native structures [6].

FARNA models an RNA sequence by a string of beads, each of which is assumed to be centered at a base. In this coarse-grained model, three bases are grouped together and considered at a time. Given a sequence of the target RNA, FARNA assembles short fragments from existing RNA crystal structures whose sequences match the subsequences of the target RNA. To this end, a 3D structure library is utilized. This library contains 3-nucleotide fragments taken from a large rRNA subunit, from which the torsion angles and ribose puckering parameters are extracted and stored. To assemble the fragments into native-like structures, FARNA relies on a Monte Carlo simulation, which is guided by a knowledge-based energy function that considers the backbone conformations and side-chain interactions in solved RNAs.

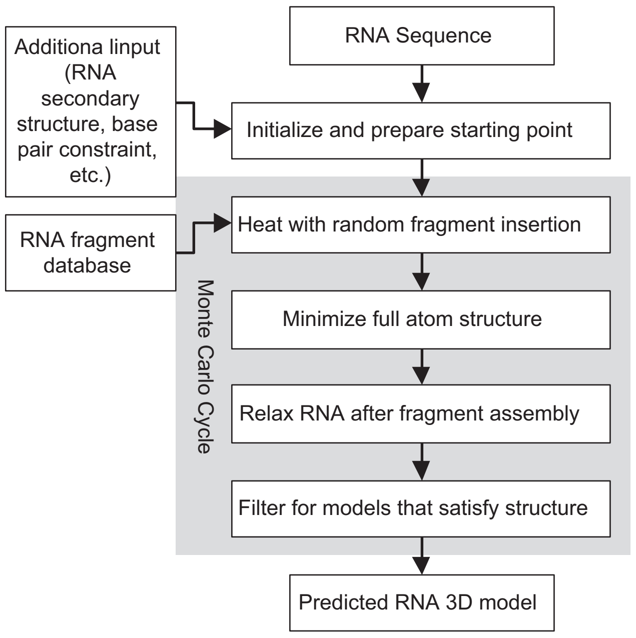
RNA Sequence
Additiona linput (RNA secondary structure, base pair constraint, etc.)
Initialize and prepare starting point
RNA fragment database
Heat with random fragment insertion
Minimize full atom structure
Relax RNA after fragment assembly
Filter for models that satisfy structure
Monte Carlo Cycle
Predicted RNA 3D model

Fig. 1. Overall flow of FARNA. Given an RNA sequence and additional input such as its secondary structure and base pair constraints, the FARNA algorithm carries out a sequence of steps to predict the tertiary structure of the input sequence. The main body of the algorithm consists of four steps (random fragment insertion, atomic structure minimization, relaxation after assembly, and filtering) based on Monte Carlo simulation cycles.

Table 1
Computational methods for RNA structure modeling and prediction. The existing computational methods for RNA tertiary prediction.

Tool	Input	Model	Simulation method	Description
BARNACLE [5]	Sequence	Coarse-grained	Relica exchange, molecular dynamics	A Python library for the probabilistic sampling of RNA structures that are compatible with a given nucleotide sequence and that are RNA-like on a local length scale
FARNA [6,7]	Sequence, secondary structure	Coarse-grained	Fragment assembly, Monte Carlo	Uses 3-nt fragment library, Monte Carlo simulations and a potential function to predict the structure
iFoldRNA [8]	Sequence	Coarse-grained	Replica exchange, molecular dynamics	Uses discrete molecular dynamics and force fields to simulate RNA folding dynamics
MC-Fold/MC-Sym [9]	Sequence, secondary structure	Nucleotide cyclic motif	Fragment assembly, Las Vegas algorithm	Predicts RNA structures using free energy minimization with structure assembled using the fragment insertion by Las Vegas algorithm
NAST [10]	Secondary structure, tertiary contacts	Coarse-grained	Molecular dynamics	Performs molecular dynamics simulations guided by a knowledge-based statistical potential function

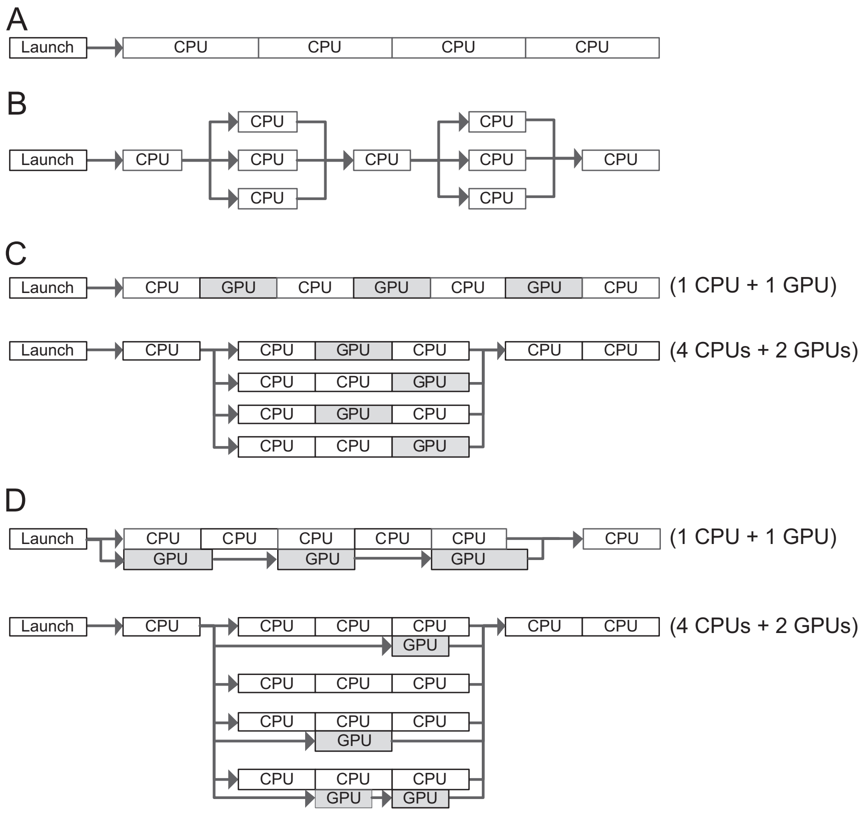

Fig. 2. Comparing different execution flows. The sequential execution in (a) corresponds to the one without any parallelization, (b) presents the fork–join model used by OpenMP and other software designed to facilitate parallelization, (c) shows the synchronous CPU/GPU hybrid execution flow and (d) depicts the asynchronous hybrid execution flow that we utilize in this paper.

Fig. 1 shows the overall flow of FARNA. The two major tasks involved are energy minimization and fragment assembly. The goal of the energy minimization process is to find the most stable state with the minimum energy. The energy function used is a sum of parameters such as the radius of gyration, penalty for steric clashes between atoms, base-paring potentials, coplanarity of pairing and stacking rewards. The fragment assembly task uses a Monte Carlo simulation [16] and, in each step of the simulation, a random position is chosen in the chain and the torsions for three bases are replaced with those from a randomly selected fragment. After initialization, the fragment assembler carries out 50,000 fragment insertions with the RNA energy function. To increase the accuracy, it is also possible to incorporate residue–residue interactions at the resolution of a single nucleotide using multiplexed hydroxyl radical ($^{\cdot}$OH) cleavage analysis (MOHCA) [7]. The incorporated residue-level interaction information is called *MOHCA constraints*. In the method section, we will elaborate the major steps involved in the FARNA algorithm, identify computational bottlenecks, and propose parallelization schemes of such rate-limiting steps.

2.3. Parallelization with CPU/GPU

Today, most PCs are equipped with multi-core CPUs, which can be exploited for accelerating bioinformatics algorithms. Programmers can utilize software tools that help them analyze their workload and eventually parallelize algorithms more easily. For shared-memory machines such as PCs with multi-core CPUs, OpenMP [17] has become popular. OpenMP provides an application programming interface (API) for programming in C, C++ and Fortran. A programmer can conveniently parallelize a piece of existing code by inserting compiler directives, API routines and environment variables. The early versions of OpenMP relied heavily on the fork–join model for multi-threading, in which a master thread dynamically creates ("forks") and merges ("joins") a number of slave threads. Additional details on the fork–join model and other related approaches can be found in Section 3.1.

The newest release of OpenMP (version 3.0 or later) additionally supports a new task-sharing model in which a unit of parallel work called *an explicit task* is used to express unstructured parallelism and to define dynamically generated units of work. Our approach described in this paper utilizes OpenMP 3.0 and this new task-sharing model, which was impossible to implement by the traditional fork–join model. Additional examples of parallelization aids for shared memory machines include POSIX threads (Pthreads) [18], which allows a lower level of abstraction and tools than OpenMP.

While we consider parallelization on CPUs as software multi-threading, parallelization via GPUs can be thought of as hardware multi-threading. The programmable GPU devices from NVIDIA are structured as arrays of multi-threaded streaming microprocessors [19]. CPU cores aim at processing a single instruction as fast as possible, but GPU cores try to increase the throughput of running a set of instructions. The compute unified device architecture (CUDA) developed by NVIDIA [19,20] is a development toolkit that programmers can exploit for expressing and accessing massively parallel graphic cores for general-purpose computing. Using CUDA API routines, programmers can modify their C/C++ code so that certain portions of it can be executed on a GPU device. Only those functions that can benefit from GPU acceleration need to be developed in CUDA, while the rest of the code can remain in C/C++. The CUDA-modified code will be executed on a GPU device, whereas the rest will be run on the host CPU. The memory architecture of a GPU is different from that of a GPU. For example, a CUDA device typically consists of registers, shared memory, global memory, constant memory and texture memory [20]. It is important to understand and exploit this memory hierarchy to maximize the degree of parallelization and performance of a GPU.

For better performance, we need to minimize the frequency and amount of data transferred between a GPU and its host CPU. This is because of high communication cost between them. When use the global memory in a GPU, we need to access it in a coalesced manner so that the access latency can be reduced. Additionally, we can exploit the fact that shared memory is normally much faster than the global memory in a GPU, although the former is smaller than the latter.

3. Methods

This section describes the proposed methodology for parallelizing the FARNA algorithm. Before describing our approach, we first compare the different execution flows for parallelization depicted in Fig. 2.

3.1. Execution flows

The sequential execution in Fig. 2(a) corresponds to the one without any parallelization. The CPU is utilized sequentially. Fig. 2(b) presents the fork–join model used by OpenMP [17] and other software designed to facilitate parallelization. In this model, the root process of the CPU spawns multiple child processes (or threads) when there is work to do in parallel and, when the work is done, the threads end and the master process takes up the flow and continues. Typically, each thread is assigned to a single CPU core. The above two models assume that there is no use of GPUs.

The top diagram in Fig. 2(c) shows the synchronous CPU/GPU hybrid execution flow, where the "host" CPU and its "device" GPU take turns in processing the workloads, but they never run simultaneously. For instance, the work proposed in [21,22] belongs to this category. In this model, a CPU acts as the host for a GPU and, after the CPU launches a kernel (or a set of threads) on the GPU, the CPU idles until the GPU completes its job and signals this to the CPU. This model is not much different from the sequential execution flow in the sense that only one processing core (either a host CPU or a GPU) is utilized at a time. Typically, different types of workloads are assigned to CPUs and GPUs in this model. This conventional model can be extended to a system with multiple CPUs and GPUs, as depicted in the bottom of Fig. 2(c). Still, the host (CPU) and device (GPU) pair never runs simultaneously, although multiple execution flows consisting of a CPU/GPU pair can exist.

In this work, we utilize the so-called *asynchronous hybrid execution flow*, as shown in Fig. 2(d). In this model, the host CPU launches a kernel for the GPU as in the conventional model, but the CPU does not idle waiting for the GPU to signal the CPU. Instead, the CPU performs its own task after launching a kernel for its device GPU. Consequently, the utilization of computing elements can be increased over the synchronous model. This asynchronous hybrid model is well supported by the newest OpenMP task constructs and the asynchronous communication capability of the CUDA framework NVIDIA recently provides.

Note that most algorithms that utilize GPUs are CPU/GPU hybrid in that the main algorithm is executed on CPU and computational kernels on CPU and GPU. Nonetheless, the use of the asynchronous execution flow is relatively new. The asynchronous hybrid execution flow can be implemented in any of the CUDA versions currently available.

3.2. Hierarchical examination of the FARNA algorithm

Fig. 3(a) shows the top-level flow of the FARNA algorithm [6]. It is implemented in C++, and a few notable classes include `ChemicalManager` (manages different sets of atoms and residues), `ScoreFunctionFactory` (a collection of functions to calculate energy scores), `MonteCarlo` (responsible for all the functions applying the Metropolis criterion in Monte Carlo simulation), `RNA_DeNovoProtocol` (handles the RNA modeling), and `RNA_Minimizer` (minimizes the RNA full atom structure). Step 1 carries out input argument processing, and Steps 2 and 3 initialize the `ChemicalManager` and `ScoreFunctionFactory` classes, respectively. Step 4 reads in the input sequences in a FASTA-formatted file. Steps 6–10 perform additional initialization. Step 11 is the main step responsible for applying the fragment assembly protocol to the input sequence, taking on average 93.3% of the total running time, according to our profiling. We parallelize Step 11, namely the `RNA_DeNovoProtocol::apply` function. Step 12 releases the `RNA_DeNovoProtocol` class and wraps up the whole procedure.

At the second-level, Fig. 3(b) shows more details of the `RNA_DeNovoProtocol::apply` function. Steps 13 and 14 initialize the starting positions of simulation and scoring functions. Step 15 is to insert random fragments and corresponds to the "Heat with random fragment insertion" step in Fig. 1. Through Steps 16–18, various position candidates are tried, and the minimization occurs in Step 19. This `RNA_Minimizer::apply` function corresponds to the "Minimize full atom structure" step in Fig. 1 and is to apply the loop-rebuild protocol to the minimization of the RNA full atom structure. Step 19 takes most of the running time of Step 12 (approximately 91.8% of the total running time) and is therefore the target of parallelization. Steps 20–22 correspond to the last two steps in Fig. 1, relaxing RNA after the assembly procedure and identifying the lowest scoring pose.

Lastly, Fig. 3(c) shows the details of Step 19, or the `RNA_Minimizer::apply` function. This step is to find the state in which the residue atoms are arranged in such a way that their pairwise energy is minimized. The minimization problem is formulated using a graph, where vertices correspond to residue atoms and the weight of an edge between two vertices represents their pairwise energy. The core in the minimization procedure is Step 25, which is to carry out the line minimization based on the minimum bracketing and Brent's method [23]. Note that this minimization is realized by traversing four paths in order, as indicated in Fig. 4. According to our profiling, paths 1–4 took on average 46.6%, 12.9%, 8.4%, and 22.6% of the total running time, respectively. The other steps in Fig. 4 took negligible time. Consequently, we targeted parallelizing the procedures inside Step 25, as will be explained next.

3.3. Identification of parallelization target and details of parallelization

Step 25 in Fig. 3(c) consists of many subroutines. Among these, Steps 27–30 are inappropriate for parallelization because there is strong loop dependency coming from iterative updates of parameters (Steps 27, 28, and 30) or no loop to exploit for parallelization (Step 29). Steps 31–35 calculate the pairwise energy between edges in the minimization graph stated above, and provide ample opportunities for parallelization because of lack of dependency between calculating energy for different states.

We map the nodes (residues in RNA) in the minimization graph to GPU blocks and let each thread in a block compute the pairwise energy between edges in the graph. More specifically, the nested for-loop inside Step 33 is assigned to GPU block, and each thread in a block runs Step 35, which is composed of nested for-loops for energy computation. (Step 34 needs no parallelization in that it simply calls Step 35 once). The number of kernels executed is the same as the number of calling Step 33. Note that the thread assignment is independent of the input sequence length, because the number of atoms in a residue does not change. (As the

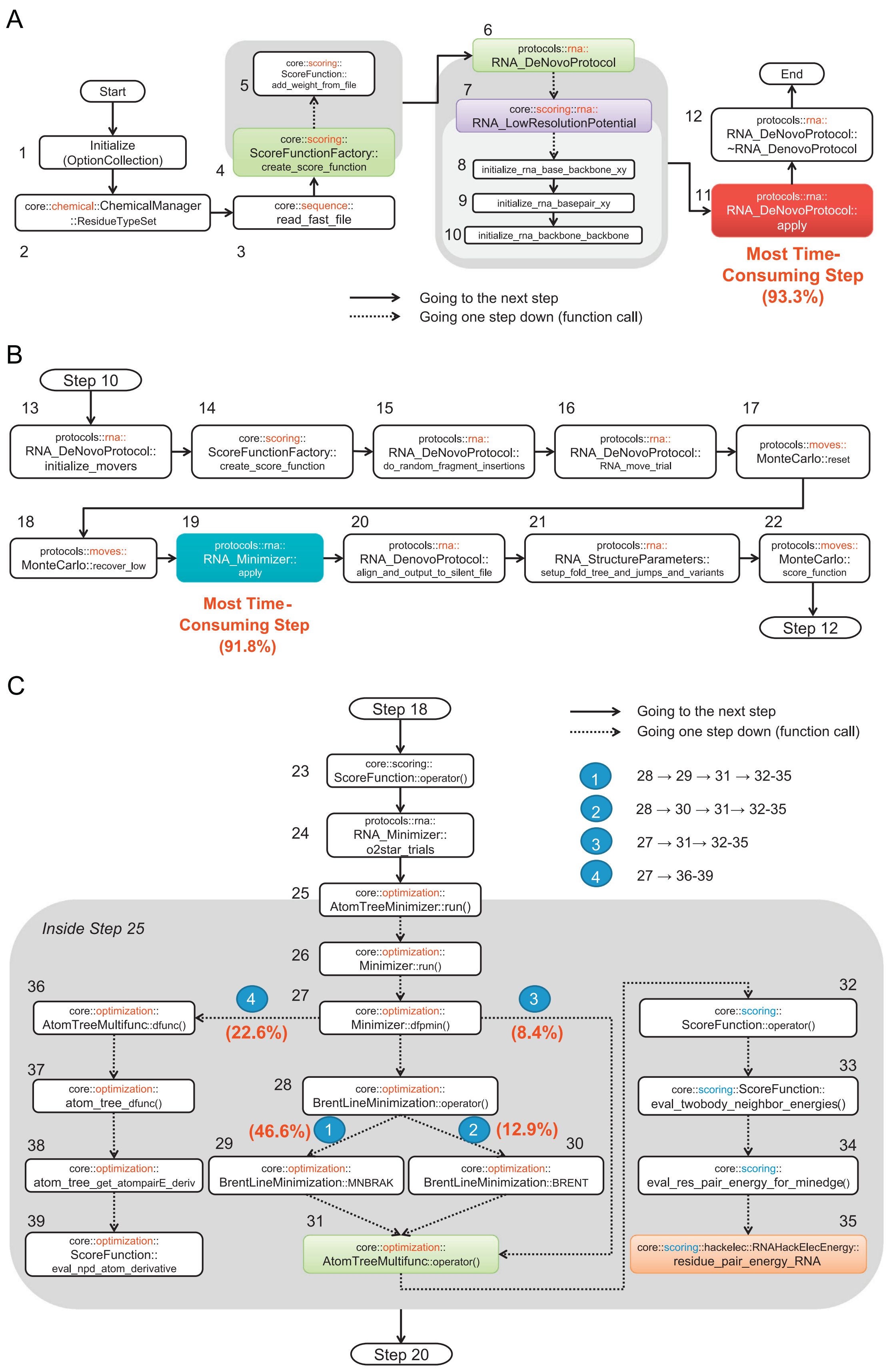
A
Start
1 Initialize (OptionCollection)
2 core::chemical::ChemicalManager ::ResidueTypeSet
3 core::sequence:: read_fast_file
4 core::scoring:: ScoreFunctionFactory:: create_score_function
5 core::scoring:: ScoreFunction:: add_weight_from_file
6 protocols::rna:: RNA_DeNovoProtocol
7 core::scoring::rna:: RNA_LowResolutionPotential
8 initialize_rna_base_backbone_xy
9 initialize_rna_basepair_xy
10 initialize_rna_backbone_backbone
11 protocols::rna:: RNA_DeNovoProtocol:: apply
12 protocols::rna:: RNA_DeNovoProtocol:: ~RNA_DenovoProtocol
End
Most Time-Consuming Step (93.3%)
Going to the next step
Going one step down (function call)
B
Step 10
13 protocols::rna:: RNA_DeNovoProtocol:: initialize_movers
14 core::scoring:: ScoreFunctionFactory:: create_score_function
15 protocols::rna:: RNA_DeNovoProtocol:: do_random_fragment_insertions
16 protocols::rna:: RNA_DeNovoProtocol:: RNA_move_trial
17 protocols::moves:: MonteCarlo::reset
18 protocols::moves:: MonteCarlo::recover_low
19 protocols::rna:: RNA_Minimizer:: apply
20 protocols::rna:: RNA_DenovoProtocol:: align_and_output_to_silent_file
21 protocols::rna:: RNA_StructureParameters:: setup_fold_tree_and_jumps_and_variants
22 protocols::moves:: MonteCarlo:: score_function
Step 12
Most Time-Consuming Step (91.8%)
C
Step 18
Going to the next step
Going one step down (function call)
1 28 → 29 → 31 → 32-35
2 28 → 30 → 31→ 32-35
3 27 → 31→ 32-35
4 27 → 36-39
23 core::scoring:: ScoreFunction::operator()
24 protocols::rna:: RNA_Minimizer:: o2star_trials
25 core::optimization:: AtomTreeMinimizer::run()
Inside Step 25
26 core::optimization:: Minimizer::run()
27 core::optimization:: Minimizer::dfpmin()
36 core::optimization:: AtomTreeMultifunc::dfunc()
37 core::optimization:: atom_tree_dfunc()
38 core::optimization:: atom_tree_get_atompairE_deriv
39 core::optimization:: ScoreFunction:: eval_npd_atom_derivative
28 core::optimization:: BrentLineMinimization::operator()
29 core::optimization:: BrentLineMinimization::MNBRAK
30 core::optimization:: BrentLineMinimization::BRENT
31 core::optimization:: AtomTreeMultifunc::operator()
32 core::scoring:: ScoreFunction::operator()
33 core::scoring::ScoreFunction:: eval_twobody_neighbor_energies()
34 core::scoring:: eval_res_pair_energy_for_minedge()
35 core::scoring::hackelec::RNAHackElecEnergy:: residue_pair_energy_RNA
(22.6%)
(8.4%)
(46.6%)
(12.9%)
Step 20

Fig. 3. Details of implementation of FARNA algorithm. This figure shows the details of the FARNA implementation in a hierarchical fashion, (a) presents the top-level view of the major steps of FARNA, among which the `RNA_DeNovoProtocol::apply` function (Step 12) is the most time consuming, (b) magnifies Step 12, which consists of a series of procedures to drive the minimum energy configuration. Step 19 is the most time-consuming one, which is further detailed in (c). There are four paths in (c), and the exact sequence of steps for each path is shown in the legend.

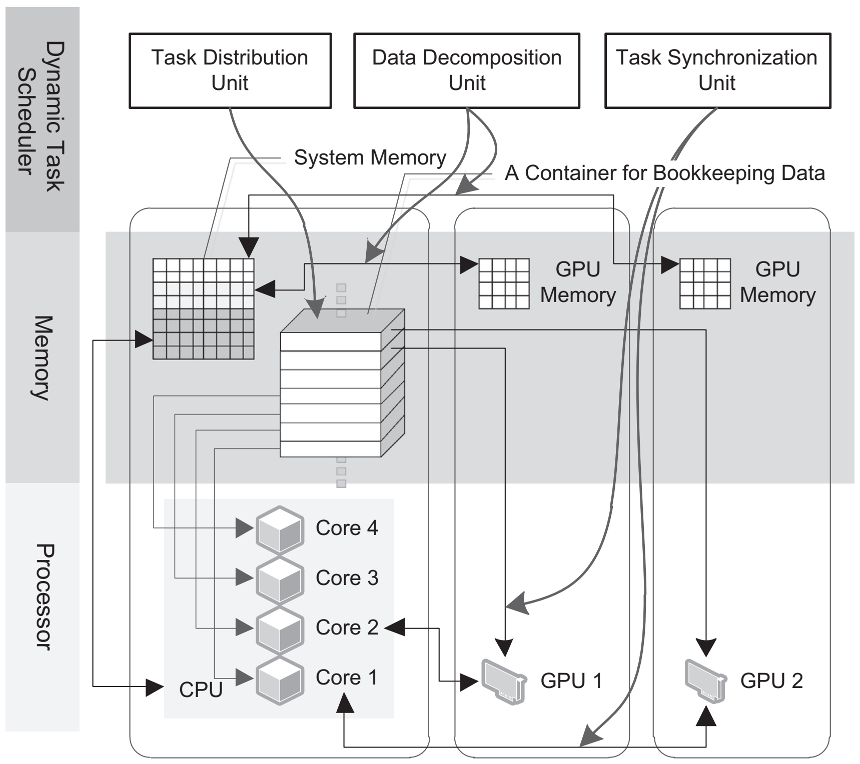

Fig. 4. Overview of dynamic task scheduler. This figure shows the overall architecture of the dynamic task scheduler, which is responsible for dynamically scheduling tasks for CPUs and GPUs.

sequence length increases, only the number of blocks changes, because the number of residue increases.)

As for using GPU memory, we use the shared memory in order to store the pairwise energy values computed from a block in a coalesced fashion. Regarding the host-to-device transfer, we need to send the information on the edges and nodes in the minimization graph: the base sequence, the tables for storing various statistics and the energy parameters for computing the loop energies. The number of transfers is roughly the same as the number of edges in the graph. By using the asynchronous execution flow, the overhead of this transfer is reduced. To implement the asynchronous hybrid execution flow, we exploited the asynchronous communication capability NVIDIA provides. On top of this capability, the notion of pinned memory is defined, and the host CPU and the GPU device can share the same memory space. The number of the device-to-host transfer is identical to the number of kernel executions. This overhead can also be alleviated by using the asynchronous execution flow.

To increase the utilization of CPU, we implemented the software called *the dynamic task scheduler*, which is responsible for scheduling tasks for CPUs and GPUs dynamically. Fig. 4 shows the overall architecture of our approach to implementing this hybrid parallel execution flow. For the sake of simplicity, we assume in the figure that there are four CPU cores and two GPU modules in the system. The dynamic task scheduler consists of three units: the data decomposition unit (DDU), the thread distribution unit (TDU) and the thread synchronization unit (TSU). These units cooperate in order to decompose the data to be processed, distribute workloads to computing elements and synchronize them.

3.4. Details of dynamic task scheduler

We provide more details of the dynamic task scheduler, which can be explained best by using an example, as shown in Fig. 5. As before, we assume that the system has four CPU cores and two GPU modules installed. Furthermore, we assume that each GPU module has eight GPU cores. We also assume that the input is an array of 99 elements of a certain type.

First, the data decomposition unit examines the input array and partitions it into three chunks, as shown in Fig. 5 (steps 1 and 2). Two of these three chunks have eight elements each, and each chunk will be processed by a GPU module. The remaining chunk has 83 (=99−16) elements and will be dynamically processed by either the CPU cores or the GPU cores. Each of the two chunks with eight elements is assigned a single data ID, since all of the elements in each of these chunks will be collectively copied from the main memory to the device memory. In contrast, each element in the largest chunk with 83 elements is assigned a data ID for the time being (as will be explained shortly, the elements in this largest chunk can be grouped into new chunks consisting of eight elements each later).

Once the data decomposition has completed its task, the thread distribution and thread synchronization units come into the picture. We first introduce the data structures used by these two units. As indicated in Fig. 5, the thread distribution unit keeps two tables, namely the ongoing-task list and the upcoming-task list. The former is used to store the information of the tasks that are currently being executed in a processing element (either a GPU or CPU), whereas the latter is used to manage the tasks that are waiting for assignment to processing elements. The thread synchronization unit maintains the idle status of each processing element in the system using a table called the processor status table.

Now, we describe how the dynamic task scheduler works by exploiting the bookkeeping information stored in these tables. In step 3, the upcoming-task list is filled with two entries, one for each GPU module. Each entry in this table contains a task index (TI) and a data index (DI). For a GPU, each TI corresponds to the kernel (*i.e.*, a set of threads in the GPU card) index; for a CPU, each TI indicates the thread index executed on the CPU core. (For

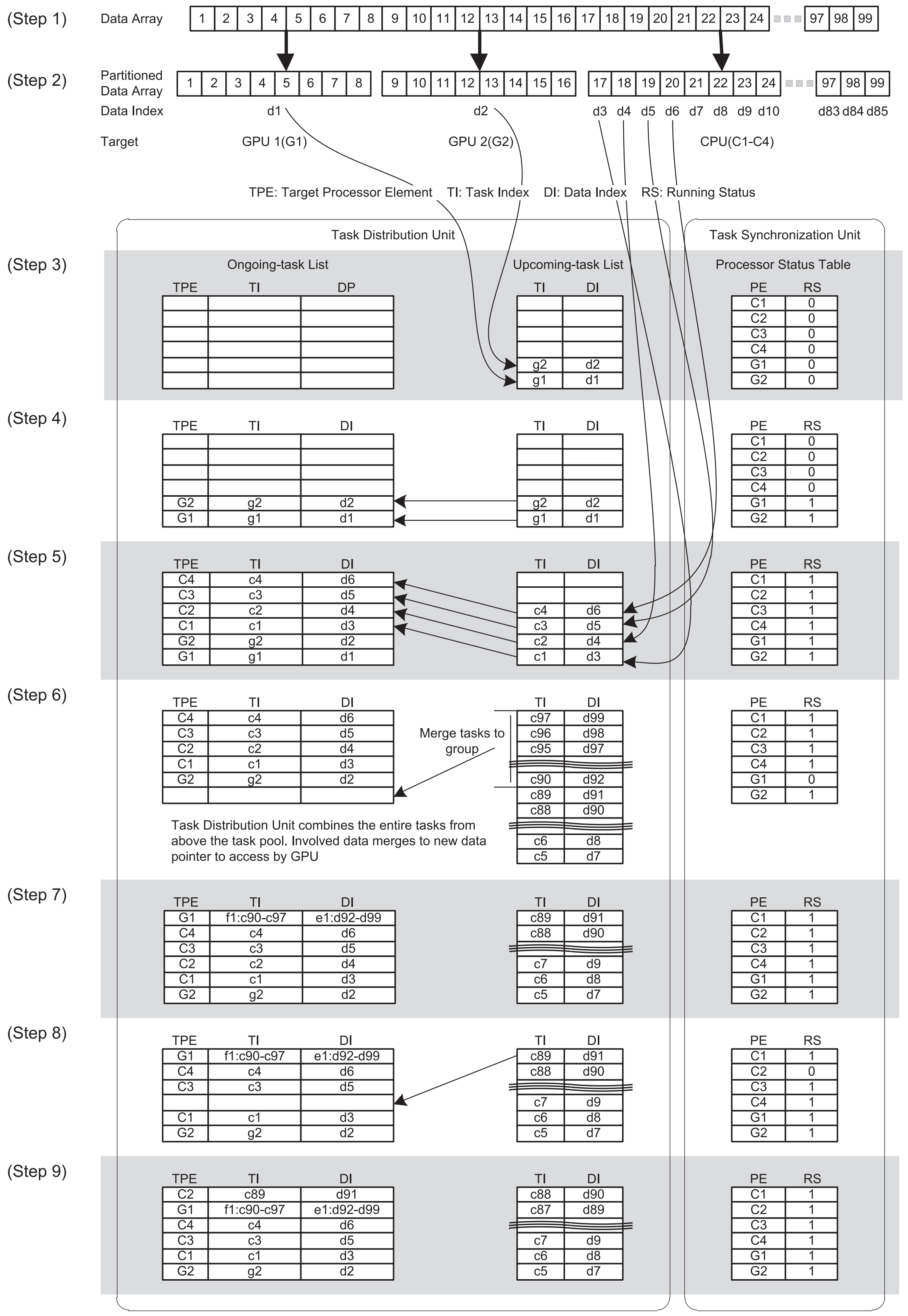

Fig. 5. Dynamic task scheduler. Example explaining how the dynamic task scheduler works.

convenience, a GPU task index has prefix g and a CPU task index has prefix c in this example.) A DI indicates the index of the data chunk or the elements, as explained above. At this moment, no processing element has work to do, and this information is recorded in the processor status table. In this table, each entry contains the processing element ID (C* for a CPU and G* for a GPU in this specific example) and its running status (0 for running and 1 for idle).

In step 4, the two entries in the upcoming-task list are moved to the ongoing-task list. That is, data chunks d1 and d2 are copied from the main memory to the device memory and the two GPU modules start working. The processor status table indicates this fact by changing the status of processing elements G1 and G2 from 0 to 1. Similarly, in step 5, elements d3–d6 are moved from the upcoming-task list to the ongoing-task list, indicating that these elements are assigned to the four CPU cores for processing. Now, all of the processing elements in the system are busy, and the RS fields in the processor status table are all set to one.

Step 6 is included to explain the situation in which a GPU module completes its task and becomes idle. For instance, in the ongoing-task list, we do not see the entry for G1 any more, and the RS field for entry G1 in the processor status table is 0. This indicates that G1 can receive a new workload and start working. Unlike a CPU core, we assume that a GPU module has eight cores in this example. Consequently, eight elements in the upcoming-task list are grouped into a new data chunk and are assigned to G1. This chunk, indexed by e1, contains d92–d99 or the top eight entries in the upcoming-task list. In step 7, the ongoing-task list now shows that G1 is processing data chink e1 (or equivalently d92–d99).

In step 8, the ongoing-task list and the processor status table indicate that CPU core C2 is done with its job. Consequently, a new data element (d91) is assigned to this processing element, and it resumes working in step 9.

4. Results and discussion

4.1. Experiment setup

We measured the runtime of the parallelized FARNA algorithm (pFARNA) using 52 different architecture combinations. We used the four types of CPUs and six types of GPUs listed in Tables 2 and 3, respectively, in order to generate these combinations. Note that we use a unique identifier for each type of CPU or GPU for notational convenience.

We generated the 52 combinations mentioned above as follows: First, we executed pFARNA on each of the four types of CPUs without utilizing GPUs at all. Second, we combined each of the four CPU types and each of the six GPU types, giving 24 additional

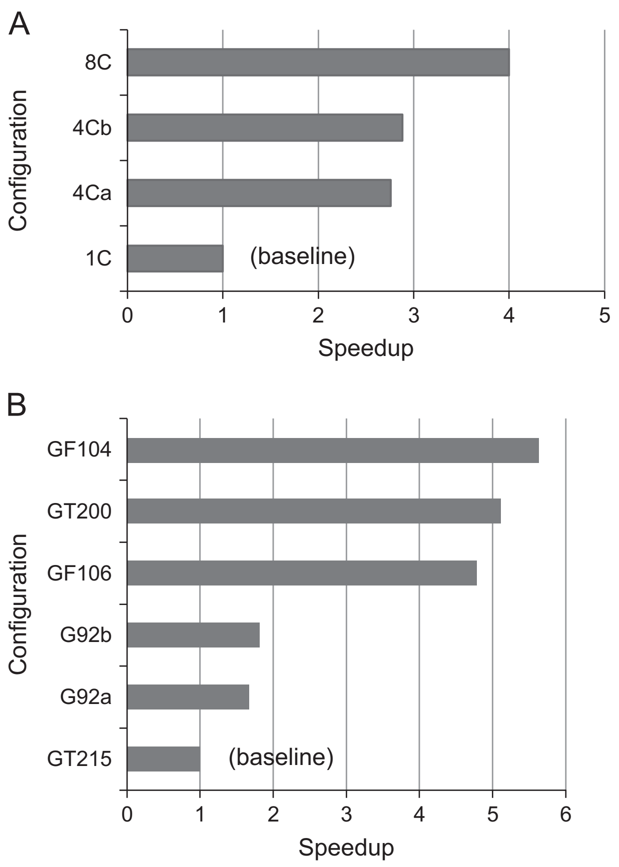

Fig. 6. Speedup of different CPU/GPU. This figure shows the speedup of different configurations with respect to the CPU baseline (1C) and the GPU baseline (GT215). Fig. 6(a) reveals that the speedup is not linear to the number of CPU cores due to data dependencies and other complications hindering parallelization. Examining Fig. 6(b) and Table 3 suggests that the performance is roughly proportional to the memory bandwidth and the amount of GPU cores used.

Table 2
CPU Configuration. The CPU configurations used in the experiments.

Configuration ID	**1C**	**8C**	**4Ca**	**4Cb**
Processor	2 × Intel Xeon E5620		Intel Core 2 Quad Q6600	Intel Core i5-750
Number of cores	Only 1 of 8 cores used	8	4	4
Frequency (GHz)	2.4		2.4	2.66
L2 Cache	256 KB per core		4 MB per two cores	256 KB per core
L3 Cache	8 MB per processor		None	8 MB
System memory (GB)	12		8	

Table 3
GPU Configuration. The GPU configurations used in the experiments.

Configuration ID	**G92a**	**G92b**	**GT200**	**GT215**	**GF104**	**GF106**
Processor (NVIDIA)	8800 GT	GTS 250	GTX 260	GT 240	GTX 460	GTS 450
Number of cores	112	128	216	96	336	192
Frequency (MHz)	1500	1836	1242	1340	1350	1566
Memory (MB)	512	512	896	512	768	1024
Memory bandwidth (GB/s)	57.6	70.4	111.9	32	86.4	57.73

combinations. Finally, we considered 24 additional scenarios by adding an extra GPU card to each of the previous 24 combinations.

The machine in each configuration was equipped with DDR3 main memory (size indicated in Table 2) and a single 1 TB SATA2 hard disk drive. The operating system used was 64-bit Microsoft Windows Vista Business, and for the development of pFARNA, we used the NVIDIA CUDA Toolkit 3.1 and Intel C++ compiler version 11.1 with OpenMP 3.0 support.

As the input to pFARNA, we prepared 32 RNA sequences from the nucleic acid database (NDB) [24]. Given that the main usage of FARNA is to predict the structure of small RNAs, the average length of the 32 sequences used was 54, with the minimum and maximum lengths being 6 and 158, respectively. To test pFARNA with the MOHCA constraints mentioned earlier, we also prepared the 158-nucleotide P4–P6 domain of the group I intron (PDB ID: 1GID) with its MOHCA constraints incorporated. Note that incorporating the MOHCA constraints typically increases the running time of FARNA significantly, and the running time for this 1GID sequence was the longest in our experiments.

4.2. Speedup by basic configurations

As the most basic test, we checked how the running time of pFARNA is affected by using different CPU and GPU cores. Fig. 6(a) shows the speedup of 4Ca, 4Cb and 8C over the reference case 1C. Having more cores seemed helpful to reduce the runtime, but the speedup was not directly proportional to the number of cores, but rather more limited. Although the FARNA code contains many parts appropriate for parallelization, there still exist data dependencies and other complications unsuitable for multi-threading. The non-ideality presented in this plot reflects these limitations. This figure also suggests that simply adding more CPU cores would not be very helpful for acceleration.

Fig. 6(b) compares the speedups of the different GPU cores with respect to the baseline architecture, GT215. All of the results include the GPU kernel launch overhead, as well as the PCI-Express data transfer overhead. Note that adding GF104 or GT200 to the system accelerates the algorithm to a similar extent as that obtained using configuration 8C. We observed that the performance was roughly proportional to the memory bandwidth and the number of cores in the system.

4.3. Speedup by various configurations

For each of the 32 input sequences, we measured the running time of pFARNA on each of the 52 configurations. Fig. 7 shows the speedup of each configuration over the baseline architecture 1C for the two sequences 1GID (with 158 nucleotides) and 1XJR (with 47 nucleotides). To see the effect of doubling the number of GPU cards, the speedups for the 24 single-GPU configurations and the 24 double-GPU configurations are shown together in the plot. Evidently, using GPUs on top of multi-core CPUs further accelerated the execution of pFARNA significantly. For 1GID (the most time-consuming case in our experiments), the maximum speedup

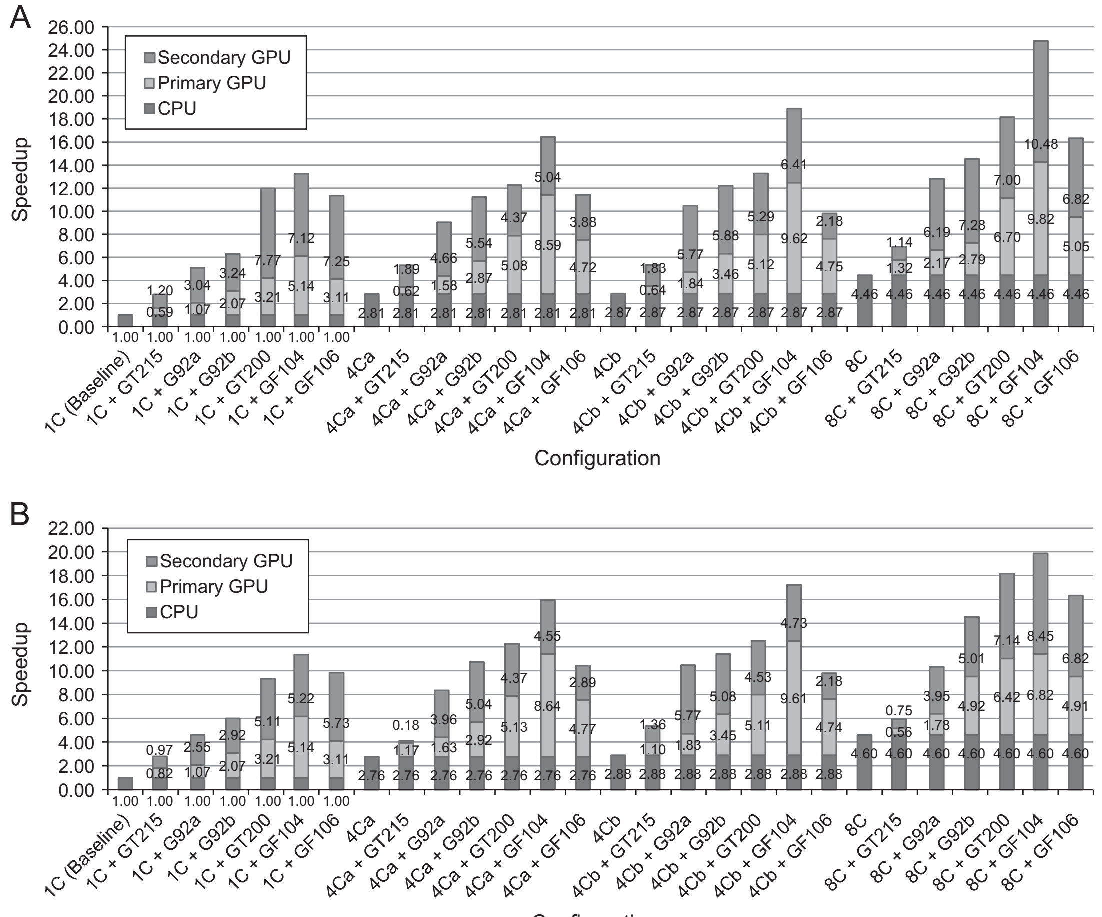

Fig. 7. Speedup of each configuration per the baseline architecture. The speedup of each configuration over the baseline architecture 1C for the two sequences: (a) 1GID (with 158 nucleotides) with MOCHA constraints and (b) 1XJR (with 47 nucleotides). Each bar shows the breakdown of speedups by CPU, primary GPU, and secondary GPU. The overall speedups can be obtained by cumulating the breakdowns. This breakdown of speedups can also be useful for comparing various configurations such as the GPU/CPU hybrid and GPU only cases.

Table 4
Comparison of Speedup. This table summarizes the speedup of the various configurations with respect to the 1C case.

PDB ID	CPU	Without GPU	With GPU	
			Minimum	Maximum
1GID	1C	1.00	1.59 (GT215)	10.36 (2 × GF104)
	4Ca	2.81	3.43 (GT215)	16.44 (2 × GF104)
	4Cb	2.87	3.46 (GT215)	18.90 (2 × GF104)
	8C	4.46	5.78 (GT215)	24.76 (2 × GF104)
1XJR	1C	1.00	1.82 (GT215)	11.36 (2 × GF104)
	4Ca	2.76	4.93 (GT215)	15.95 (2 × GF104)
	4Cb	2.88	5.28 (GT215)	17.22 (2 × GF104)
	8C	4.60	5.16 (GT215)	19.47 (2 × GF104)

was 24.76 (in the case of the configuration using 8C with two GF104 units), whereas the speedup by the CPU cores alone (8C) was 4.46. For 1XJR, we observed a similar trend: using CPU cores alone (8C) gave a 4.60 times speedup, whereas utilizing the GPUs together with the CPUs produced a 19.47 times speedup over the reference case. Table 4 summarizes the speedup of the various configurations with respect to the 1C case.

Additionally, we measured how the running time of pFARNA varies depending on the input size. Obviously, feeding a longer sequence would take more time for structure prediction. We tested pFARNA with the 32 different input sequences and measured the running time of each of the 52 configurations. Fig. 8 shows the two fastest (8C+GF104 and 8C+GT200) configurations, along with the slowest (1C) and an intermediate configuration (8C). For each input sequence on each architecture configuration, we ran pFARNA 1000 times and recorded the average running time, as plotted in Fig. 8. For sequences with less than 20 nucleotides, the differences among the different configurations were not noticeable. As the size of the input sequence increases, however, the performance gap between the fastest and slowest configurations becomes more salient. Once again, we confirmed the effectiveness of using GPUs on top of multi-core CPUs for additional acceleration over the whole range of RNA sequence lengths.

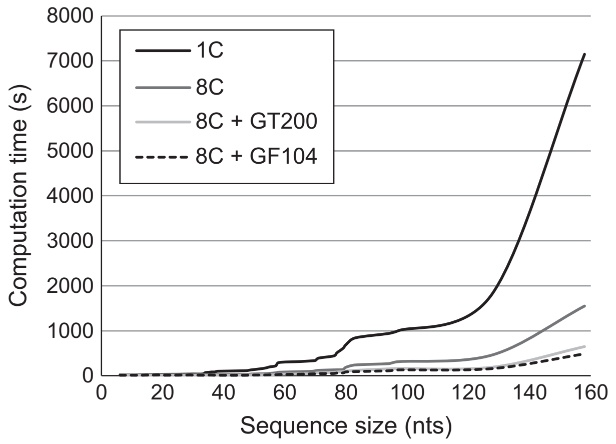

Fig. 8. Average running time versus input sequence length on different configurations. We measured how the running time of pFARNA varies depending on the input size. This plot shows the average running time versus input sequence length on the two fastest (8C+GF104 and 8C+GT200) configurations, along with the slowest (1C) and an intermediate configuration (8C). Evidently, as the length of the input sequence grows, the performance gap between the baseline and the CPU/GPU hybrid case widens.

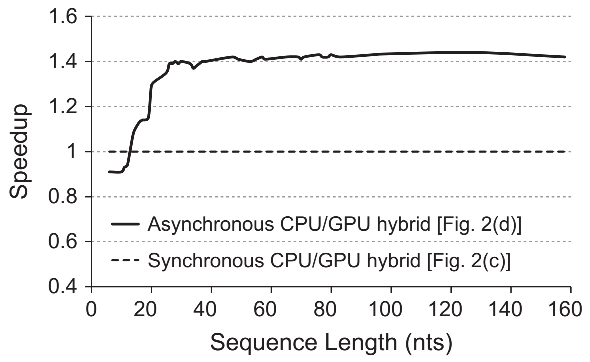

Fig. 9. Running time comparison of synchronous and asynchronous CPU/GPU hybrid execution flows. This plot compares the running time of the two models taken to predict the tertiary structure of the 32 RNA sequences of various lengths (configuration: 4Cb+GF104). For short sequences (less than 20 nucleotides), the synchronous model wins due to the overhead of the asynchronous model. However, this overhead pays off as the length of the input sequences increases, and the asynchronous hybrid shows better results.

4.4. Speedup by asynchronous execution flow

To see the effect of using the asynchronous hybrid execution flow shown in Fig. 2(d), we compared it with the synchronous model depicted in Fig. 2(c) in terms of two aspects.

First, Fig. 9 compares the running time of the two models taken to predict the tertiary structure of the 32 RNA sequences of various lengths. The architecture configuration used for this experiment was 4Cb with GF104 (a single GPU). For each sequence, the time for structure prediction was measured 1000 times, and the figure shows the average running time for each case. For very short RNA sequences that have about a dozen or fewer bases, using the synchronous model was faster. This is mainly due to the additional overhead incurred by using the proposed dynamic task scheduling. However, for longer sequences, using the asynchronous model produced better results, producing 32.9% faster running time on average.

Second, we compared the utilization of CPU cores and GPUs by each execution flow under comparison, as shown in Fig. 10. We predicted the tertiary structure of the 1XJR sequence (47 nucleotides) using the two different CPU/GPU execution flows and measured the utilization of CPU cores and GPU for each model. We used 4Cb with GF104 configuration. In the plot, grey areas indicate that a CPU or GPU core is being utilized and white areas represent idle states. Note that all white areas are not visible due to limited image resolution. For the synchronous model, the average utilization of the four CPU cores was 62.96% and the utilization of GPU was 51.32%. The CPU core that serves as the host for the GPU is idle when the GPU is running, although this fact is not clearly visible in the plot due to the limitations in image resolution. By using the asynchronous model, we could improve the utilization of both CPU cores and GPU. Consequently, for the asynchronous model, the average utilization of CPU cores and the GPU utilization increased to 83.92% and 74.34%, respectively. Overall, using the asynchronous model gave over 33% gain in terms of CPU/GPU utilization.

4.5. Remarks

Recently, the use of cloud computing is increasing due to its cost-effectiveness and other advantages. Commercial services such as Amazon Web Services (http://aws.amazon.com) provide convenient and flexible parallelization platforms researchers can utilize when they need powerful computing capabilities. The FARNA simulation may also benefit from utilizing cloud computing. For example, we can distribute instances of the Monte Carlo cycles in the FARNA flow (see Fig. 1) over multiple computing nodes. Using thousands of cores is not uncommon these days, and

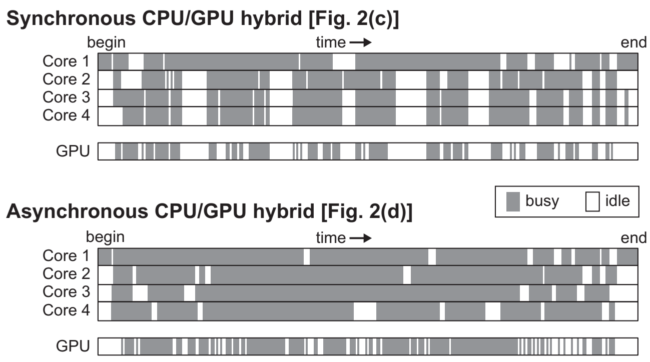

Fig. 10. Comparison of CPU/GPU utilization by synchronous and asynchronous execution flows. We compared the utilization of CPU cores and GPUs by each execution flow under comparison (configuration: 4Cb+GF104). By using the asynchronous model, the CPU utilization increased from 62.96% to 83.92% and the GPU utilization from 51.32% to 74.34%. The running time (from 'begin' to 'end' in the plot) was approximately 8.13 and 5.81 s for the upper and bottom plots, respectively.

we may reduce the overall running time of the simulation significantly. Of course, to maximize the utilization of massively parallel computing elements, we need to resolve dependencies among tasks and also minimize the frequency of inter-node communication. Otherwise, the use of cloud computing may fail to deliver the expected performance improvement.

Certain tasks in the FARNA flow fit well single-instruction multiple-data (SIMD) machines such as GPUs. Calculating pairwise energy between configurations is an example. Equipped with massively parallel light-weighted cores, GPUs can deliver an unprecedented level of computing power even at moderate price. In this paper, we used GPUs with 96–336 cores and 512–1024-MB memory to produce the presented results. Currently, the performance of GPUs is getting improved at the pace exceeding the Moore's law. By utilizing more powerful GPUs to come, the expected performance gain by employing the proposed method will increase, assuming that we can effectively handle the limitations of GPU computing such as difficulty in programming and limited memory.

5. Conclusion

We profiled the FARNA algorithm for predicting the structure of small RNAs and parallelized it using GPUs. We also describe our approach to using both multi-core CPUs and GPUs for maximum utilization of both types of computing cores. With respect to the baseline architecture that uses a single CPU core, using eight CPU cores along with two GPU cards resulted in a speedup of up to 24 times. Although we parallelized a specific algorithm to show the effectiveness of our CPU-GPU hybrid framework, nothing prevents it from being applied to other applications with similar assumptions and premises. Given that most PCs these days are equipped with a moderate GPU card, adopting our methodology will be very helpful for accelerating algorithms.

Conflict of interest statement

None declared.

Acknowledgements

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Education, Science and Technology) (No. 2011-0009963, No. 2012-R1A2A4A01008475, and No. 2010-0015504). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Author's contributions: Y.J. carried out the parallelization and drafted the manuscript. E.J. designed and conducted the analysis. H.M. analyzed the result and drafted the manuscript. E.C. and S.Y. participated in the design and analysis. S.Y. conceived and wrote the manuscript. All authors read and approved the final manuscript.

References

[1] E. Westhof, Toward atomic accuracy in RNA design, Nat. Methods 7 (4) (2010) 272–273.
[2] Ra'ed M. Al-Khatib, Rosni Abdullah, Nur'Aini A. Rashid, A survey of compute intensive algorithms for ribo nucleic acids structural detection, J. Comput. Sci. 5 (10) (2009) 680–689.
[3] Bruce A. Shapiro, Yaroslava G. Yingling, Wojciech Kasprzak, Eckart Bindewald, Bridging the gap in RNA structure prediction, Curr. Opin. Struct. Biol. 17 (April (2)) (2007) 157–165.
[4] C.E. Hajdin, F. Ding, N.V. Dokholyan, K.M. Weeks, On the significance of an RNA tertiary structure prediction, RNA 16 (7) (2010) 1340.
[5] Jes Frellsen, Ida Moltke, Martin Thiim, Kanti V. Mardia, Jesper Ferkinghoff-Borg, Thomas Hamelryck, A probabilistic model of RNA conformational space, PLoS Comput. Biol. 5 (June (6)) (2009) e1000406+.
[6] Rhiju Das, David Baker, Automated de novo prediction of native-like RNA tertiary structures, Proc. Natl. Acad. Sci. 104 (September (37)) (2007) 14664–14669.
[7] Rhiju Das, Madhuri Kudaravalli, Magdalena Jonikas, Alain Laederach, Robert Fong, Jason P. Schwans, David Baker, Joseph A. Piccirilli, Russ B. Altman, Daniel Herschlag, Structural inference of native and partially folded RNA by high-throughput contact mapping, Proc. Natl. Acad. Sci. 105 (March (11)) (2008) 4144–4149.
[8] Shantanu Sharma, Feng Ding, Nikolay V. Dokholyan, iFoldRNA: three-dimensional RNA structure prediction and folding, Bioinformatics 24 (September (17)) (2008) 1951–1952.
[9] Marc Parisien, Francois Major, The MC-Fold and MC-Sym pipeline infers RNA structure from sequence data, 452 (March (7183)) (2008) 51–55.
[10] Magdalena A. Jonikas, Randall J. Radmer, Alain Laederach, Rhiju Das, Samuel Pearlman, Daniel Herschlag, Russ B. Altman, Coarse-grained modeling of large RNA molecules with knowledge-based potentials and structural filters, RNA 15 (February (2)) (2009) 189–199.
[11] Andrew Leaver-Fay, Michael Tyka, Steven M. Lewis, Oliver F. Lange, James Thompson, Ron Jacak, Kristian Kaufman, P. Douglas Renfrew, Colin A. Smith, Will Sheffler, et al., Rosetta3: an object-oriented software suite for the simulation and design of macromolecules, Methods Enzymol. 487 (2011) 545–574.
[12] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, et al. The landscape of parallel computing research: a view from Berkeley, Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.
[13] G. Rizk, D. Lavenier, Gpu accelerated rna folding algorithm, Computational Science–ICCS 2009, 2009, pp. 1004–1013.
[14] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, K. Schulten, Accelerating molecular modeling applications with graphics processors, J. Comput. Chem. 28 (16) (2007) 2618–2640.
[15] David Baker, Andrej Sali, Protein structure prediction and structural genomics, Science 294 (October (5540)) (2001) 93–96.
[16] K. Binder, Monte-Carlo Methods, Wiley-VCH Verlag GmbH & Co. KGaA, 2006.
[17] L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1) (1998) 46–55.
[18] David R. Butenhof, Programming with POSIX Threads, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
[19] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, Wen-mei, Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '08, New York, NY, USA, 2008. ACM, pp. 73–82.
[20] David B. Kirk, Wen Mei, Programming Massively Parallel Processors: A Hands-on Approach, 1st edition, Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 2010.
[21] Stanimire Tomov, Rajib Nath, Jack Dongarra, Accelerating the reduction to upper Hessenberg, and bidiagonal forms through hybrid GPU-based computing, Parallel Comput. 36 (December (12)) (2010) 645–654.
[22] Wenfeng Shen, Daming Wei, Weimin Xu, Xin Zhu, Shizhong Yuan, Parallelized computation for computer simulation of electrocardiograms using personal computers with multi-core CPU and general-purpose GPU, Comput. Methods Programs Biomed. 100 (October (1)) (2010) 87–96.
[23] W.H. Press, B.P. Flannery, S.A. Teukolsky, W.T. Vetterling, et al., Numerical Recipes, vol. 547, Cambridge University Press, 1986.

[24] Helen M. Berman, John Westbrook, Zukang Feng, Lisa Iype, Bohdan Schneider, Christine Zardecki, The nucleic acid database, Acta Crystallogr. Sect. D 58 (June (6)) (2002) 889–898.

Yongkweon Jeon is a Ph.D. student at the Department of Electrical and Computer Engineering, Seoul National University, Seoul, Korea. His research interests include high-performance bioinformatics, GPU programming, and data mining.

Eesuk Jung received the M.S. degree in electrical engineering from Yonsei University, Seoul, Korea in 2012. His research area is RNA bioinformatics.

Hyeyoung Min received the B.S. and M.S. degrees in pharmacy from Seoul National University, Seoul, Korea, in 1996 and 1998, respectively, and the Ph.D. degree in cellular and molecular pathology from the University of California, Los Angeles, in 2004, and the M.S. degree in statistics from Stanford University, Stanford, CA, in 2008. She was a Postdoctoral Research Scientist at Stanford University. She is currently an Assistant Professor in the College of Pharmacy, Chung-Ang University, Seoul, Korea. Her current research interests include RNA bioinformatics and microRNA function in pathogenesis.

Eui-Young Chung received the B.S. and the M.S. degree in electronics and computer engineering from Korea University, Seoul, Korea. in 1988 and 1990, respectively, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 2002. From 1990 to 2005, he was a Principal Engineer with SoC R&D Center, Samsung Electronics, Yongin, Korea. He is currently a professor with the School of Electrical and Electronics Engineering, Yonsei University, Seoul, Korea. His research interests include bio-computing and VLSI design.

Sungroh Yoon received the B.S. degree in electrical engineering from Seoul National University, Korea, in 1996, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, CA, in 2002 and 2006, respectively. From 2006 to 2007, he was with Intel Corporation, Santa Clara, CA. Previously, he held research positions with Stanford University, CA, and Synopsys, Inc., Mountain View, CA. Dr. Yoon was an assistant professor with the School of Electrical Engineering, Korea University from 2007 to 2012. Currently, he is an assistant professor with the Department of Electrical and Computer Engineering and also with the Interdisciplinary Program in Bioinformatics, Seoul National University, Korea. His research interests include RNA bioinformatics, metagenomics, and high-performance bioinformatics.