Computers in Biology and Medicine 113 (2019) 103421

Contents lists available at ScienceDirect

Computers in Biology and Medicine

journal homepage: http://www.elsevier.com/locate/compbiomed

Fast single individual haplotyping method using GPGPU

Joong Chae Na [a], Inbok Lee [b], Je-Keun Rhee [c,*], Soo-Yong Shin [d,e,**]

[a] Department of Computer Science and Engineering, Sejong University, Seoul, 05006, South Korea
[b] Department of Software, Korea Aerospace University, Goyang, 10540, South Korea
[c] School of Systems Biomedical Science, Soongsil University, Seoul, 06978, South Korea
[d] Department of Digital Health, SAIHST, Sungkyunkwan University, Seoul, 06351, South Korea
[e] Big Data Research Center, Samsung Medical Center, Seoul, 06351, South Korea

ARTICLE INFO

Keywords:
Single individual haplotyping
PEATH/G
GPGPU
CUDA
Next generation sequencing

ABSTRACT

Background: Most bioinformatic tools for next generation sequencing (NGS) data are computationally intensive, requiring a large amount of computational power for processing and analysis. Here the utility of graphic processing units (GPUs) for NGS data computation is assessed.
Method: In a previous study, we developed a probabilistic evolutionary algorithm with toggling for haplotyping (PEATH) method based on the estimation of distribution algorithm and toggling heuristic. Here, we parallelized the PEATH method (PEATH/G) using general-purpose computing on GPU (GPGPU).
Results: The PEATH/G runs approximately 46.8 times and 25.4 times faster than PEATH on the NA12878 fosmid-sequencing dataset and the HuRef dataset, respectively, with an NVIDIA GeForce GTX 1660Ti. Moreover, the PEATH/G is approximately 13.3 times faster on the fosmid-sequencing dataset, even with an inexpensive conventional GPGPU (NVIDIA GeForce GTX 950).
Conclusions: PEATH/G can be a practical single individual haplotyping tool in terms of both its accuracy and speed. GPGPU can help reduce the running time of NGS analysis tools.

1. Introduction

Since bioinformatic tools typically impose a high computational burden, high-performance computing (HPC) remains one of the main challenges in the field of bioinformatics [1]. The traditional approach to solve this problem is to use multicore central processing units (CPUs) or cluster machines [2]. The advent of the multicore CPU or cluster systems has allowed many bioinformatic tools to run in parallel to process large-scale and high-throughput biological data.

More recently, graphics processing units (GPUs) have facilitated extremely high-performance computing at relatively low costs [3,4]. GPU usage for parallel and throughput intensive algorithms is referred to as general-purpose computing on GPUs (GPGPUs). The practical adaptation of GPGPUs has been facilitated using compute unified device architecture (CUDA) [5]. GPUs are useful in the investigation of diverse computationally intensive problems and numerous GPU-based software programs have been developed. GPGPUs have also been successfully applied to several bioinformatics problems [4]. In particular, over the past decade, advances in high-throughput technologies such as next generation sequencing (NGS) have facilitated a tremendous expansion of personal genomic sequence datasets. The utilization of GPGPU could help manage vast amounts of sequence data and performing genomic studies [6,7]. However, in genomic research based on NGS, the proposed applications have been mostly limited to sequence alignments and GPU-accelerated software for secondary analysis combined with machine learning.

Herein, we have developed software to address the single individual haplotyping (SIH) problem of NGS data by elaborating parallelized GPU programming. Previously, we proposed a novel method for SIH called probabilistic evolutionary algorithm with toggling for haplotyping (PEATH), based on the estimation of distribution algorithm (EDA) and successfully demonstrated its performance in terms of the phased length, N50 length, switch error rate, and minimum error correction compared to previous algorithms [8]. In addition, PEATH has the advantage of being able to solve SIH in the case of noisy datasets. However, EDA, which is the main optimization algorithm of PEATH, generally involves time-consuming processes. Thus, EDA has also been implemented with GPGPU [9,10]. In this study, we implemented PEATH with GPGPU

* Corresponding author. School of Systems Biomedical Science, Soongsil University, Seoul, 06978, South Korea
** Corresponding author. Department of Digital Health, SAIHST, Sungkyunkwan University, Seoul 06351 South Korea.
E-mail addresses: jkrhee@ssu.ac.kr (J.-K. Rhee), sy.shin@skku.edu (S.-Y. Shin).

https://doi.org/10.1016/j.compbiomed.2019.103421
Received 4 March 2019; Received in revised form 28 August 2019; Accepted 28 August 2019
Available online 29 August 2019

(PEATH/G) to reduce computational time. This is effectively a parallelized re-implementation of PEATH using CUDA.

2. Methods

2.1. PEATH

We provide a brief explanation of PEATH (Table 1) and refer the readers to Na *et al*, 2018 [8] for a detailed explanation.

Given two $n \times m$ matrices $\boldsymbol{M}$ and $\boldsymbol{Q}$, PEATH determines one haplotype expressed as $h = \{0, 1\}^m$, where m is the length of the haplotype. $\boldsymbol{M}$ is a matrix of n reads whose row denotes a read sequence and column denotes a position with a heterozygous variant. $M_{ij} \in \{\text{`0'}, \text{`1'}, \text{`}-\text{'}\}$ is an element in the i th row and j th column of $\boldsymbol{M}$. The value '0' means the sequence at the position is the same as the reference sequence, while '1' means the heterozygous variant occurs at the position. The symbol '–' indicates that the read does not cover the position. $\boldsymbol{Q}$ is a quality score matrix for $\boldsymbol{M}$, whose elements represent the probability of sequencing errors.

The PEATH consists of two stages. In the first stage, the PEATH determines the haplotypes using EDA, which is a probabilistic evolutionary algorithm. More specifically, we use a univariate marginal distribution algorithm (UMDA) to search for a good candidate haplotype. Each position of the candidate individual corresponds to a position on the input matrix $\boldsymbol{M}$, and the value for each position is {'0','1','–'}. The UDMA first randomly generates an initial population of L individuals where each individual is a candidate haplotype sequence of length m. For the next generation, it replaces the low ranked $L \times r$ parent individuals (r is the replacement ratio) by offspring generated using a probability of '1' at each position of the highly ranked $L \times (1 - r)$ individuals. After *gen_iter* generations, we choose the best individual as a solution of the UMDA. In UMDA, population size L, replacement ratio r, and the number of iterations *gen_iter*, are hyperparameters which can be chosen by the user beforehand.

In the second stage, to correct the switching errors in the solution of the UMDA, the PEATH adopts an exhaustive toggling approach. The range switch toggling (resp. the single switch toggling) identifies and corrects range switch errors (resp. single switch errors) by flipping bits in a range of positions (reps. a bit at a single position) in a candidate. The PEATH alternates the two toggling methods *tog_iter* times, where *tog_iter* is a constant.

In both stages, we use a fitness function for evaluation of candidate haplotypes. The fitness function of a haplotype h is the additive inverse of the sum of the quality-weighted errors, considering the characteristics of two haplotypes, h and its complement h^*. Specifically, the PEATH attempts to search for a haplotype with the minimal sum of the quality-weighted errors.

In real implementation, we can partition the matrix $\boldsymbol{M}$ into several segments called *phasing-units*, which are sets of overlapping sequence reads that do not overlap with any NGS sequence read in other units. Thus, we can perform the phasing for each unit independently. Moreover, for each phasing-unit, the PEATH performs the phasing several times to obtain more accurate haplotypes, since EDA is a non-deterministic algorithm using random numbers. In essence, the PEATH produces several haplotypes by repeating the phasing and chooses the best among them as a final solution.

2.2. PEATH/G: CUDA implementation of PEATH

CUDA is a parallel computing platform and programming toolkit developed by NVIDIA for creating high performance GPU-accelerated applications. Since CUDA Toolkit 1.0 was released in 2007, it has been rapidly developed and extended to support more operating systems, programming languages, and libraries over the past decade. In our

Table 1
Outline of the PEATH method. Adapted from Table 1 of Na et al. 2018 [8].

Procedure PEATH

input: input matrices $\boldsymbol{M}$ for sequence reads and $\boldsymbol{Q}$ for quality scores

output: haplotype sequences for phasing-units

for each phasing-unit **do**
  for $\boldsymbol{i} := \mathbf{1}$ to $\boldsymbol{ph\_iter}$ **do** // $\boldsymbol{ph\_iter}$ is a fixed user-defined parameter
    $\boldsymbol{h^i}$:= candidate haplotype obtained by EDA
    for $\boldsymbol{k} := \mathbf{1}$ to $\boldsymbol{tog\_iter}$ **do** // $\boldsymbol{tog\_iter}$ is a constant parameter
      $\boldsymbol{h^i}$:= haplotype improved from $\boldsymbol{h^i}$ by range switch toggling
      $\boldsymbol{h^i}$:= haplotype improved from $\boldsymbol{h^i}$ by single switch toggling
    end for
  end for
  select the best haplotype among $\boldsymbol{h^i}$'s
end for
end procedure

implementation, we used CUDA Toolkit 10.1, which is the latest version released in 2019.

Haplotype phasing of NGS data is suitable to parallelize because one can handle phasing-units independently. Moreover, the PEATH method iterates phasing for each phasing-unit and the iterations for a phasing-unit are independent of one another. We call an iteration for a phasing-unit a *phasing-instance*. If the number of phasing iterations is denoted by *ph_iter* and the number of phasing-units by *num_units*, the PEATH method has *ph_iter* × *num_units* phasing-instances, all of which are independent of each other and thus can be phased in parallel.

We implemented a parallel version (PEATH/G) of the PEATH method on a GPGPU using CUDA. Table 2 shows the outline of PEATH/

Table 2
The outline of PEATH/G.

```
__host__ Procedure PEATH/G                          // CUDA implementation of PEATH
    input: input matrices M for sequence reads and Q for quality scores,
    output: haplotype sequences for all phasing-units

    Build data structure for input matrices and copy them to the device
    Phasing_unit <<<num_units, ...>>> ()            // call kernel running in the device
    Copy haplotype sequences of all phasing-units to the host
end procedure

__global__ Procedure Phasing_unit    // kernel for a phasing-unit
    output: haplotype sequence for each phasing-unit
    size of grid: {num_units, 1, 1}

    if threadIdx.x = 0 then
        Phasing_instances <<< ph_iter, ...>>>() // call child kernel for phasing instances
    cudaDeviceSynchronize()                     // synchronize all thread-blocks in GPU
    Find the best among ph_iter sequences       // using multiple threads
end procedure

__global__ Procedure Phasing_instance    // kernel for a phasing-instance
    output: haplotype sequence for each instance of a phasing-unit

    size of grid: {ph_iter, 1, 1}

    h = EDA()                   // EDA stage
    for k := 1 to tog_iter do   // Toggling stage
        h := Range_SWT (h)      // Range switch toggling
        h := Single_SWT (h)     // Single switch toggling
    end for
end procedure
```

G. The host (CPU) first builds data structures for the input matrices and copies them to the device (GPU). Next, the device performs all phasing processes including the EDA and the toggling, and copies phased haplotype sequences into the host. There are no intermediate data to communicate between the host and the device during phasing. To process all phasing-units, the host calls the kernel Phasing_unit() with a grid of size *num_units*. Each thread-block in the grid is responsible for one phasing-unit. The kernel Phasing_unit() first launches the child kernel Phasing_instance() with a grid of size *ph_iter* (dynamic parallelism) where one thread-block is responsible for one phasing-instance. When completing the child kernel, we have *ph_iter* candidate sequences for the phasing-unit. The best among them is chosen as a solution using a parallel reduction technique with multiple threads, which is a well-known method in CUDA implementation. The cudaDeviceSynchronize() after the child kernel Phasing_instance() call guarantees that *ph_iter* candidate sequences for the phasing-unit are determined before finding the best sequence among them.

Table 3 shows a pseudocode of the device functions EDA() called in the kernel Phasing_instance(). The EDA stage maintains a population of L individuals, each of which represents a candidate haplotype sequence, and repeats several steps using multiple threads. Parallelization in the thread-block level is rather straightforward. Assuming that the thread-block size, i.e., the number of threads per thread-block, is 32, in the first step of initializing the population, thread t randomly generates data for positions $t,\ t+32,\ t+64,\ \ldots$ of all individual sequences. In the step for calculating the fitness values, thread t calculates the fitness values of individuals $t,\ t+32,\ t+64,\ \ldots$ of a population. In the step for sorting individuals, we use the sort function in the Thrust library for CUDA. In the step for generating the next generation, thread t is responsible for positions $t,\ t+32,\ t+64$, and so on. For each position, a thread calculates the probability of 1's at the position of populations and randomly generates data for the position based on the probability. Since these steps must be executed sequentially, they are fenced by a _syncthreads() barrier, which synchronizes all threads within a thread-block. To decrease running time, we store frequently accessed data such as the fitness values of individuals in shared memory.

Table 4 shows pseudocodes of Range_SWT() and Single_SWT(), which are the device functions executed alternatively in the toggling stage. In contrast with the EDA stage with L sequences, the toggling stage maintains one candidate sequence. First, consider the procedure Range_SWT(). Let tog_h^j be the haplotype sequence obtained by flipping bits of the current best solution *best_h* at positions between 1 and j.

Table 3
Pseudocode of EDA stage for PEATH/G.

```
__device__ Procedure EDA          // EDA for a phasing-instance
output: haplotype for a phasing-instance obtained by EDA

  Initialize the population (using multiple threads)
  __syncthreads()                 // synchronize all threads in a thread-block
  Calculate the fitness values of individuals (using multiple threads)
  __syncthreads()
  Sort individuals in order of fitness values (using multiple threads)
  __syncthreads()

  while gen_iter > 0 do           // gen_iter is a fixed parameter
    Generate the population of the next generation (using multiple threads)
    __syncthreads()
    Calculate the fitness values of individuals (using multiple threads)
    __syncthreads()
    Sort individuals in order of fitness values (using multiple threads)
    __syncthreads()
    gen_iter := gen_iter − 1
  end while
  return the best individual in the population
end procedure
```

Table 4
Pseudocode of toggling stage.

__device__ Procedure Range_SWT // Toggling for range switch

input: candidate haplotype $\boldsymbol{h}$ for a phasing-instance

output: haplotype improved by range switch toggling

do

$\boldsymbol{best\_h} := \boldsymbol{h}$ (using multiple threads)

__*syncthreads*() // synchronize all threads in a thread-block

Calculate the fitness value of $\boldsymbol{tog\_h^j}$ for every position $\boldsymbol{j}$

where of $\boldsymbol{tog\_h^j}$:= Flip bits of $\boldsymbol{best\_h}$ at positions between 1 and $\boldsymbol{j}$

(using multiple threads)

__*syncthreads*()

Select the best haplotype $\boldsymbol{h}$ among of $\boldsymbol{tog\_h^j}$'s (using multiple threads)

__*syncthreads*()

while $\boldsymbol{h}$ is better than $\boldsymbol{best\_h}$

return $\boldsymbol{best\_h}$

end procedure

__device__ Procedure Single_SWT // Toggling for single switch

input: candidate haplotype $\boldsymbol{h}$ for a phasing-instance

output: haplotype improved by single switch toggling

do

$\boldsymbol{best\_h} := \boldsymbol{h}$ (using multiple threads)

__*syncthreads*() // synchronize all threads in a thread-block

Calculate the fitness value of $\boldsymbol{tog\_h^j}$ for every position $\boldsymbol{j}$

where $\boldsymbol{tog\_h^j}$:= Flip bits of $\boldsymbol{best\_h}$ at position $\boldsymbol{j}$

(using multiple threads)

__*syncthreads*()

Select the best haplotype $\boldsymbol{h}$ among $\boldsymbol{tog\_h^j}$'s (using multiple threads)

__*syncthreads*()

while $\boldsymbol{h}$ is better than $\boldsymbol{best\_h}$

return $\boldsymbol{best\_h}$

end procedure

Thread t calculates the fitness values of tog_h^j for $j = t, t+32, t+64$, and so on. Due to the call _syncthreads(), all threads within the thread-block are synchronized until they finish calculating the fitness values of all tog_h^j's. Next, the best among all candidates is chosen using a parallel reduction algorithm. We repeat this until we cannot obtain a sequence with a better fitness value. Note that we maintain only the current best sequence *best_h* explicitly but not tog_h^j since we can easily obtain a bit value of tog_h^j from *best_h* when computing the fitness value of tog_h^j. We store *best_h* in shared memory because *best_h* is accessed by all threads in a thread-block. In addition, some temporary data shared in a thread-block, such as the fitness values, are also stored in shared memory. Procedure Single_SWT() is similar except for considering sequences flipped at a single position.

2.3. Datasets and experiments

We first measured the performance (time) using the NA12878 sequencing dataset produced by fosmid-based technology [11]. NA12878 has been widely used to measure performance in the SIH problem. Moreover, to further validate the performance of our method, the HuRef dataset was also used for testing [12]. All experiments were conducted on a workstation with Intel Core i9-9980XE CPU@3.0 GHz, 128 GB RAM, running Ubuntu Linux 64-bit operating system. The GPUs used were NVIDIA GeForce GTX 950 with 768 CUDA cores and 2 GB graphic RAM, NVIDIA GeForce GTX 1050 with 640 CUDA cores and 2 GB graphic RAM, and NVIDIA GeForce GTX 1660Ti with 1536 CUDA cores and 6 GB graphic RAM.

For fair comparison and analysis on the performance of various GPUs, we set parameters the same as those in the original PEATH [8]. The number of phasing iterations *phasing_iter* and number of toggling iterations *tog_iter* were set as 50 and 10, respectively. In the EDA stage, the population size L, replacement ratio r, and number of generations *gen_iter* were set as 100, 0.5, and 50, respectively. For PEATH/G, we used diverse thread-block sizes, 16, 32, 64, 128, 256, and 512. We denote the PEATH/G using a thread-block of size t by PEATH/G-t, such as PEATH/G-16.

3. Results

First, we performed experiments to determine the optimal thread-block size of PEATH/G with fosmid-based sequencing data. Fig. 1 compares the running time of diverse thread block sizes. The x-axis is the GPU device used in the experiments and the y-axis represents the running time of phasing for all 22 chromosomes. The running time is shown as an average value of 10 repetitions. PEATH/G-64 had the fastest running time with the 950 and 1050 devices at 385.6 s and 337.7 s, respectively. With the 1660Ti device, PEATH/G-32 and PEATH/G-64 showed similar running times. Thus, we chose the thread block size as 64 for further experiments. The 1660Ti device showed much faster running times with all block size settings compared to the other two GPU devices.

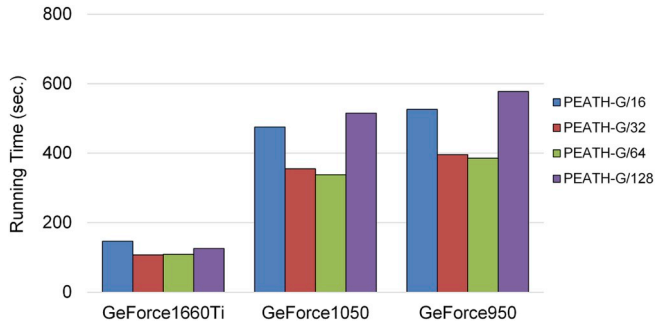

Fig. 1. Running time of diverse thread block sizes of CUDA for PEATH/G in three GPU devices.

To confirm the improvement in running time using GPGPU, PEATH (CPU-based sequential method) and PEATH/G-64 (GPU-based parallel method with a thread block size of 64) were compared. Table 5 shows the running times of the original PEATH algorithm and PEATH/G-64 for all 22 chromosomes.

As shown in Table 5, PEATH/G-64 clearly outperformed PEATH in all cases. PEATH/G was 13.25–46.84 times faster than PEATH. To further validate the usefulness of the CUDA-based implementation of PEATH/G, we ran the original PEATH algorithm in simple multi-thread environments. When we executed the original PEATH algorithm with 20 threads in parallel, a long running time (1730.895 s) was still required.

In this instance, we do not show the accuracy comparison results of PEATH/G and PEATH, since the basic algorithm was not changed. PEATH/G is another implementation of PEATH using CUDA, and we experimentally confirmed that there is no accuracy difference between those two implementations.

Moreover, to show that the CUDA implementation can also be useful for other datasets, we performed additional experiments using HuRef data [12]. In this dataset, PEATH/G-128 showed the fastest running time in all the GPU devices (Fig. 2). Table 6 compares the results for running time.

As expected, the GPGPU-based implementation was much faster than the original PEATH algorithm executed with only a CPU. The running time was reduced to 667.362 s with the 1660Ti GPU from 16941.53 s in the original CPU-based implementation.

4. Discussion

Managing and analyzing a vast number of datasets involves tremendous computational cost. With the development of high throughput technologies, it is becoming more critical to consider cost-effective analysis. Implementation of GPU processing instead of CPU processing could be a possible solution. For many computationally complex problems, GPGPU has been shown to dramatically reduce the overall execution time [13–15]. Herein, we implemented PEATH/G based on a probabilistic evolutionary algorithm using GPGPU. The evolutionary algorithms generally require an inordinately long time to identify an appropriate solution. Basically, most evolutionary algorithms involve evaluation of a large number of individuals; however, time-consuming fitness evaluation procedures can be performed in parallel. Several previous studies on the acceleration of evolutionary algorithms with a GPU have been performed [16,17]. We developed a GPU-based parallelization approach of the probabilistic evolutionary algorithm for SIH and showed that its implementation significantly reduced execution time.

There are additional possible solutions in terms of reducing computational costs. One example is the use of a hardware accelerator such as the field-programmable gate array (FPGA) [18–20]. Although this approach can reduce the computational costs, the FPGA implementation is relatively difficult. Another approach is grid computing, which can also facilitate the identification of solutions to very complex problems in a shorter time using distributed computing resources. However, it may incur notable challenges in terms of resource sharing because of non-defined standards and tools. Furthermore, several security and network connectivity issues exist. Thus, we improved our

Table 5
Running time of original PEATH and PEATH/G-64 for the fosmid dataset.

GPGPU	Running time (s)
CPU i9-9980XE	5107.92
CPU i9-9980XE (using 20 threads)	1730.895
GPU GeForce GTX 950	385.624
GPU GeForce GTX 1050	337.721
GPU GeForce GTX 1660Ti	109.282

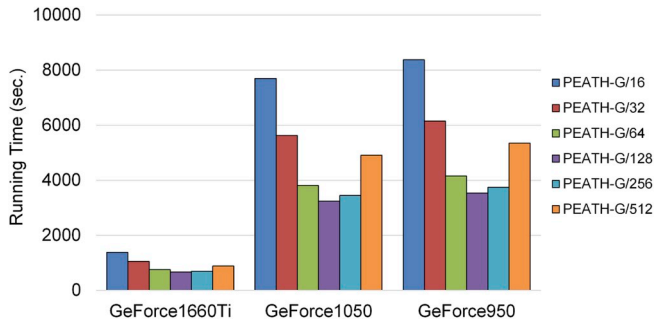

Fig. 2. Running time of diverse thread block sizes of CUDA for PEATH/G using the HuRef dataset.

Table 6
Running time of original PEATH and PEATH/G-128 for the HuRef dataset.

GPGPU	Running time (s)
CPU i9-9980XE	16941.53
GPU 950	3533.565
GPU 1050	3243.925
GPU 1660Ti	667.362

previous PEATH method using GPGPU. Moreover, another advantage of using GPGPU is that it does not always require a large-scale server system. As shown in our PEATH/G results, we can solve computationally complex problems efficiently.

Previously, there were some methods to solve the SIH problem on HPC platforms, such as PWHATSHAP [21] and GenHap [22,23]. PWHATSHAP is a parallel version of the WHATSHAP algorithm [24]. PWHATSHAP first decomposes chromosome datasets into independent sets of SNPs, which can be haplotyped independently. Next, it utilizes MapReduce framework for a general multi-core platform. GenHap adopts a genetic algorithm, which has a natural parallelism. GenHap was parallelized using MPI programming. However, to our knowledge, there were no methods applied to GPGPU to solve the SIH problem.

Most NGS data processing pipelines that handle raw sequencing data could be easily parallelized [25]. For example, in case of DNA-based sequencing data, the basic processing workflow comprises two steps: read alignments and variant detection. The processes can reduce the computational times by using both multi-threading and input data splitting and distribution. Thus, CUDA implementation should be considered for practical applications [26–28]. Moreover, many of the secondary analysis tools based on the NGS data could be converted to parallelized software, just as in the case of reconstructing PEATH into PEATH/G. For real clinical usage of SIH or for other NGS-based applications, a short turnaround time (the time required from sample [blood or tissue] acquisition to reporting of results) is critical [29]. NGS data analysis with GPGPU will reduce the running time significantly, increase our knowledge of very complex biological systems, and help realize precision medicine in the future.

In summary, we implemented a CUDA-based EDA algorithm called PEATH/G to solve an SIH problem. As expected, the PEATH/G produced results on the NA12878 and HuRef datasets much faster than the previous CPU-based EDA implementation. Therefore, CUDA implementation can help reduce the running time of time-consuming NGS analysis tools.

Availability

The source code is available at https://github.com/jcna99/PEATH-G.

Authors' contributions

JCN implemented PEATH/G and wrote the manuscript. IL implemented PEATH/G. JKR administrated the project and wrote the manuscript. SYS conceptualized the idea, administrated the project, and wrote the manuscript. All authors have read and approved the final manuscript.

Conflicts of interest

None declared.

Conflicts of interest

None declare.

Acknowledgements

This research was supported by the National Research Foundation of Korea (NRF) grant funded by the MSIT (Ministry of Science and ICT), Republic of Korea (grant no. NRF-2018R1C1B6005304), by Basic Science Research Program through the NRF funded by the Ministry of Education (2017R1D1A1B03029451), and by the MSIT, Korea, under the National Program for Excellence in SW supervised by the IITP (2015-0-00938 & 2017-0-00093).

Appendix A. Supplementary data

Supplementary data to this article can be found online at https://doi.org/10.1016/j.compbiomed.2019.103421.

References

[1] E.E. Schadt, M.D. Linderman, J. Sorenson, L. Lee, G.P. Nolan, Computational solutions to large-scale data management and analysis, Nat. Rev. Genet. 11 (2010) 647–657.
[2] Z. Yin, H. Lan, G. Tan, M. Lu, A.V. Vasilakos, W. Liu, Computing platforms for big biological data analytics: perspectives and challenges, Comput. Struct. Biotechnol. J. 15 (2017) 403–411.
[3] A.R. Brodtkorb, T.R. Hagen, M.L. Saetra, Graphics processing unit (GPU) programming strategies and trends in GPU computing, J. Parallel Distrib. Comput. 73 (2013) 4–13.
[4] M.S. Nobile, P. Cazzaniga, A. Tangherloni, D. Besozzi, Graphics processing units in bioinformatics, computational biology and systems biology, Briefings Bioinf. 18 (2017) 870–885.
[5] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with CUDA, Queue 6 (2008) 40–53.
[6] P. Klus, S. Lam, D. Lyberg, M.S. Cheung, G. Pullan, I. McFarlane, G. Yeo, B.Y. Lam, BarraCUDA - a fast short read sequence aligner using graphics processing units, BMC Res. Notes 5 (2012) 27.
[7] S. Warris, F. Yalcin, K.J. Jackson, J.P. Nap, Flexible, fast and accurate sequence alignment profiling on GPGPU with PaSWAS, PLoS One 10 (2015), e0122524.
[8] J.C. Na, J.C. Lee, J.K. Rhee, S.Y. Shin, PEATH: single-individual haplotyping by a probabilistic evolutionary algorithm with toggling, Bioinformatics 34 (2018) 1801–1807.
[9] S.M. Poulding, J.P. Staunton, N.J. Burles, Full Implementation of an Estimation of Distribution Algorithm on a GPU, GECCO 2011, GPUs for Genetic and Evolutionary Computation Competition, 2011.
[10] M. Essaid, L. Idoumghar, J. Lepagnot, M. Brévilliers, GPU parallelization strategies for metaheuristics: a survey, Int. J. Parallel, Emergent Distributed Syst. (2018) 1–26.
[11] J. Duitama, G.K. McEwen, T. Huebsch, S. Palczewski, S. Schulz, K. Verstrepen, E.-K. Suk, M.R. Hoehe, Fosmid-based whole genome haplotyping of a HapMap trio child: evaluation of Single Individual Haplotyping techniques, Nucleic Acids Res. 40 (2012) 2041–2053.
[12] S. Levy, G. Sutton, P.C. Ng, L. Feuk, A.L. Halpern, B.P. Walenz, N. Axelrod, J. Huang, E.F. Kirkness, G. Denisov, Y. Lin, J.R. MacDonald, A.W. Pang, M. Shago, T.B. Stockwell, A. Tsiamouri, V. Bafna, V. Bansal, S.A. Kravitz, D.A. Busam, K. Y. Beeson, T.C. McIntosh, K.A. Remington, J.F. Abril, J. Gill, J. Borman, Y. H. Rogers, M.E. Frazier, S.W. Scherer, R.L. Strausberg, J.C. Venter, The diploid genome sequence of an individual human, PLoS Biol. 5 (2007), e254.
[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, K. Skadron, A performance study of general-purpose applications on graphics processors using CUDA, J. Parallel Distrib. Comput. 68 (2008) 1370–1380.

[14] P. Harish, P.J. Narayanan, Accelerating large graph algorithms on the GPU using CUDA, in: Proceedings of the 14th International Conference on High Performance Computing, Springer-Verlag, Goa, India, 2007, pp. 197–208.
[15] Y. Tan, K. Ding, A survey on GPU-based implementation of swarm intelligence algorithms, IEEE Trans. Cybern. 46 (2016) 2028–2041.
[16] P. Pospichal, J. Jaros, J. Schwarz, Parallel Genetic Algorithm on the CUDA Architecture, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 442–451.
[17] T.V. Luong, N. Melab, E.G. Talbi, Parallel Hybrid Evolutionary Algorithms on GPU, IEEE Congress on Evolutionary Computation, 2010, pp. 1–8.
[18] I. Kuon, R. Tessier, J. Rose, FPGA architecture: survey and challenges, Found. Trends® Electron. Des. Autom. 2 (2008) 135–253.
[19] S.M. Qasim, S.A. Abbasi, B. Almashary, A Review of FPGA-Based Design Methodology and Optimization Techniques for Efficient Hardware Realization of Computation Intensive Algorithms, 2009 International Multimedia, Signal Processing and Communication Technologies, 2009, pp. 313–316.
[20] J. Kok, L.F. Gonzalez, N. Kelson, FPGA implementation of an evolutionary algorithm for autonomous unmanned aerial vehicle on-board path planning, IEEE Trans. Evol. Comput. 17 (2013) 272–281.
[21] A. Bracciali, M. Aldinucci, M. Patterson, T. Marschall, N. Pisanti, I. Merelli, M. Torquati, PWHATSHAP: efficient haplotyping for future generation sequencing, BMC Bioinf. 17 (2016) 342.
[22] A. Tangherloni, L. Rundo, S. Spolaor, M.S. Nobile, I. Merelli, D. Besozzi, G. Mauri, P. Cazzaniga, P. Liò, High Performance Computing for Haplotyping: Models and Platforms, Springer International Publishing, Cham, 2019, pp. 650–661.
[23] A. Tangherloni, S. Spolaor, L. Rundo, M.S. Nobile, P. Cazzaniga, G. Mauri, P. Lio, I. Merelli, D. Besozzi, GenHap: a novel computational method based on genetic algorithms for haplotype assembly, BMC Bioinf. 20 (2019) 172.
[24] M. Patterson, T. Marschall, N. Pisanti, L. van Iersel, L. Stougie, G.W. Klau, A. Schonhuth, WhatsHap: weighted haplotype Assembly for future-generation sequencing reads, J. Comput. Biol. 22 (2015) 498–509.
[25] N. Kathiresan, R. Temanni, H. Almabrazi, N. Syed, P.V. Jithesh, R. Al-Ali, Accelerating next generation sequencing data analysis with system level optimizations, Sci. Rep. 7 (2017) 9058.
[26] Y. Liu, B. Schmidt, D.L. Maskell, CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows–Wheeler transform, Bioinformatics 28 (2012) 1830–1837.
[27] A. Manconi, M. Moscatelli, G. Armano, M. Gnocchi, A. Orro, L. Milanesi, Removing duplicate reads using graphics processing units, BMC Bioinf. 17 (2016) 346.
[28] G. Peng, Y. Fan, W. Wang, FamSeq: a variant calling program for family-based sequencing data using graphics processing units, PLoS Comput. Biol. 10 (2014), e1003880.
[29] S. Roy, C. Coldren, A. Karunamurthy, N.S. Kip, E.W. Klee, S.E. Lincoln, A. Leon, M. Pullambhatla, R.L. Temple-Smolkin, K.V. Voelkerding, C. Wang, A.B. Carter, Standards and guidelines for validating next-generation sequencing bioinformatics pipelines: a joint recommendation of the association for molecular pathology and the college of American pathologists, J. Mol. Diagn. 20 (2018) 4–27.