IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 14, Issue 6 (Sep. - Oct. 2013), PP 43-52
www.iosrjournals.org

Enhance similarity searching algorithm with optimized fast population count method based on parallel design

SeyedVahid Dianat[1], Yasaman Eftekharypour[2], Nurul Hashimah Ahamed Hassain Malim[3] and Nur'Aini Abdul Rashid[4]
[1,2,3,4]*(School of Computer Science, Universiti Sains Malaysia, Malaysia)*

Abstract: *In chemoinformatics, drug discovery helps the chemists find new drugs with minimum side effects. The new drug should have similar behavior in terms of the chemical molecular formula to a known drug by applying similarity searching algorithms. A lot of previous works speedup the mentioned task by applying a predefined function, popc, which is only available on new GPUs. This research attempt to provide optimized design using different parallel hardware such as Multicore and GPU processors. Here the aim is to minimize hardware memory allocation by proposed data conversion method and improve data transmission speed. We achieved significant results in terms of performance and execution time in both CUDA and OPENMP designs of fast population count method with data conversion when compared to the sequential code.*
Keywords :*CUDA, graphics processing unit, multicore architecture, OPENMP, similarity searching algorithm*

I. INTRODUCTION

Chemoinformatics is an area that makes a link or bridge between chemical science and algorithms in computer science [4]. One of the researches in chemoinformatics is drug discovery. Its lifecycle can take many years and it can cost up to hundreds of million dollars [5]. The content of the databases now has exceeded 30 million chemical molecules.

A. Motivations

Chemoinformatics has become an emerging topic for computer science. High Performance Computing (HPC) contributed to chemoinformatics by improving the speedup of computation of the optimized methods on modern CPUs and GPUs. To reach this aim, researchers and even medical industries attempted to do research and study in this area[6].

Nowadays, computer science researchers attempt to apply optimized algorithm on many applications. This application could be listed as: biochemical, drug discovery, image processing and other applications with different architecture of parallel system such as CUDA platform (Compute Unified Device Architecture is based on shared memory systems and GPU programming). The CUDA platform is used to harness the power of the GPU [7].

B. Research contributions

The contributions of this research are:

- Overcome hardware dependency of *popc* algorithm in modern GPUs by applying optimized *fast population count* method with proposed data conversion strategy in existing GPUs.
- An accelerated sequential similarity searching algorithm with *fast population count* and data conversion methods on OpenMP and CUDA platforms.

II. RESEARCH BACKGROUNDS

A. General purpose computation on GPU

Usually GPU (Graphic Processor Unit) is used to process graphic computation part but nowadays is used as general purpose that is named GPGPU. In parallelization by using GPU, it is not applied alone even GPU is applied beside the CPU and data is transferred between them [8].

The use of graphic processing units (GPU) for general-purpose computation (GPGPU) has been as an active research field for many years which has begun on machines such as the Ikonas in 1978, the Pixel Machine, which is a parallel image computer [9], and Pixel-Planes 5 (graphics engine). The wide experimental research of GPUs in the last years has resulted to increase developments of the types of computation on recent GPUs [10].

B. Shared memory architecture

In shared memory systems, there is only one shared global memory between processors which is connected with bus-based or switch-based network. So tasks on different processors communicate with each other by writing to global memory and reading from it[11]. The illustration of shared memory architecture is depicted in Figure 1. In the following subsections, two platforms of OpenMP and CUDA platforms are explained as shared memory architecture.

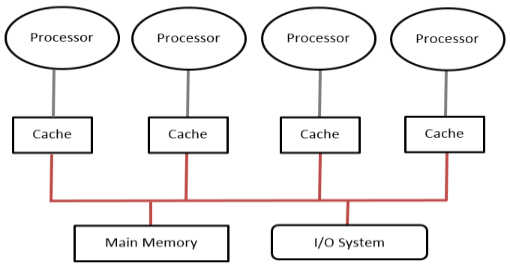


Figure 1. Shared memory architecture[2].

1) CUDA platform

In 2006, Compute Unified Device Architecture (CUDA) was introduced as a compiler which increases the performance of computing in parallelism concepts and made parallel programming easy. As a result, it makes many applications in science to compute and render the data. Also, it provides direct access to compute units. Even these features are improving with each version of CUDA APIs that are released for developers. CUDA is a GPGPU API platform for multithreaded Single Instruction Multiple Data (SIMD) model for GPGPU programmers. Sequential source code is executed on the single core of machine (host) as sequential code and parallel code is implemented as a group of kernel functions to be executed in an SIMD model which is arranged by a set of threads[12].

In CUDA, GPGPU functionality is defined by writing device functions for C language kernels. Only one kernel can be run at a time on the GPGPU. A thread is a sequence of instructions which has been defined in many papers and books. Groups of threads are called thread blocks and are executed on the Symmetric Multi-Processing (SMPs). The blocks are divided into Single Instruction Multiple Data (SIMD) groups of 32 threads called warps, which are divided into groups of 16 threads called half-warps[13]. A grid contains groups of blocks with many threads inside. The structure of CUDA architecture has been illustrated in Figure 2.

The memory hierarchy in CUDA platform is categorized into a set of registers (on-chip), local memory (residing in an off-chip DRAM) for each thread, a private shared memory for thread blocks, a global memory for all launched threads, read-only texture cache and constant memory. The CUDA platform offers three primary optimization strategies such as the Memory Optimization (MO), Execution Configuration Optimization (ECO) and Instruction Optimization (IO)[13]. At present CUDA is based on C/C++ language as its extension that supports only NVIDIA hardware but OPENCL supports both ATI and NVIDIA hardware [14].

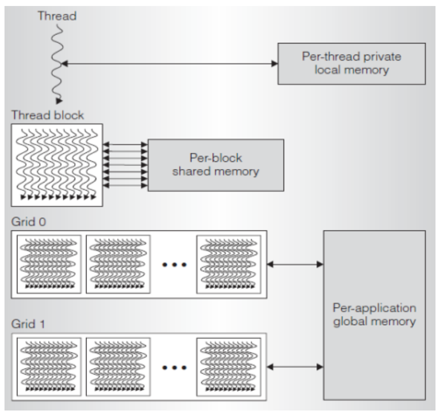

Thread
Per-thread private local memory
Thread block
Per-block shared memory
Grid 0
Grid 1
Per-application global memory

Figure 2. The CUDA structure of threads, blocks and grids of blocks[3].

2) OPENMP platform

Generally, it is a portable shared memory threading API (Application Program Interface) that standardizes tasks and loops into the parallelism system. It has provided some OpenMP clauses for both logical and dynamic extension. Another key advantage is that it allows a developer to parallelize its applications incrementally. Since OpenMP is a directive based compiler for developers, it can compile serial and parallel code together inside a single source code easily [15].

C. Similarity searching algorithm

Similarity searching is one of the most important tasks in chemoinformatics. It tries to retrieve molecules data from a large database. This way is based on exploring the data of molecules which is searching in a big chemical database that needs thousands of comparisons to return high level of similarity degree between database compounds and a query compound [16].

III. PREVIOUS WORKS

Chao Ma et.al (2011) applied predefined function of "popc( )" in CUDA platform to calculate the number of one bits in the string of fingerprint in chemical compounds, but this function is not available for general GPUs in public servers or laptops. Its limitation is that it could be executed only on modern and a new version of GPUs [17].

Peter Bakkum and Kevin Skadron (2010) in their paper have focused on an acceleration way of SELECT queries and investigation of an efficient GPU implementation related to the SQLite command library. They have provided results of execution on NVIDIA Tesla family C1060. They achieved speed up around 20 to 70 times. They believed the results depended on the size of the dataset. They explained the exist limitations of GPU technology in that day. They mentioned limitations such as the GPU memory size and the transfer time of data memory transmission from the host (CPU) to the device (GPU). These two limitations are the most relevant technical limitations. Their project only supports for numeric data types[18].

Pu Liu et al. (2011) proposed a method to accelerate similarity searching algorithm on the chemical databases by using GPU. This method is called lossless feature count fingerprints and integer entropy coding. It has comparable results better than using fixed length 1024 bits fingerprint because of using lossless fingerprint compression algorithms. They claimed this method can increase the efficiency of storage. There is an obvious drawback of using this method that the compressed fingerprints on a large database need to be decompressed then they can be used. This process takes extra time to do decompress task. They invested around $500 to buy GPU NVIDIA GeForce GTX 580 model. This GPU card has 1536MB GDDR5 off chip DRAM Memory, 64KB constant memory, 16 multiprocessors and each of its processors have 32 CUDA cores with 1.54GHz speed. Also, it supports both single-precision and double-precision data computing. They could search and scan the entire PubChem database which contains around 32 million chemical compounds by using this GPU around 0.2 to 2 second on average. They achieved 2 orders of magnitude faster compare to conventional CPU. They illustrated if multiple query patterns are processed in a batch of queries, then the speedup become substantially around 0.02 to 0.2 seconds per query for 1000 queries. They have applied the Elias gamma compression algorithm with a compression rate of 0.097 [19].

As Liao et al. (2011) achieved around 73 to 143 times speedup on GPU based common 2D Tanimoto selection compound algorithms in comparing with running on conventional CPU. But, their work was not optimized enough [20].

Imran S.Haque et al. (2011) in their paper described an efficient method for population count on modern x86 processors, cache efficient matrix traversal and leader clustering algorithms reduce bottleneck problem because of memory bandwidth in clustering and similarity matrix construction. They combined two methods of fast population count primitive (which is a method to count the number of 1 bits in the vector) and architecture agnostic algorithms to reduce the traffic of memory. It enabled around 20 to 40 times speedup against traditional CPU methods and achieved 65% of performance in theoretical peak. Also, they demonstrated the performance of their methods based on two models of problems: first, similarity matrix construction and second, leader clustering [1].

IV. RESEARCH METHODOLOGY

One parts of similarity searching algorithm is Tanimoto similarity coefficient calculation. It has high level of time consumption in sequential design because this calculation should be done for a huge database with many fingerprints of chemical compounds. That's why the recent researches attempt to improve this part of algorithm based on parallelism concepts. We parallelized this part of the algorithm based on OPENMP and CUDA designs.

We have applied the optimized algorithm on MDL Drug Data Report (MDDR) database. This type of database has been used to apply single reference similarity searching algorithm for this work. The database file has two columns such as compound Ids and 1024 bit binary fingerprints (fixed length fingerprints). The type of fingerprints are ECFP4 (binary strings). We have duplicated the database row, which is number of compounds in database, with the different amount of compounds in the database such as 12817, 25635, 51270, 102540, 205080 and 410160 compounds in this work. At the following section, it presents the pre-computing stage which is same in both OPENMP and CUDA design.

A. Pre-computing with data conversion strategy

This step read the data from the database files, fetch and store the data in 1D arrays also data conversion process are done to keep less memory allocation. We used 1D arrays in this design because when 2D arrays had higher execution times in sequential design of drug discovery in experimental tests. This part of are done in CPU part with single core. The details of this step with applying data conversion are described in the following:

1) Read and store the compound Id and fingerprints of query and database files into the 1D respective arrays. In this step, 1024 binary fingerprints are 1024 chars (after fetching from file). This 1024 chars are divided into 128 chunks of 8 character. Finally 8 chars are converted to one unsigned integer so we will have 128 unsigned integer values for each compound. Next 128 unsigned integer values represent the second fingerprint and so on. So the 1D array contains 128*102540 for all 102540 compounds.
2) Initialize and allocate memory for 1D arrays (access to 1D memory is only one time but access to 2D memory are twice).

This strategy called by data conversion method that has some benefits to carry data in GPU design very fast. In addition, the execution process on the 2D arrays is a heavy task for GPU. That's why, the program stored compound fingerprints in 1D arrays to achieve high speed of transferring results for the CUDA design with the less memory allocation by data conversion strategy. It helps to manage memory. The pseudocode of data conversion strategy is described in Figure 3.

```
1.  For int=1 to the number of total row in data.txt total
2.      Read the fingerprints from Data.txt and Query.txt files each as a string of characters
3.      Convert each 8 character to an integer
4.      Store in arrays of integers
5.  End for
6.  For int=1 to the number of total row in query.txt
7.      Read the fingerprints from Query.txt files each as a string of characters
8.      Convert each 8 character to an integer
9.      store in arrays of integers
10. End for
```

Figure 3. Pseudocode of data conversion strategy.

B. Fast population count method with converted data structure

Each query and data fingerprints has 1024 binary bits that is used in this work. It has already decomposed and converted to 128 unsigned integer numbers that each unsigned integer described 8 binary bits. To calculate the number of one's bits, fast population count method [1] is applied on each 128 unsigned integer values in the program, and sums all 128 results to calculate all numbers of one's bits for all 128 unsigned integer values. This process is done for both respective compounds A and B, also it is done on the result of (compound A & compound B). The pseudocode of population count calculation for each unsigned integer value is described in Figure 4.

```
1.m1= 0x55;                          //constant value of 01010101
2.m2= 0x33;                          //constant value of 00110011
3.m4= 0x0f;                          //constant value of 00001111
4.//input=input number
5.input = (input & m1) + ((input  >> 1) & m1);
6.input = (input & m2) + ((input >> 2) & m2);
7.output = (input & m4) + ((input >> 4) & m4);
8.convert output variable to unsigned integer value;
9.//output is the result of number of one's bits in the first chunk and will be done for all chunks
```

Figure 4. Pseudocode of population count calculation.

C. OPENMP design

The sequential code could be parallelized with OpenMP platform by adding only OpenMP directives, so the new code is compiled on OpenMP compiler and the compiled file is executed by multicore system.In this design, we decided to keep the default of OpenMP about applying two threads, which are default number of threads with two cores. Also, the directives such as "#pragma omp parallel" and "#pragma omp for schedule (static)" should be added to achieve parallelization. We apply fast population count method to evaluate its ability on CUDA design against sequential design in terms of increasing datasize. The details of each function have been explained in sequential part of previous sections. In OpenMP design, it makes the sequential code parallelized based on OpenMP platform and data parallelism.

D. CUDA design with data conversion strategy

There is a pop count function, known as predefined CUDA function, in CUDA APIs to count the number of 1 bits in a binary string. It could be easy for developers to implement this part of Tanimoto calculation with high speed programming and good performance of computation. It is only available on the new versions of GPU cards with special hardware specific which is not available on any portable laptop. This type of GPU cards could be found in special research laboratories, big drug discovery research groups and may be available in rich companies. On the other hand, the fast population count method is a method that does not have this type of hardware dependency with special GPU cards. By applying this method, it is possible to achieve good results in similarity searching algorithm in terms of good performance and saving time without buying expensive GPU cards and without paying a lot of money.

This method is based on counting the number of one's bits in query fingerprints and fingerprints inside the database also it calculates the number of one's bits in the results of "&" operation between each query fingerprint and each fingerprint of the database. Each query and database compounds have two same parts which are compound Id and 1024 bits (128 unsigned integer values) fingerprint. In the other hand, we have 10 query compounds and 102540 compounds in the database. At the first, this method counts the number of one's bits for fingerprint of the first query compound and the first compound in the database to calculate Tanimoto similarity coefficient between each query compound and database compounds. All fingerprints have fixed length of 1024 bit binary numbers to count the number of one's bits inside of the fingerprint. The steps are described with more details below:

1) 1024 bits fingerprint has been converted to 128 unsigned integer values for each fingerprint. Each unsigned integer value is presented as eight binary bits in the code.
2) Parallel reduction population count calculation is applied on 8 bits from left to right side of fingerprint in Figure 5. Its structure has showed in Figure 4 as Pseudocode.
3) After calculating the number of one's bits in the first 8 bits (first unsigned integer value) then the result is stored.
4) The steps 2 and 3 are repeated until the index reach to the last chunk of 1024 bit binary fingerprints (128 unsigned integer values) which means the number of one's bits for each chunk are calculated.
5) To count and store for all query fingerprints, step 4 are executed in parallel. It means one thread attempt on each unsigned integer value to count the number of one's bits, so the program applies 128 threads for each query fingerprint since the number of query fingerprints is ten. 128 multiply by 10 is equal to 1280 threads. That means it needs minimum 1280 threads. In the other hand, we defined 512 threads per block so the number of blocks should be rounded to 3.
6) Next step is to calculate the sum of all 128 unsigned integer values for each query fingerprint. Also, this part is done in parallel with 512 threads per blocks. It returns number of one's bits in each query fingerprint.
7) Then the program runs step 4 to count and store for all database fingerprints in parallel. It means one thread attempt on each unsigned integer value (128 unsigned integer value for each fingerprint) to count the number of one's bits, so the programs applies 128 threads for each database fingerprint. Since the number of database fingerprints are based on datasize (102540 compound) so 128 multiply by datasize is equal to the number of threads that need to do this task. We defined 512 threads per block, so the number of blocks are calculated by (datasize*128/threads)+1.
8) The next step is the sum of all 128 unsigned integer values for each database fingerprint. Also, this part is done in parallel with 512 threads per block. As we defined 512 threads per block, so the number of blocks is calculated by (datasize/threads)+1. Then it returns the number of one's bits in each database fingerprint.
9) In this step, the number of one's bits in the "&" operation result of each query fingerprint and all database fingerprints are calculated same calculation of the number of one's bit in each query fingerprint. The parallel part for 10 query is done ten times and the number of threads per block has been defined 512 threads. Also, the number of blocks is same as step 8 in this step.

Since we have different number of compounds in the database to evaluate the results, so we have different datasize for each database. That's why, we defined number of blocks dynamically. At this stage, all three parameters for the Tanimoto similarity coefficient formula are available. Its formula are mentioned in Eq.(1).

$$Tanimoto_{A,B} = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \quad (1)$$

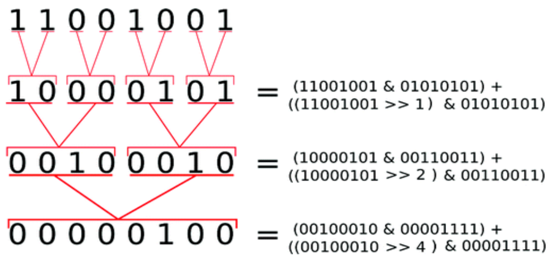


Figure 5. Parallel reduction population count[1].

E. CUDA design without data conversion strategy

In this design, fast population count method is applied without using data conversion to show their differences. Here, each character (its ASCII code minus 48) of fingerprint is considered as an integer value. Each fingerprint will be 1024 integer values. The other steps are same as previous designs where data conversion strategy has been applied in CUDA design with fast population count method. So the amount of (1024*4*datasize bytes) allocated for database fingerprints after fetching the database text file without data conversion process and amount of (128*4*datasize bytes) with the data conversion process.

F. Popc sequential algorithm

Each fingerprint has 1024 bits of zero and one bits that are applied in this work. We have already converted the fingerprint to 128 unsigned integer value which each one has 8 binary bits. The popc algorithm is run for each chunk (each unsigned integer value) then it sums each chunk of 128 results with each other. This function will be done on results of logical operation for (compound A & compound B) to calculate the number of one's bits for compound A&B. The pseudo code of this algorithm based on its semantic code[21] are described in Figure 6.

```
1)  Repeat until (input value!= 0){ //check input value isn't zero
2)      if (input value &0x1==1){//bitwise and with hexadecimal of 1
3)              count++;          //plus one because it has found one bit
4)      }
5)      shift one bit to right;   //check next bit
6)  }
```

Figure 6. Pseudocode of popc algorithm in sequential.

G. CUDA design with popc method

In this design, popc method has been applied instead of fast population count method and only this part is replaced with the popc method. Popc is one of the predefined APIs of CUDA platform that known as integer arithmetic instruction in CUDA. These types of functions have some advantages such as fast computation, high performance, easy to develop and even in the less time in comparing with implementing the method manually. Regular implementation of the algorithm by developer does not have that type of mentioned hardware dependency [21]. CUDA design with popc method should be compiled with CUDA Capability of version 2.0 and above. So there is hardware dependency because it needs to buy modern and new expensive model of GPUs that support this feature.

V. Evaluation

Results of execution times show the ability and optimization feature of the CUDA design of the *fast population count* method with data conversion strategy. The mentioned CUDA design of this work is executed on my laptop (GeForce GT325M) and second device (GeForce GTX260) of the server that both device do not support CUDA capability of higher than version 2.0 to use fast predefined functions of CUDA.

We illustrated the CUDA design with *fast population count* method on the second device in the server provides very near results of performance in comparing with the results of the CUDA design with predefined *popc* method on the first device (Tesla C2050) in the server. The GPGPU University server machine supports

CUDA capability version that both methods need. The CUDA design with *popc* method needs the minimum CUDA capability of version 2.0, but CUDA design with *fast population count* method needs the minimum version 1.0. It shows that the proposed design can achieve good performance and acceptable execution times without using *popc* method. So we can do drug discovery based on similarity searching with the single reference by applying CUDA design with *fast population count* method. That's why, we do not have to buy modern GPUs to achieve good results. The CUDA design with *fast population count* method reduces hardware dependency without losing the time by using data conversion strategy.

Another issue is that when the database is growing every day in many fields and the memory allocations in the programs is increasing so in this work was attempted to reduce hardware dependency in terms of memory usage in memory allocation. This is achieved by using data conversion, reduce memory allocations which is one type of memory managements. It helps to reduce hardware dependency that forces users to buy new GPUs and new hardware devices with more capacity of memory. We defined execution times as a running time of each design to solve the problem. The performance gained percentages are calculated by Eq.(2) and Eq.(3).

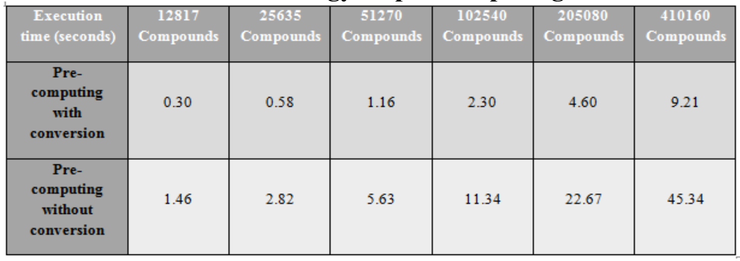
$$PG_{cuda} = \frac{T_{seq} - T_{cuda}}{T_{seq}} * 100 \qquad (2)$$

$$PG_{OpenMP} = \frac{T_{seq} - T_{omp}}{T_{seq}} * 100 \qquad (3)$$

According to the Table 4.1, the execution times go down when we use pre-computing with data conversion. The data conversion method chunks 1024 char to 8 char then the program converts each 8 char to one integer value, and this process is done 128 times for each 1024 binary character fingerprints. The program without data conversion converts 1024 binary character fingerprints to 1024 integer values that it is done 1024 times for each fingerprint. The ability of data conversion strategy is illustrated in pre-computing step by Table 1.

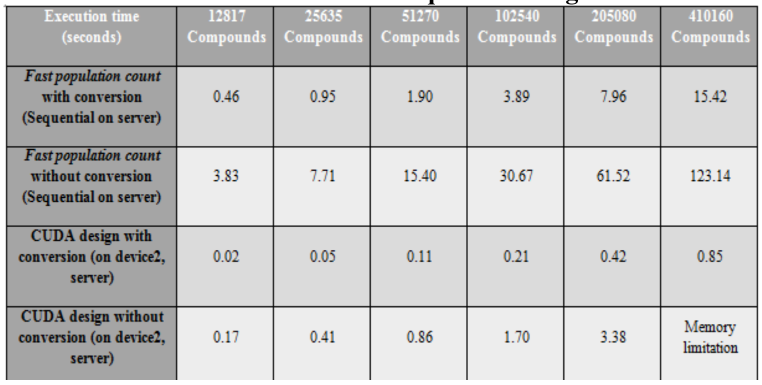
Table 2 shows execution times of *fast population count* design after pre-computing step with and without the data conversion strategy on sequential code and CUDA. According to Table 2, execution times of both designs with data conversion are faster than same design without data conversion. Because the program has to do the process for each integer value. On the other hand, the program has fewer tasks when we have applied the data conversion strategy in *fast population count* design because the fingerprint length reduced with data conversion. As a result, the design with conversion have been optimized as it is faster than the design without data conversion strategy. It is obviously clear by increasing the datasize. The data conversion makes the memory traffic reduction in data transmission between host and device, also it has less memory allocation.

Table 1. Effect of data conversion strategy on pre-computing in terms of execution time



Execution time (seconds)	12817 Compounds	25635 Compounds	51270 Compounds	102540 Compounds	205080 Compounds	410160 Compounds
Pre-computing with conversion	0.30	0.58	1.16	2.30	4.60	9.21
Pre-computing without conversion	1.46	2.82	5.63	11.34	22.67	45.34

Table 2. Effect of data conversion on CUDA and sequential design in terms of execution time



Execution time (seconds)	12817 Compounds	25635 Compounds	51270 Compounds	102540 Compounds	205080 Compounds	410160 Compounds
Fast population count with conversion (Sequential on server)	0.46	0.95	1.90	3.89	7.96	15.42
Fast population count without conversion (Sequential on server)	3.83	7.71	15.40	30.67	61.52	123.14
CUDA design with conversion (on device2, server)	0.02	0.05	0.11	0.21	0.42	0.85
CUDA design without conversion (on device2, server)	0.17	0.41	0.86	1.70	3.38	Memory limitation

Figure 7 shows the speedup of each platform which is calculated by dividing sequential execution time to parallel execution time. The Figure 7 illustrated the speedup of CUDA design with data conversion strategy has best speedup in compare with other mentioned design. The reason is the fast computation and applying many GPU cores in CUDA design. There is no result for 410160 compounds when CUDA design is applied without conversion strategy because of GPU memory limitation weakness.

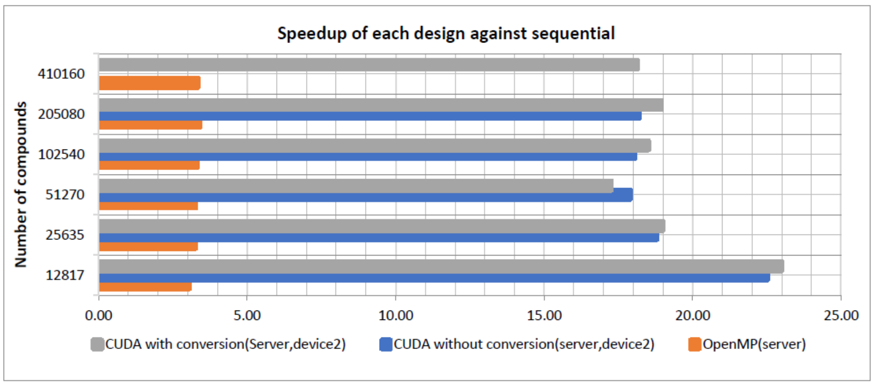
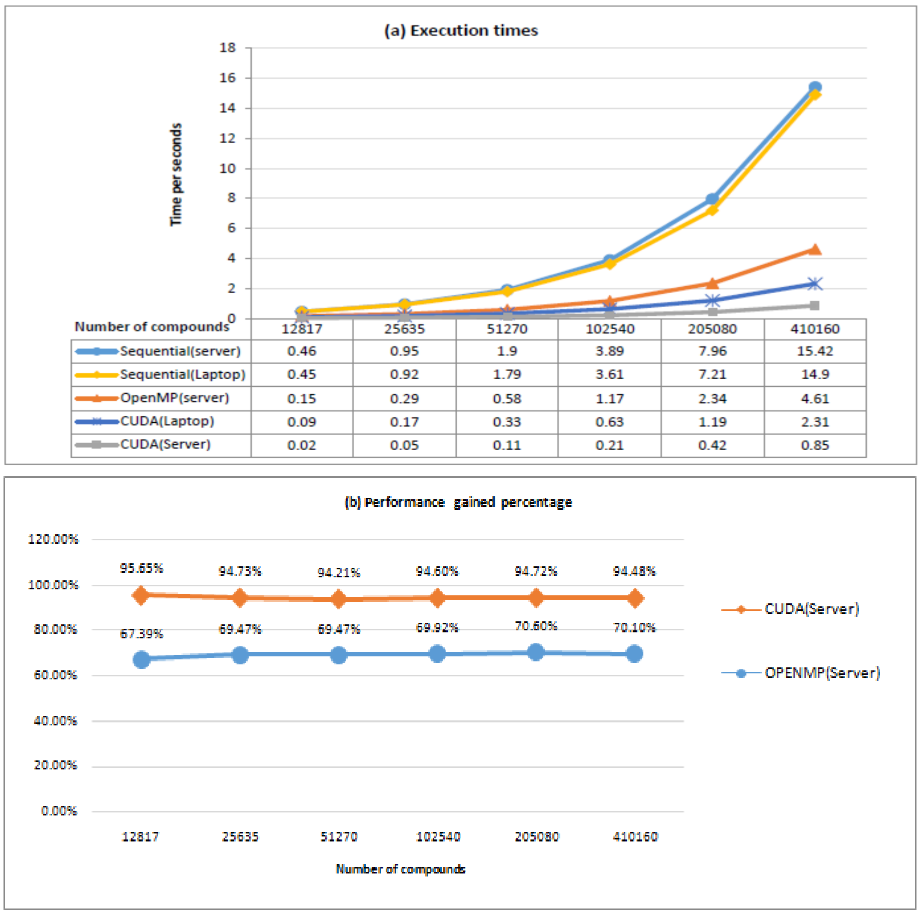


Figure 7. Speedup of each design against sequential.

According to the Figure 8, execution time on sequential design is the highest result because it is executed on single core system. About OPENMP design, it has better results in compare with the sequential design because it is executed based on multicore system. We executed OPENMP design with two threads and cores by default. In CUDA design, we executed it on the first device of server and device of laptop. We applied many cores of GPU in this design that's why it has very better results in terms of performance and execution times against sequential even OPENMP design. The results of CUDA design on server is better results in compare with its execution on laptop. The reason is that the memory and speed of the cores in device of server is higher than my laptop device feature.



Number of compounds	12817	25635	51270	102540	205080	410160
Sequential(server)	0.46	0.95	1.9	3.89	7.96	15.42
Sequential(Laptop)	0.45	0.92	1.79	3.61	7.21	14.9
OpenMP(server)	0.15	0.29	0.58	1.17	2.34	4.61
CUDA(Laptop)	0.09	0.17	0.33	0.63	1.19	2.31
CUDA(Server)	0.02	0.05	0.11	0.21	0.42	0.85

Figure 8. Execution times and performance gained percentages of fast population count implementation.

Table 3 shows the execution times of both methods of fast population counts and popc has very fast computation even in sequential design. The execution times when we use predefined popc function has better results in compare with applying fast population count method. We executed CUDA design with popc method on the first device of USM GPGPU server and we executed another design on the second device. Because popc function is only available in GPU cards that have CUDA capability version of above 2.0. It needs to be mentioned that although first device has faster cores against second device but both design has very near execution time and performance gained percentage results. The Figure 9 shows the power of both design with more than 94% performance against sequential design by increasing the datasize.

Table 3. Execution times of fastpopcount and popc methods with their sequential code.

Execution time (seconds)	12817 Compounds	25635 Compounds	51270 Compounds	102540 Compounds	205080 Compounds	410160 Compounds
POPC **(Sequential on server)**	0.39	0.77	1.54	3.11	6.32	12.43
Fast population count **(Sequential on server)**	0.46	0.95	1.90	3.89	7.96	15.42
POPC **(CUDA on server, device1)**	0.01	0.03	0.05	0.11	0.22	0.43
Fast population count **(CUDA on server, device2)**	0.02	0.05	0.11	0.21	0.42	0.85

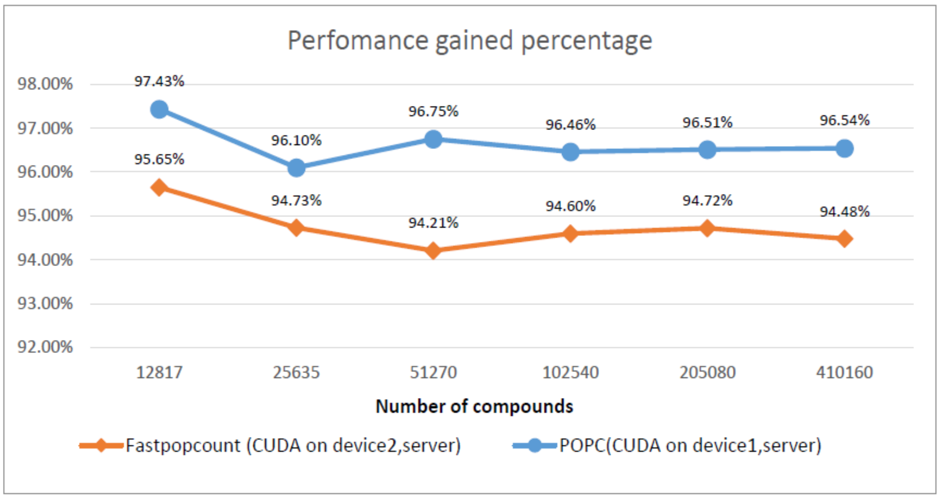


Figure 9. Performance percentages gain for each method on different devices.

VI. Conclusion

The aims of this work are to achieve an optimized similarity searching algorithm on different platforms such as OpenMP and CUDA platforms. The algorithm minimizes hardware dependency by applying the parallel design based on the fast population count method with data conversion. The fast population count method with data conversion strategy is used in similarity searching algorithm instead of applying predefined intrinsic popc function which needs special and expensive modern GPU. We have illustrated that the CUDA design with optimized fast population count method with data conversion could be run on general laptops that have general family of NVIDIA GPU device. My laptop and second device in the USM GPGPU server machine is a possible hardware to implement the algorithm and it produces very near results for execution times in comparison with applying the predefined intrinsic popc function.

According to the trend of OpenMP version of the parallel algorithm, the OpenMP design has good results in compared with the sequential code. However compared to the CUDA design it has longer execution times when we have a small amount of data. This work provided some features such as overcome the hardware dependency and memory reduction by data conversion strategy. As a result, users do not have to buy special and modern GPU device because the user can achieve good performance and execution times against sequential code.

Acknowledgements

The authors thank from Universiti Sains Malaysia (USM) that provided us GPGPU server to test and develop our implementation. This is the place to thanks from the anonymous reviewers for their professional comments about our work.

REFERENCES

[1] I. S. Haque, V. S. Pande, and W. P. Walters, "Anatomy of high-performance 2D similarity calculations," *Journal of chemical information and modeling,* vol. 51, pp. 2345-2351, 2011.

[2] S. Deshpande, P. Ravale, and S. Apte, "Cache Coherence in Centralized Shared Memory and Distributed Shared Memory Architectures," *International Journal on Computer Science and Engineering (IJCSE) Special Issue,* 2010.

[3] J. Nickolls and W. J. Dally, "The GPU Computing Era," *Micro, IEEE,* vol. 30, pp. 56-69, 2010.

[4] N. Brown, "Chemoinformatics—an introduction for computer scientists," *ACM Computing Surveys (CSUR),* vol. 41, p. 8, 2009.

[5] J. A. DiMasi, R. W. Hansen, and H. G. Grabowski, "The price of innovation: new estimates of drug development costs," *Journal of health economics,* vol. 22, pp. 151-186, 2003.

[6] J. Pitera, "Current developments in and importance of high-performance computing in drug discovery," *Current opinion in drug discovery & development,* vol. 12, p. 388, 2009.

[7] G. Rastelli, "Emerging Topics in Structure-Based Virtual Screening," *Pharmaceutical Research,* pp. 1-6, 2013.

[8] H. El-Rewini and M. Abd-El-Barr, *Advanced computer architecture and parallel processing* vol. 30: Wiley. com, 2005.

[9] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra, "Graphics processing unit (GPU) programming strategies and trends in GPU computing," *Journal of Parallel and Distributed Computing,* vol. 73, pp. 4-13, 2013.

[10] I. Rajani and G. N. Gopal, "Advanced Trends of Heterogeneous Computing with CPU-GPU Integration: Comparative Study."

[11] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran, "An approach to scalability study of shared memory parallel systems," *ACM SIGMETRICS Performance Evaluation Review,* vol. 22, pp. 171-180, 1994.

[12] Nvidia. (2012, NVIDIA's Next Generation CUDA Compute Architecture Kepler GK110. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[13] M. Misic, D. Durdevic, and M. Tomasevic, "Evolution and trends in GPU computing," in *MIPRO, 2012 Proceedings of the 35th International Convention*, 2012, pp. 289-294.

[14] J. Ghorpade, J. Parande, M. Kulkarni, and A. Bawaskar, "GPGPU Processing in CUDA Architecture," *arXiv preprint arXiv:1202.4347,* 2012.

[15] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming* vol. 10: The MIT Press, 2008.

[16] M. Maggioni, M. D. Santambrogio, and J. Liang, "GPU-accelerated Chemical Similarity Assessment for Large Scale Databases," *Procedia Computer Science,* vol. 4, pp. 2007-2016, // 2011.

[17] C. Ma, L. Wang, and X.-Q. Xie, "GPU Accelerated Chemical Similarity Calculation for Compound Library Comparison," *Journal of chemical information and modeling,* vol. 51, pp. 1521-1527, 2011.

[18] P. S. Bakkum, Kevin, "Accelerating SQL database operations on a GPU with CUDA: Extended results," *University of Virginia Department of Computer Science Technical Report CS-2010-08,* 2010.

[19] P. Liu, D. K. Agrafiotis, D. N. Rassokhin, and E. Yang, "Accelerating Chemical Database Searching Using Graphics Processing Units," *Journal of chemical information and modeling,* vol. 51, pp. 1807-1816, 2011.

[20] Q. Liao, J. Wang, and I. A. Watson, "Accelerating Two Algorithms for Large-Scale Compound Selection on GPUs," *Journal of chemical information and modeling,* vol. 51, pp. 1017-1024, 2011.

[21] Nvidia. (2012, Parallel thread execution. Available: http://docs.nvidia.com/cuda/pdf/ptx_isa_3.1.pdf