Massive Parallelism with GPU Computing for Similarity Searching in 3-D Chemical Database

Hai Ha Do (Student ID: 1495694)
Supervisor: Dr Dan Ghica

Submitted in conformity with the requirements
for the degree of MSc in Computer Science
School of School of Computer Science
University of Birmingham

Abstract

Massive Parallelism with GPU Computing for Similarity Searching in 3-D Chemical Database

Hai Ha Do (Student ID: 1495694)

Similarity search is one of the standard tools for drug discovery. It bases on a rationale that given a compound with desirable biological activities, structurally 'similar' compounds are likely to have a similar activity. Atom mapping algorithm measures the degree of similarity between pairs of three-dimensional (3D) chemical molecules represented by their inter-atomic distance matrices. A strong relationship was reported between the structural similarities resulting from the use of atom mapping method and the corresponding molecules' biological activities (Artymiuk et al. 1992). The algorithm was implemented in parallel with Distributed Array Processor (DAP) resulting in a search time of 1 hour for a typical input query in a database of 250,000 compounds. This thesis discusses a parallel implementation of atom mapping method using modern GPU computing techniques resulting in about 70 seconds to search for a typical query of a molecule containing 20 non-hydrogen atoms in a database of 300,000 compounds.

Declaration

The material contained within this thesis has not previously been submitted for a degree at the University of Birmingham or any other university. The research reported within this thesis has been conducted by the author unless indicated otherwise.

Signed .....

Acknowledgments

I would like to thank my supervisor Dr. Dan Ghica for his continuous guidance, advice and support of my summer project. I also would like to thank Dr. Hayo Thielecke for his informative lectures on C/C++ programming language. In addition, I would like to express my sincere gratitude to my family: my parents and my husband for their love and support during my MSc program.

Contents

Chapter 1

Introduction

The initial aim of this project is to implement a compute-intensive scientific algorithm in massive parallelism with CUDA C language and modern GPU architecture. Atom mapping algorithm for searching in a large database of 3D chemical structures was chosen.

In pharmaceutical and drug discovery research, lead compounds with desirable biological effects are compared with millions of other compounds in existing databases to find similar compounds with the hope of maximising its desirable biological effects. The few selected potential compounds would then be synthesised and tested. This process is called lead optimisation. The ability to search and rank similar compounds with relevant biological response is of great importance to lead optimisation. Several algorithms for similarity searching were reported (Stumpfe and Bajorath 2011). Among them is atom mapping algorithm (Pepperrell1 and Willett 1991) which is a robust quantitative approach to calculate structural similarity scores between a pair of molecules. Extensive testing by Artymiuk et al. (1992) had confirmed the strong relationship between structural similarities resulting from atom mapping method and the corresponding molecule's biological activities. The algorithm, however, is highly demanding of computer resources.

Parallel implementation of atom mapping method to speed-up the search time had been reported by Artymiuk et al. (1992) in distributed array processor(DAP). A single search of a typical compound requires about 60s for a database of 4500 compounds and from 20 to 60 minutes for a database of 180,000 compounds, depending on the size of the compounds (Willett 1995). In a typical cooperate database with a few millions compounds. a single search could take from several hours to even days. As chemical databases get larger over time with new compounds discovered and recorded, it is increasingly more important to shorten search time and therefore allow scientists to explore more hypothesis, accelerate the lead optimisation process and potentially cut costs of drug development.

This project aims to massively cut down search time by implementing atom mapping algorithm in parallelism efficiently in which computing intensive arithmetic calculations are performed in modern graphic processing unit (GPU). The prototype is written in C/C++

for CPU operations and CUDA/C for GPU operations. A desirable response time is a few minutes for a typical search in a large database for 300,000 compounds. A simple command prompt program would allow users to specify inputs and receive results.
This project is an example of the great potential of massively parallel applications in science and engineering fields with GPU computing.

Chapter 2

Background

2.1 Literature Review

Similarity search is an important and widely applied approach in chemical and pharmaceutical research to rationally select compounds with desired biological properties from databases. The search for new lead compounds bases on 'Similarity Property Principle' which intuitively states that similar (small) molecules should have similar biological properties (activity). Despite its simplicity, research efforts associated with this principle have profoundly changed and extended chemistry community (Stumpfe and Bajorath 2011).

There are two approaches to similarity search: descriptor-based methods and direct- comparison methods (Sheridan et al. 1996). In descriptor-based methods, sets of chemical descriptors generated based on the molecule's geometry and components are compared against each other for similarity search. Descriptor-based methods, however, cannot be used to obtain a superposition of the probe with the database entry, but they are very fast and thus more suitable for searches of large databases (Sheridan et al. 1996). In direct-comparison methods, structures of two molecules are superimposed to calculate similarity scores. These methods tend to be compute-intensive and thus are not common in database searches.

Atom mapping algorithm is a direct-comparison method that measures the degree of similarity between pairs of 3-D chemical structures represented by their inter-atomic distance matrices. It had demonstrated a strong relationship between top similar chemicals resulting from the method and the corresponding molecules' biological activities (Artymiuk et al. 1992). It is also particularly well-suited in typical situations when there is no information of what part(s) of the lead chemical are important.

2.1.1 Atom Mapping Method

Algorithm

Pepperrell1 et al. (1991) proposed an atom-mapping algorithm, a quantitative measure of the similarity between a pair of 3-D molecules, A and B, based on their inter-atomic distance matrices. The measure is calculated in two stages:

1. The geometric environment of each atom in one molecule, A, is compared with the corresponding environment of each atom in the other molecule, B, to determine the similarity between each possible pair of atoms.

2. The inter-atomic similarities resulting from Step 1 are used to identify pairs of geometrically-related atoms and these equivalences allow the calculation of the overall, inter-molecular similarity.

Calculation atom matching matrices matrices

The first stage compares each atom from A with each atom from B to identify the similarity between each pair of atoms.

For molecule A containing N(A) non-hydrogen atoms, the distance matrix for A, DA, is an N(A) x N(A) matrix such that for the ith atom in A, *A(I)*, the ith row of the distance matrix DA contains the distances from i to all of the other atoms in A.

This set of distances is compared with each of the rows, j, from the distance matrix DB to identify the matching distances, where a matching distance is within some user-defined tolerance, e.g., 0.5 Å.

Let there be C(i, j) such distance matches when A(i) is compared with B(j). The similarity, S(i,j), between A(I) and B(J) is given by the Tanimoto coefficient (Willett 1988):

$$S(i,j) = \frac{C(i,j)}{N(A) + N(B) - C(i,j)} \tag{2.1}$$

S(i,j) is the ijth element of an $N(A) \times N(B)$ matrix. This matrix, S, is called atom match matrix and contains the similarities between all pairs of atoms, A(i) and B(j), from molecule A and molecule B.

Calculation similarity score

Once the atom-match matrix has been created, the inter-atomic similarities contained within it are used to match every atoms from A and B that are very similar to each other. The basic matching algorithm is as follow:

1. Sort the elements of the atom-match matrix into order of decreasing similarity.

2. Scan the atom match matrix to find the remaining pair of atoms, one from A and one from B, that have the largest calculated value for S(i,j).

3. Store the resulting equivalences as a set of values of the form A(i) $\leftrightarrow$ B(j); S(i,j).

4. Remove A(i) and B(j) from further consideration.

5. Return to Step 2 if it is possible to map further atoms in A to atoms in B.

The overall degree of similarity between A and B is then calculated as the mean of the similarities over all of the atoms in A, using the information in the sets of values that are stored in Step 3. Thus, the inter-molecular similarity is given by:

$$Similarity = \frac{1}{N(A)} \sum_{i=1}^{N(A)} S(i,j) \tag{2.2}$$

Computation Requirements

Pepperrell1 et al. (1990) worked out computational complexity for the algorithm to be the order of O(N(A)^3) if N(A) $\approx$ N(B) with the computation being dominated by the calculation of atom match matrix.
In the first step, the comparison of the entire set of distances involves each atom in A with the entire set of distances involving each atom in B. The lists of distances associated with each individual atom are sorted into increasing order, hence the comparison of one atom with another requires order O(N(A) + N(B)) time. The comparison of all of the $N(A) \times N(B)$ pairs of atom thus requires order O(N(A) x N(B)(N(A) + N(B)) time (Pepperrell1 et al. 1990).
In the second step, the elements of the atom-match matrix is repeatedly scanned to find the next most similar pair of atoms. The scanning step is carded out N(A) times, once for each atom in A, and the matrix contains $N(A) \times N(B)$ elements; this stage requires order $O(N(A)^2 \times N(B))$ operations (Pepperrell1 et al. 1990).

Experimental data

Pepperrell1 and Willett (1991) carried out extensive testing for atom mapping method using ten small datasets from the medicinal chemistry literature that have been used in previous QSAR studies on the comparison of fragment weighting schemes for substructural analysis. These datasets cover a wide range of structural types, including both structurally homogeneous and structurally heterogeneous sets of molecules (Pepperrell1 and Willett 1991).

Validation of Method

Pepperrell1 et al. (1990) described a robust validation procedure to confirm that the rankings resulting from the use of the atom mapping method are, in fact, significantly better

than random rankings of the molecules in a dataset.
The basic approach is to create a frequency distribution of mean numbers of actives by randomly assigning activity labels. It is then possible to calculate whether the actual results for that dataset differ significantly from the mean of the random distribution of means, i.e. whether the actual result is significantly different. Pepperrell1 et al. (1990) listed out a detailed testing procedure for each dataset.

Limitations

Pepperrell1 and Willett (1991) stated three main limitations of atom mapping method:

1. The measures only compare molecules on structural basis. It does not take into account many other possible determinant factors e.g., lipophilicity or charge density.

2. The measures also take no account of the precise molecular mechanisms that result in the observed activity. For example, several different mechanisms are involved in some of the tested carcinogenicity datasets. Atom matching method is more suitable when limited knowledge about the mode of action is available.

3. Only a single conformation, specifically the low energy conformation, for each of the molecules is considered. Conformational flexibility may have some effects on 3-D substructures and thus its biological activities.

Parallel Implementation of Atom Mapping Method

Artymiuk et al. (1992) reported three parallel implementations of atom mapping method using Distributed Array Processr (DPA). DPA is an array processor operates on single instruction, multiple data principle. In other words, DPA with much less memory is a early similar version of modern GPU. Speed up, S, compares the performance of the most efficient sequential implementation with the massively-parallel implementation:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \tag{2.3}$$

The mean speed-up of parallel implementation of atom mapping method reported by Artymiuk et al. (1992) is only up to 39.6 times. With current GPU structure, the speed-up could potentially up to 100 times (Kirk 2013). A typical search of a file including 4500 structures for 50 nearest neighbours requires about 60 seconds (Artymiuk et al. 1992). A search in database of the 180,000 structures has a response time in the range 20-60 minutes with the precise time depending primarily upon the size of the target structure (Willett 1995).

2.2 Graphics processing unit (GPU)

A graphics processing unit (GPU) is a specialised parallel processor originally designed to rapidly manipulate and alter memory to accelerate the creation of images for smooth graphic display. The design philosophy is driven by strong demands of being able to perform massive floating-point calculation per video games by the video game industry. As of 2012, the ratio between peak floating-point calculation throughput between many-thread GPUs and multi-core CPU is about 10 (Kirk 2013, pp. 3).
Simulations in scientific and engineering applications often are compute-intensive with large number of arithmetic calculations. Such scientific program in traditional CPU sequential implementation is time consuming and thus a limiting factor. GPU with the ability to efficiently perform massive floating point calculation in parallel is particularly well-suited to these compute-intensive scientific and engineering applications.

2.2.1 Terminology

Compute Unified Device Architecture(CUDA) is a parallel computing platform and application programming interface (API) model created by NVIDIA (Kirk 2013). CUDA C is an extension of the popular C programming language with additional special keywords and API. Its simplicity allows programmers to efficiently work with GPU and take advantage of heterogeneous computing systems that contain both CPUs and massively parallel GPU's.
The primary difference between CPU and GPU lies on their hardware architectures (Figure

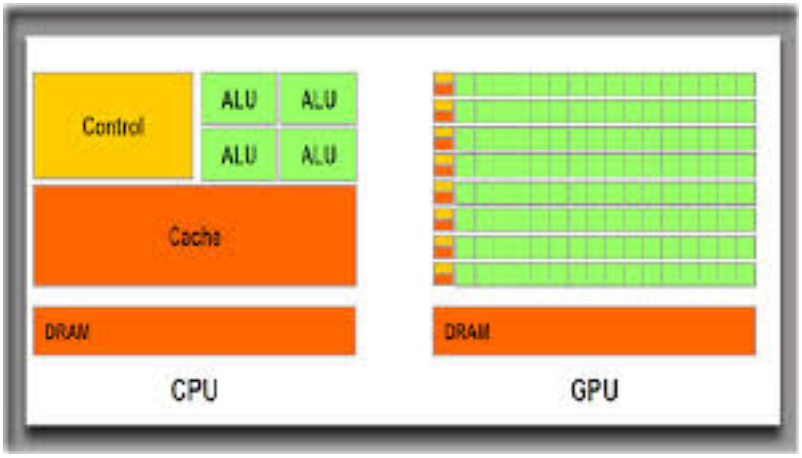


Figure 2.1: CPUs and GPUs have fundamentally different design philosophies

2.1). GPU contains far more ALU (Arithmetic Logical Unit) and fewer components for flow control and smaller cache memory than a typical CPU. Therefore, GPU is capable of very high arithmetic intensity over memory operations and thus parallel arithmetic operations.

CPUs, on the other hand, are designed to minimise the execution latency of a single thread with large cache memory. low-latency arithmetic units and sophisticated operand (Kirk 2013, pp. 4-6).

A CUDA device consists of one or more stream multiprocessors (SMP), each of which is composed of many stream processors (SM) (Figure 2.2). An SM is designed to execute all threads in a warp following the single instruction, multiple data (SIMD) logic. That is, at any time, one instruction is fetched and executed for all threads in a warp. As a result, all threads in a warp will always have the same execution timing.

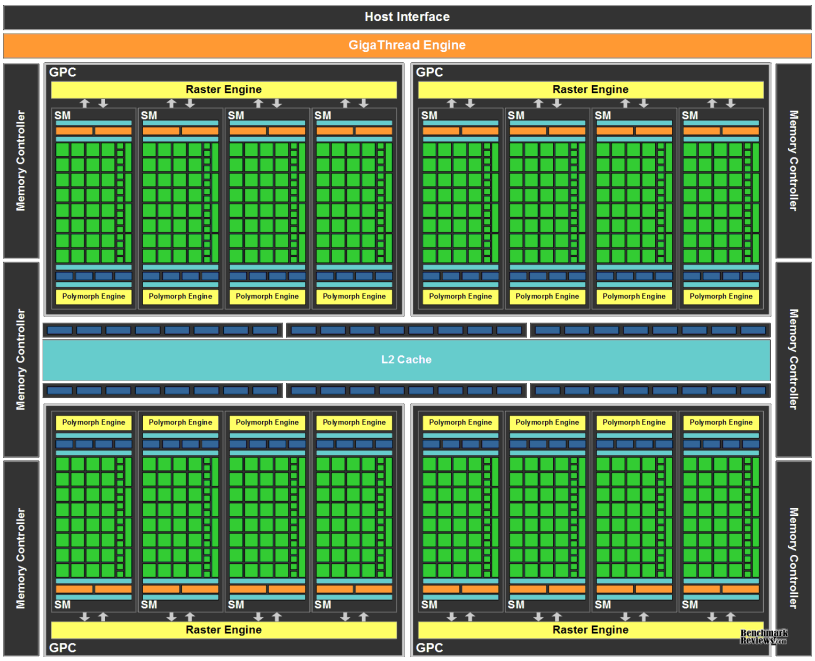


Figure 2.2: Architecture of the NVIDIA GTX580 GPU [NVIDIA]

GPUs have three main types of memory: global memory, local shared memory and private thread memory (Figure 2.3). Global memory, GPU's DRAM, serves as a fast cache to the motherboard RAM. It can be accessed by host (CPU) to transfer raw data to the GPU for processing and to store computation results prior to reading out back to motherboard RAM. The second memory type, local shared memory, is a much smaller but extremely fast bank of memory. Local shared memory can be accessed by all threads in GPU. Proper data allocation in local memory can significantly speed up computation. Lastly, private thread memory is an extremely small bank of memory that can be used within each thread for variables and temporary storage during the computation. In the Nvidia implementation, this uses registers for a certain amount, then starts using global memory when registers are exhausted.

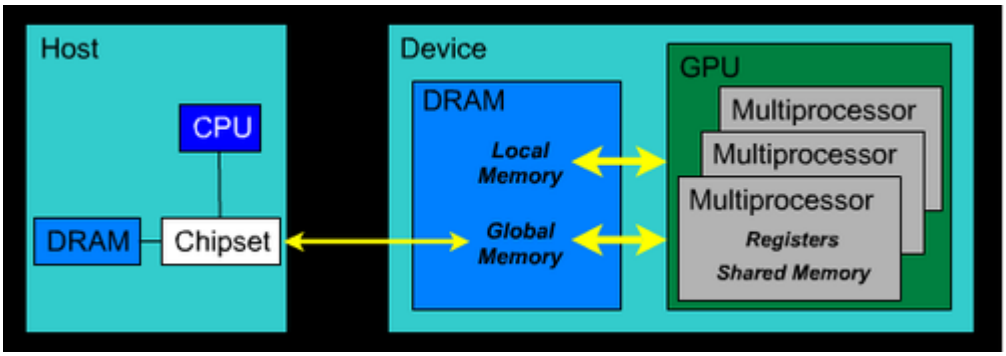


Figure 2.3: GPU memory architecture

The structure of a CUDA program reflects the coexistence of a host (CPU) and one or more devices (GPUs) in the computer. Each CUDA source file can have a mixture of both host (GPU) and device (CPU) code (Figure 2.4). The host code is straight C code which is compiled with the host's standard C/C++ compiler and is run on CPU. The device code, called kernel and marked with CUDA keywords. is compiled by runtime component of NVCC and is run on GPU (Kirk 2013, pp. 43). Each kernel is executed by a large number of threads which are organised into thread blocks of the same size. Users can specify and optimise the number of threads per block and the total number of blocks when launching the kernel. For more details, see the CUDA Programming guide in the CUDA Toolkit.

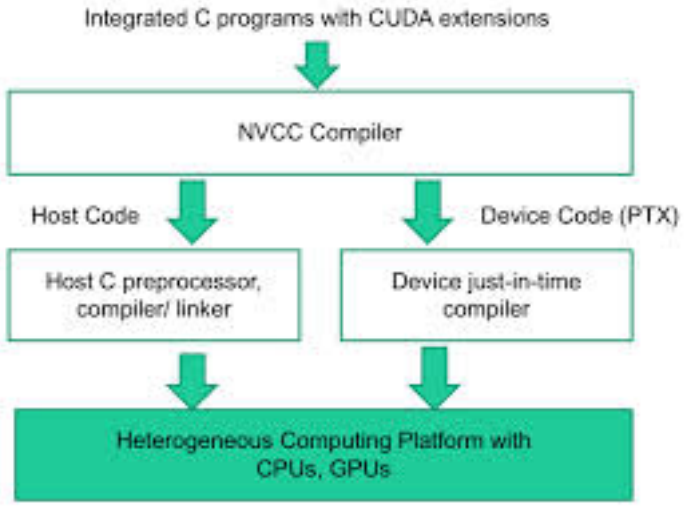


Figure 2.4: Overview of compilation process of a CUDA program

2.2.2 Main challenges

There are three main challenges when programming for the GPU:
Parallelism: GPU has a huge number of stream processors (SP). A typical professional GPU card could have up to several thousands of SP. Accessing registers and performing mathematical operations take several cycles, while accessing the GPU memory might take up to hundreds of cycles. Switching between different threads in GPU has zero latency. This strongly suggests that executing tens of thousands of threads at any given time is desirable for efficiency. Developing algorithms with such a level of massive parallelism is challenging and requires a paradigm shift from the conventional sequential programming style.
Divergence: The stream processors execute all threads in a single instruction, multiple data (SIMD) logic, i.e. they must execute the same instruction in the same cycle. Threads in each thread block are grouped into warps of 32 threads each. Threads in the same warp execute in step-wise manner. That is, the warp is split into two, each with some threads disabled when they need to take different paths. These warps are merged again as soon as the divergent path is completed. In the worst case when all 32 threads are on all different paths, their execution is effectively serialized. Implementation with less divergent in each kernel can have a significant effect on the performance.
Memory: Accessing memory system in GPU may not always be parallel. For every step, memory access of some threads in a warp can be combined in a single request if their memory locations are near each other. The memory system, in that case, can serve these threads in parallel. However, if these accesses cannot be combined, multiple requests are required, and these threads will be served sequentially and the performance is reduced. Combining that with the very high latency of the memory and the very tiny amount of cache available (typically less than 100KB per multiprocessor), accessing the memory becomes a serious bottleneck especially for applications with lots of memory access.
The three challenges mentioned above lead to some important design principles for GPU programming.
Firstly, the same computation performed by many threads on multiple pieces of data, data-parallel computation, is preferred. Therefore, input data should be as simple, uniform and large as possible. The work load of each thread should also be similar, since load balancing techniques such as work stealing or work donation might be costly.
Secondly, divergence should be avoided by minimising the use of if-else statements. Some simple checks are employed to break the set of jobs into several smaller groups with no conflicting paths. That effectively makes the algorithms lock-free, while still allows the algorithm to have very high parallelism.
Thirdly, the implementation should strive for locality across threads to achieve high utilization of the small cache. Memory accessed by a thread should also be local, if possible, not only for better caching efficiency but also for reducing the chance of conflicting with other threads.

Chapter 3

An example

3.1 Inputs and outputs

The program takes inputs of type 'mol' files. Each 'mol' file contains information about number of non-hydrogren atoms, number of bonds, 3-D coordinates of each atom and the list of bond connection between atoms in one molecule (Figure 3.1). The number of hydrogen could be easily calculated based on basic chemical rules. The output is a list of top chemicals having the highest similarity scores with the input query (Table 3.1).

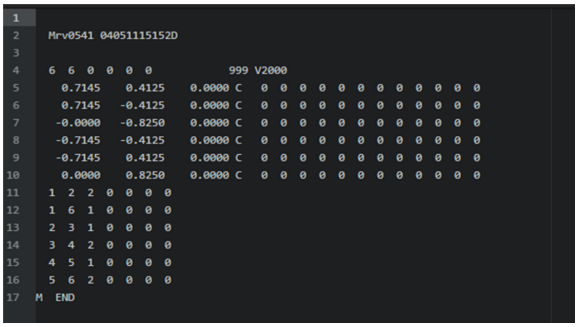


Figure 3.1: A mol file of benzene contains information about: (a) number of non-hydrogen atoms (6 atoms), (b) number of bonds (6 bonds), (c) 3-D coordinates of each atom (line 5-10) and (d) the list of bonds between connected atoms (line 11-16).

3.2 Calculation

An example of comparing toluene ((C_7H_7) against benzene (C_6H_6) is illustrated here with all major calculation steps.

Rank	Similarity score	Chemical Formula	Compound ID	Compound Name
1	1.0000	C18H12N4O3	CHEMBL6328	
2	0.9938	C18H16ClN3O3	CHEMBL265667	
3	0.9938	C16H8Cl3N3O3	CHEMBL268097	
4	0.9938	C17H11BrClN3O3	CHEMBL6352	
5	0.9216	C17H12ClN3O3	CHEMBL267864	

Table 3.1: An example of the result of a query molecule (CHEMBL6328) with database of 5 molecules

3.2.1 Calculate distance matrices

Distance matrix of a molecule with N atoms is a $N \times N$ matrix, storing the distance from one atom to another atom in that molecule. In the first step, the bond distances between each connecting atom is calculated using 3-D coordinates of each atom (Equation 3.1). Bond distances between an atom to itself and between non-connecting atoms are simply denoted as 0.

$$distance = \sqrt{(x-a)^2 + (y-b)^2 + (z-c)^2} \tag{3.1}$$

For example, a toluene molecule (Figure 3.2) has 7 carbons and 7 bonds. In toluene's distance matrix (Table 3.2), the bond distance between atom 1 and itself is 0. The bond distance between atom 1 and atom 2 is 0.8250 Å. Since there are no connections between atom 1 and atom 3 to atom 7, their bond distances are 0.

	1	2	3	4	5	6	7
1	0	0.825	0	0	0	0	0
2	0.825	0	0.825	0		0	0.825
3	0	0.825	0	0.825	0	0	0
4	0	0	0.8250	0	0.825	0	0
5	0	0	0	0.825	0	0.8250	0
6	0	0	0	0	0.825	0	0.825
7	0	0.825	0	0	0	0.825	0

Table 3.2: A Distance Matrix of Toluene

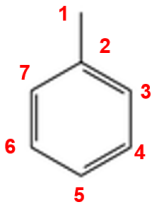


Figure 3.2: Toluene

A closer observation reveals that distance matrix is unchanged for any molecule. Saving calculated distance matrix into files would reduce unnecessary future calculation and save significant computation time. Distance matrices of equal sized molecules are written in the same file for the ease of organisation, code writing and visual inspection. The strategy also allows reading multiple molecules of the same size into a huge array which can be compared in massive parallelism.

3.2.2 Calculate atom mapping matrices

In the second step, every element in atom matching matrix is calculated using Equation 2.1. This example illustrates the comparison between toluene (Figure 3.2) and benzene (Figure 3.3). Comparing every atom in toluene's distance matrix (Table 3.2) to every atom in benzene's distance matrix (Table 3.3) with a specified threshold produces atom matching matrix (Table 3.4).

	1	2	3	4	5	6
1	0	0.8250	0	0	0	0.8250
2	0.8250	0	0.8250	0	0	0
3	0	0.8250	0	0.8250	0	0
4	0	0	0.8250	0	0.8250	0
5	0	0	0	0.8250	0	0.8250
6	0.8250	0	0	0	0.8250	0

Table 3.3: Distance Matrix of Benzene (C_6H_6)

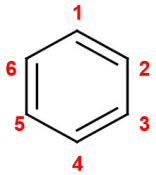


Figure 3.3: Benzene

For instance, comparing the first atom of toluene to the first atom of benzene produce 5 matched numbers (0, 0.8250, 0, 0 ,0), thus C(*1*,*1*) = 5. While toluene has 7 non-hydrogen atoms (N(A) = 7) and benzene has 6 non-hydrogen atoms (N(B) =6), the atom match score (S(1,1)) is 0.625 (Equation 3.2). The complete atom match matrix is shown in Table 3.4.

$$S(1,1) = \frac{C(1,1)}{N(A) + N(B) - C(1,1)} = \frac{5}{7+6-5} = 0.625 \tag{3.2}$$

	1	2	3	4	5	6
1	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250
2	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250
3	0.8571	0.8571	0.8571	0.8571	0.8571	0.8571
4	0.8571	0.8571	0.8571	0.8571	0.8571	0.8571
5	0.8571	0.8571	0.8571	0.8571	0.8571	0.8571
6	0.8571	0.8571	0.8571	0.8571	0.8571	0.8571
7	0.8571	0.8571	0.8571	0.8571	0.8571	0.8571

Table 3.4: Atom Matching Matrix of Toluene against Benzene

3.2.3 Calculate similarity scores

In this step, the atom match matrix is repeatedly sorted to find the maximum. That maximum is then accumulated in a value. All atom match scores of the corresponding atoms are also removed from future consideration by denoting those values to 0 which is

the minimal value of match scores. For example, scanning atom match matrix in Table 3.4 gives the first maximal value of 0.8571 at S(3,1). S(3,1) is the atom match score of atom 3 of toluene to atom 1 of benzene. After storing 0.8571 in an specific accumulated value (accSum), all elements in row 3 and column 1 are set to 0 as atom 3 of toluene and atom 1 of benzene are removed from further consideration. The resultant matrix is shown in Table 3.5. The whole process is repeated for another 6 times as the total number of atom in toluene is 7. The accSum value is divided by 7 to get the final similarity score using Equation 2.2

	1	2	3	4	5	6
1	0	0.6250	0.6250	0.6250	0.6250	0.6250
2	0	0.6250	0.6250	0.6250	0.6250	0.6250
3	0	0	0	0	0	0
4	0	0.8571	0.8571	0.8571	0.8571	0.8571
5	0	0.8571	0.8571	0.8571	0.8571	0.8571
6	0	0.8571	0.8571	0.8571	0.8571	0.8571
7	0	0.8571	0.8571	0.8571	0.8571	0.8571

Table 3.5: Atom Match Matrix of Toluene to Benzene after first iteration

3.2.4 Sort top k^{th} results

The similarity scores is sorted using standard 'sortrows' function in MATLAB. In C/C++ (CPU) implementation, sort function in Thurst library is used. A special sort_by_key function in CUDA Thurst library is used in CUDA GPU implementation.

Chapter 4

GPU Implementation

4.1 Calculate atom match matrices

The "calculate atom match matrices" kernel takes in the bond distance matrix of molecule A (float* a), the bond distance matrix of molecules B (float* b) and produces the atom match matrix (float* c). The number of atom in molecule A and B is NA and NB. respectively. There are NMax number of molecules type B in matrix float* b, i.e NMax is 1 if one molecule of A is compared with molecule of B. This implementation allows the comparison of one molecule of A to multiple (NMax) molecules of B in parallel provided that all B molecules have the same number of non-hydrogen atoms (NB). Global thread index (tid) is calculated based on its block index, its block dimension and its local thread index (Listing 1, line 8). Each thread will compare one atom of molecule A with another atom in a molecule B. Thus, comparison between a molecule A of NA atoms with a number (NMax) of molecules each containing NB atoms requires NA*NB*NMax number of threads to be executed in parallel (Listing 1, line 12). tempA and tempB is the index of the first elements of float* a and float* b that each thread is working on (Listing 1, line 15-16).
When comparing each element of float* a to every element of float* b, if a match is found, the count will be incremented. That particular element in array b should be removed from future comparison to avoid false positive match. An example of false positive match is the comparison float* a = {2,2} and float* b = {1,2} without removing a matched element from future comparison. Initially, the count result is 0. In the first round, a[0] is compared to b[0] and b[1]. Since a[0[] == b[1], the result is increased to 1. In the second round, a[1] is compared to b[0] and b[1]. Since a[1] == b[1], the result is incremented to 2. This result is wrong, because the b[1] is matched against both a[0] and a[1]. A correct solution is using a boolean array of NB value for each thread. Initially, all the value will be set to false, i.e. all B's element are not matched with any A's element (Listing 1, line 19). When looping through all elements of A and B, only if both the boolean value at B's element is false and B's element is equal to A's element in a specified threshold, then the count will

be incremented by 1 and the boolean value at that particular B's position will be changed to true (Listing 1, line 22-30). This mechanism ensures that each element of B will only be matched with one element of A.

This is an efficient algorithm that minimised most of divergence with only one if-statement (Listing 1). Boolean array is simply allocated in thread's local memory, thus is private for every thread. However, the size of boolean array has to be per-defined at compile time (Listing 1, line 1), making it unsuitable for comparing matrices with variable sizes. In addition, very small thread private memory implies that this function may fail if matrix B is too large.

In the second optimisation (Listing 2), the boolean array is dynamically allocated on a local memory runtime heap. This approach allows variable size of NB but is significantly slower (Table 4.1) and does not work if the number of molecule B (NMax) is too large because local memory is very small.

In the third optimisation (Listing 3), float* a and float* b are copied from global memory to shared memory based on the rationale that accessing shared memory is faster than global memory. The boolean array is dynamically allocated to allow NB to be defined at runtime. The result is, however, significantly slower and it does not work if NMax is too large (Table 4.1). The slow response time is due to the combined effects of copying data from global memory to shared memory and dynamically allocation of a boolean array for each thread.

In the fourth optmisation (Listing 4), bool* array is allocated in thread private memory while float* a and float* b are copied to shared memory. This results in significantly faster runtime of 1.3 ms on average (Table 4.1) which implies that memory allocation of bool* array is more important than that of float* a and float* b. A possible explanation is that for every iteration, bool* array is always checked first while float* a and float* b are checked only when the boolean array value is false. In addition, due to limited shared memory, the implementation will only work correctly for a small number of NMax.

In the fifth optmisation (Listing 5), bool* array is moved to shared memory while float* a an float* b stay in global memory. Shared memory is accessible to all threads, thus bool* array is now a large array combining all the individual small bool* array which is used for each thread in the previous implementation. This approach results in a slightly better runtime (1.161 ms) with variable NB but still suffers the limitation of small shared memory (Table 4.1).

Searching in big databases means the number of molecule to be compared (NMax) could get very large. The ability to correctly search with very large number of molecule is desirable. A normal GPU typically has much larger global memory compared to shared memory. This implies that bool* array should be in global memory to ensure correct result even for very large NMax. In the sixth implementation, bool* array serves as an input and stored in global memory (Listing 6). The approach allows very large number of molecule to be compared in parallel with a reasonable running time (Table 4.1).

Float* a containing distance matrix of only one molecule is used by every thread to compare

Name	Changes	Time (ms)	Large NMax	Flexible NB
1	Put the bool array into thread's local memory	1.068	No	No
2	Dynamically allocate an bool array for each thread	7.055	No	Yes
3	Load float* a and float* b into shared memory.Dynamically allocate an bool array for each thread	25.651	No	Yes
4	Load float* a and float* b into shared memory. Put the bool array into threads local memory	1.249	No	No
5	Put the bool array into shared memory	1.161	No	Yes
6	bool array is an input and stored in global memory	1.056	Yes	Yes
7	Put float* a into shared memory	1.017	Yes	Yes

Table 4.1: Summary of results for optimisation of function "Calculate Atom Match Matrix" The bench-marking tests involve molecule A of NA atoms with NMax number of molecules B of NB atoms. NA = 2, NB =3 and NMax = 1000.

with all molecules in float* b. In the final optimisation, copying float* a to shared memory (Listing 7) is strategic resulting better running time (Table 4.1).

4.2 Calculate similarity scores

The "calculating similarity scores" kernel takes in a float* a containing atom match matrices of NA*NB*NMax elements and produces a float* b containing similarity scores of NMax molecules (Listing 8). A total of NMax threads will work on NMax parts of array each containing NA*NB elements (Listing 8, line 7). The atom match scores are repeatedly sorted and maximum score is accumulated in 'total' value (Listing 8, line 18-25). The atoms relating to that maximum score are removed from further consideration by setting their other atom match scores to the minimum value of 0 (Listing 8, line 29-40). The final similarity score is the average of NA maximum atom match score.

In the first implementation (Listing 8), float* a and float* b are stored in global memory, thus the function will work well for even very large NMax. There is also a flexibility for size of inputs and the response time is reasonable.

In the second implementation (Listing 9), float* a is split up into smaller arrays and then is copied to local memory of each thread. Each smaller array contains an individual atom

Name	Changes	Time (ms)	Large NMax	Flexible NB
1	Modify values directly from the input stored in global memory	2.464	Yes	Yes
2	Copy array to local memory of each thread	3.709	No	No
3	Dynamically allocate an array for each thread	3.315	No	Yes

Table 4.2: Summary of results for optimisation of function "Calculate Similarity Score" Matrix" The bench-marking tests involve molecule A of NA atoms with NMax number of molecules B of NB atoms. NA = 2, NB =3 and NMax = 1000.

match matrix. The result time is, however, almost 1.5 times higher which could be due to the cost of copying from global to thread local memory. In addition, the function will not work flexible array size nor with very large NMax.
In the third implementation (Listing 10), float* a is also split up and dynamically allocated for each thread which allows the size of array to be defined at runtime. The result is, however, still slower than the first implementation and the function will fail with very large NMax.
In conclusion, the first implementation is the best choice with the fastest running time, flexible molecule size (NB) and very large NMax.

4.3 Sort top k^{th} scores

Sorting top k^{th} scores is a straight forward implementation with the function sort_by_key in CUDA Thurst library. Sort_by_key allow key-value sort with key is similarity score and value is the natural position of each score in the array. The sort results are recorded in a specific files. No further optimisation is done as this function is an optimised library function.

```
#define NC 3 /*In this case NB =3*/
__global__ void atomMatchingOpt1(float* c, const float *a,const float *b,
                const int NA, const int NB,const int NMax, float threshold){

float result =0;

//Calculate thread's index
int tid= blockIdx.x*blockDim.x+threadIdx.x;

/* To compare 1 molecule containg NA atoms and NMax molecules containing
NB atoms is NA*NB*NMax */
if (tid <NB*NA*NMax) {

/*First element in float *a and float*b for each thread to work on */
int tempA = (( tid / NB ) % NA ) * NA;
int tempB = ( ( tid / (NA*NB) ) *NB + (tid% NB) )*NB;

/*Local memory for each thread containg NB boolean value. (NC = NB)*/
bool array[NC] = {0};

/*Looping through all elements in float *a and float *b */
for (int k =0; k<NA; k++) {
  for (int t =0; t<NB; t++) {
    /* Check a b's element has been matched with an a's element.If yes,check
    whether the a's and b's elements are similar within a threshold */
    if (!array[(tempB+t)%NB] && (abs(a[tempA+k]-b[tempB+t])<= threshold)) {
    result = result  +1; /* increase the count */
    //Note that the b's element has been matched with an a's element
    array[(tempB+t)%NB] = true;
    break; /*stop the current loop*/
   }
  }
}
c[tid] = result;
}
```

Listing 1: First GPU implementation for calculating atom match matrices

```
//Dynamically allocate an bool array for each thread
bool* array = new bool [NB];
```

Listing 2: Snippet of code from the second GPU implementation for calculate atom match matrices

```
  //Allocation of float*a and float*b into shared memory
  //Note: Due to limited shared memory, kernel call will fall with too large NMax
  //C "Extern" keyword to allow size of array to be defined at run time
  extern __shared__ float aShare[];
  extern __shared__ float bShare[];

  //Copy data from global to shared memory
  if(tid <NA*NA)            aShare[tid]= a[tid];
  __syncthreads(); //Synchronise all threads

  if(tid <NB*NB*NMax)             bShare[tid]= b[tid];
  __syncthreads();  //Synchronise all threads

  //Dynamically allocate an bool array for each thread
  bool* array = new bool [NB];
```

Listing 3: Snippet of code from the third GPU implementation for calculate atom match matrices

```
  #define NC 3 /* NB = 3 for this example*/

  //Allocation of float*a and float*b into shared memory
  extern __shared__ float aShare[];
  extern __shared__ float bShare[];

  //Copy data from global to shared memory
  ...

  //Local memory for each thread containing NB boolean value. (NC = NB)
  bool array[NC] = {0};
```

Listing 4: Snippet of code from the fourth GPU implementation for calculate atom match matrices

```
  //Allocation of bool* array into shared memory
  //Note: Due to limited shared memory, kernel call will fall with too large NMax
  extern __shared__ bool array[];

  if(tid <NA*NB*NB*NMax)  array[tid]= false;
  __syncthreads();
```

Listing 5: Snippet of code from the fifth GPU implementation for calculate atom match matrices

```
 // bool * array is an input and stored in global memory
 __global__ void atomMatchingOpt6(float* c, const float *a, const float *b,
 bool * array, const int NA, const int NB, const int NMax, float threshold)
{
        ...
}
```

Listing 6: Snippet of code from the sixth GPU implementation for calculate atom match matrices

```
 // bool * array is an input and stored in global memory
__global__ void atomMatchingOpt7(float* c, const float *a, const float *b,
 bool * array, const int NA, const int NB, const int NMax, float threshold)
 {
  ...
  //Allocation of float*a into shared memory
  //Note: float*a reprents one single molecule, therefore takes a small memory
  extern __shared__ float aShare[];

  if(tid <NA*NA)          aShare[tid]= a[tid];
  __syncthreads();
  ...
}
```

Listing 7: Snippet of code from the seventh GPU implementation for calculate atom match matrices

```
__global__ void calculateSimilarity1(float* c, float *a, const int NA,
                const int NB, const int NMax){

int position,start;
int tid= blockIdx.x*blockDim.x+threadIdx.x;

if (tid<NMax) {
  //start is the first element every thread to check
  start = tid*NA*NB;
  // Initialised each thread's total to 0
  float total = 0;
  //loop through NA atoms of molecule A
  for (int k =0;k<NA; k++) {

 /* Step 1: Scan the atom match matrix to find the remaining pair of atoms,
    one from A and one from B, that has the largest value for S(i,j)*/
 // Find the max_element and its position in the float array of NA*NB element
 int position = 0;
 float max =  a[start];
 for (int t = 0; t<NA*NB; t++) {
   if ( a[start + t] > max) {
         max =  a[start + t];
         position=t;
   }
 }

 /* Step 2: Store the resulting equivalences as a tuple of the form
    [A(i) <-> B(j); S(i,j)] */
  total = total + max;// Sum the max into total
  // Get the position of max_element in 2D array
  int x = position/NB; //y axis a1
  int y = position%NB; // x axis b1

 /* Step 3: Remove A(i) and B(j) from further consideration        */
 // Set all the elements in the same row and column of max_element to 0
 // set all elements in the same y axis of max = 0
        for (int i =0; i<NB; i++ )  a[start + x*NB+i] =0;
 // set all elements in the same x axis of max = 0
        for (int j =0; j<NA; j++)  a[start + j*NB+y] =0;
 }
  c[tid] = total /NA; //The similarity score is total/NA
 }
}
```

Listing 8: First GPU implementation for calculating similarity scores

```
1  float temp[ARRAY_SIZE];
2  ...
3  // Each thread work on comparing 1 molecule of A to 1 molecule of B
4  // If we have NMax molecule B, we need NMax threads
5  if (tid < NMax) {
6    // Copy the appropriate part of a big array into a small one.
7    for (int q = 0; q<NA*NB; q++)    temp[q] = a[tid*NA*NB + q];
8    ...
9  }
```

Listing 9: Snippet of code in the second GPU implementation for calculating similarity scores

```
1  float* temp = new float[NA*NB];
2  ...
3  if (tid < NMax) {
4    for (int q = 0; q<NA*NB; q++)   temp[q] = a[tid*NA*NB + q];
5    ...
6  }
```

Listing 10: Snippet of code in the third GPU implementation for calculating similarity scores

```
1  void sort(float* keys, string* values, int size, int top, string output, string A) {
2
3   //initialized position from 0 to size-1 to use with sort_by_key function
4   //This position is used for future reference of each similarity score
5   int *position = new int [size];
6   for (int i= 0; i<size; i++)    position[i] = i;
7
8    //sort_by_key function in CUDA Thurst library
9    thrust::sort_by_key(keys, keys + size, position, thrust::greater<float>());
10
11   //Write the result to the "ouputFileName" with the name of "molecule2Compare"
12   writeTopSimilarityScore2File (keys, position, values, size, top,output, A);
13
14   delete [] position; //free memory the end
15 }
```

Listing 11: Sort function using sort_by_key function in CUDA Thurst library

Chapter 5

Bench-marking

5.1 Methodology

Running a GPU code includes two main steps: moving data from CPU to GPU and running GPU kernels. Moving data from CPU to GPU involves a significant amount of memory access in both GPU and CPU, therefore is a fixed and expensive cost. GPU kernel is the only part that could be optimised to improve speed up. A standard bench-marking methodology is to run the GPU kernel multiple times to amortise the expensive set-up cost. The result reflects the true potential speed-up that could be obtained by processing multiple inputs in batches (batch processing).

5.2 Results

5.2.1 Calculate atom matching matrices

Figure 5.1 shows the comparison of GPU and CPU operation time. CPU operation sharply increases runtime with increasing NMax while GPU runtime stays almost the same. This reflects the potential of GPU computing: the computing time does not increase proportionally with the size of inputs. When NA is varied, the speed-ups sharply increase with NMax and plato at a range from 90 times to 150 times for NB = 20 (Figure 5.2). Varying NB, the speed-ups plato at two main ranges 80 and 150 times for NA = 20. The speed-ups are up to 150 times for NB = 10 and 20, while are only about 90 times for NB larger than 30. This could be due to the effect to data locality: if memory access is in the similar location, memory access of all threads in a warp of 32 could be combined in a single request. Otherwise, the request would have to be consequential thus increase run time. In general, this kernel is implemented efficiently with up to 150 times speed-up. CPU/GPU speed-ups depend on the architecture of both CPU (model Q8300) and GPU (model GeForce GT 730).

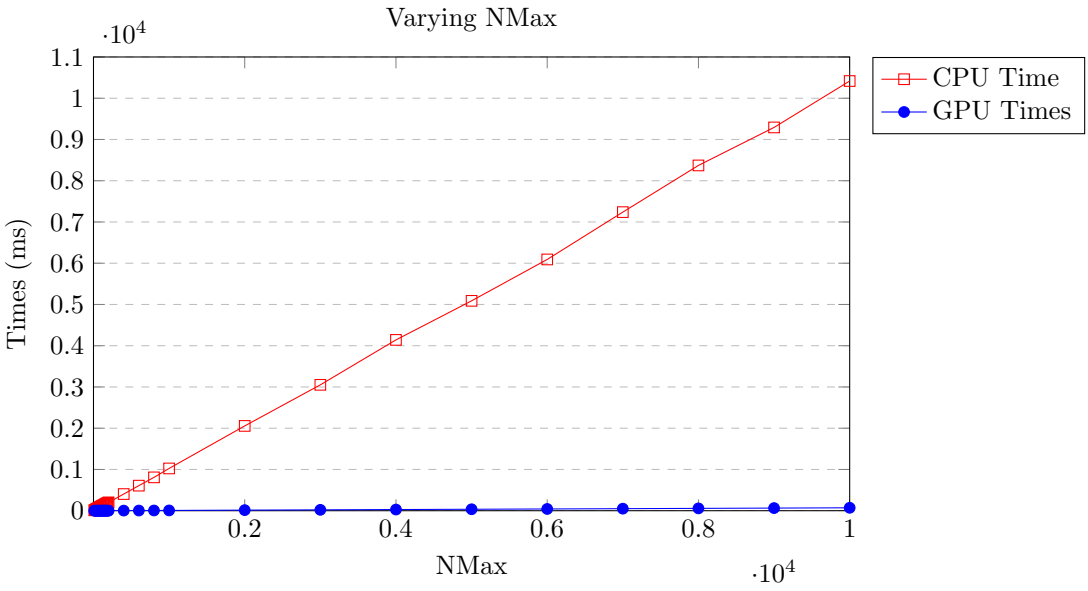


Figure 5.1: Calculate atom matching matrix step: varying NMax, NA=20 and NB = 20

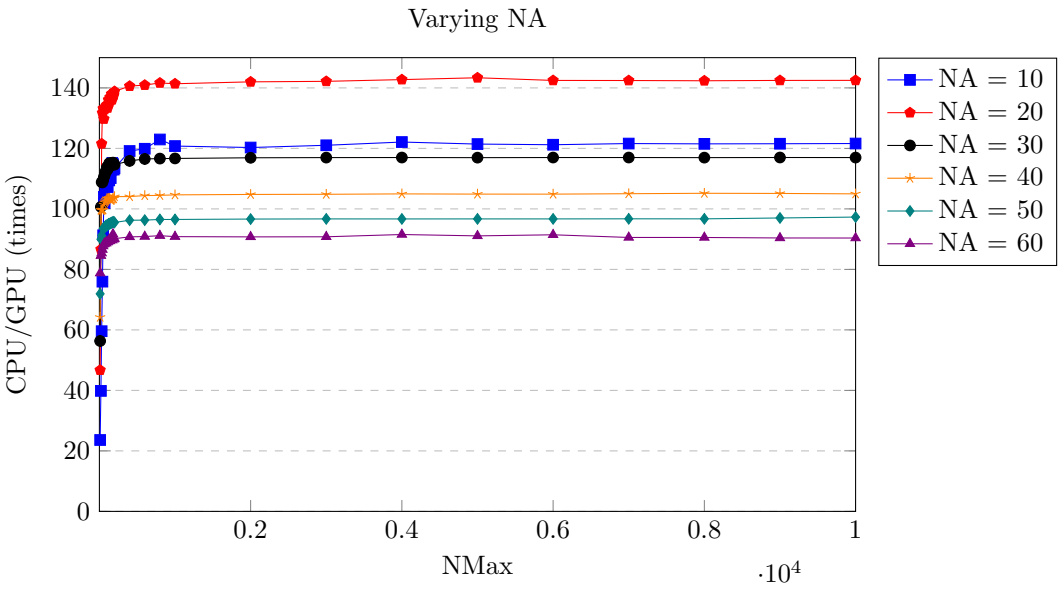


Figure 5.2: Calculate atom matching matrix step: varying NA and NB = 20

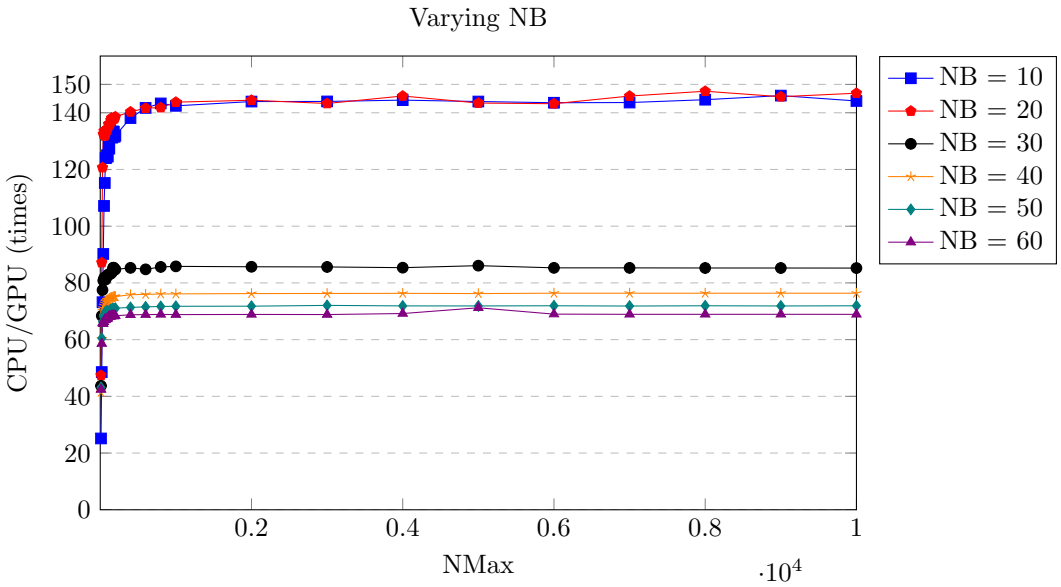


Figure 5.3: Calculate atom matching matrix step: varying NB and NA = 20

5.2.2 Calculate similarity scores

For the "calculate similarity scores" kernel, GPU and CPU operation time grow at a similar rate (Figure 5.4). When NA or NB is varied, the speed-up is up to 3 times. This lower speed-up could be due to the kernel's high ratio of memory access to arithmetic calculation, which CPU is more suitable for. CPU has better cache and logical control uni while GPU is much better at performing arithmetic than memory access. In this case, GPU operation can perform only slightly better than CPU version with intensive memory access required. This reflects the reality that GPU and CPU have their own strengths: there are functions that CPU perform particularly well but not GPU and vice versa.

5.2.3 Overall implementation

The overall implementation is tested with a single query in a real database of 300,000 molecules. Figure 5.7 shows that CPU operation time is rising at a significantly faster rate than GPU implementation's time even if only one query is used and the implementation has a portion of I/O operations that can only be performed by CPU. A speed-up of up to 6 times is achieved for a typical input of a chemical with 20 non hydrogen atoms.

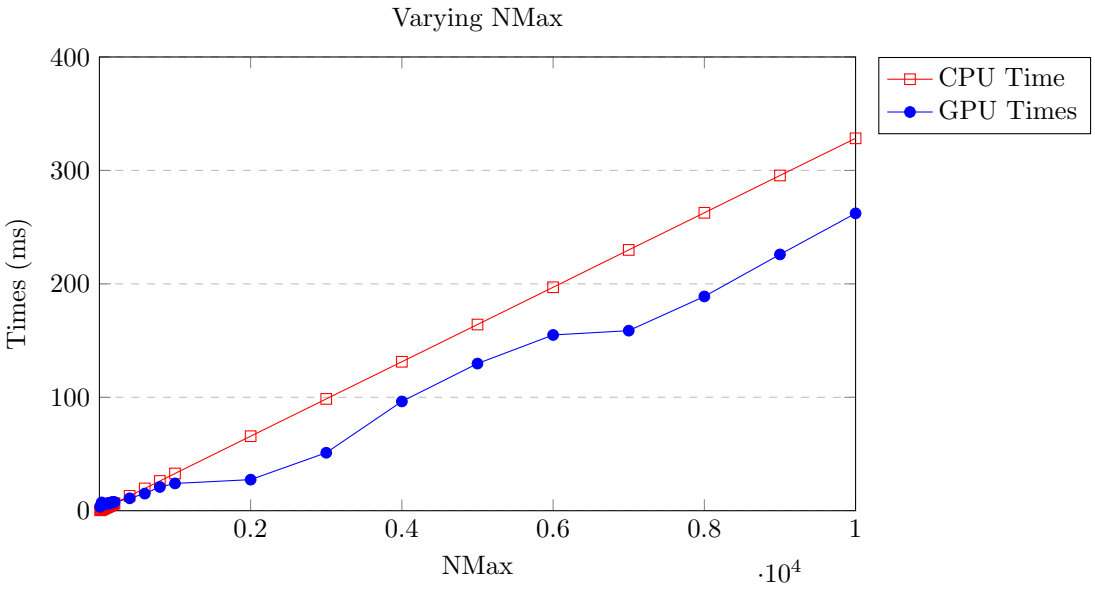


Figure 5.4: Calculate similarity score step: varying NMax, NA =20 and NB = 20

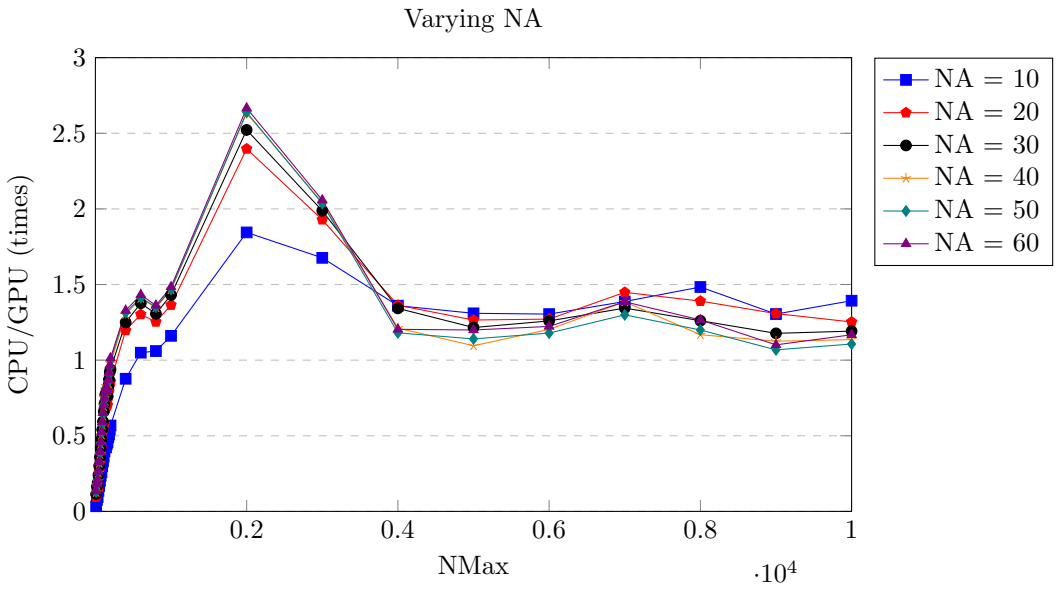


Figure 5.5: Calculate similarity scores step: varying NA and NB = 20

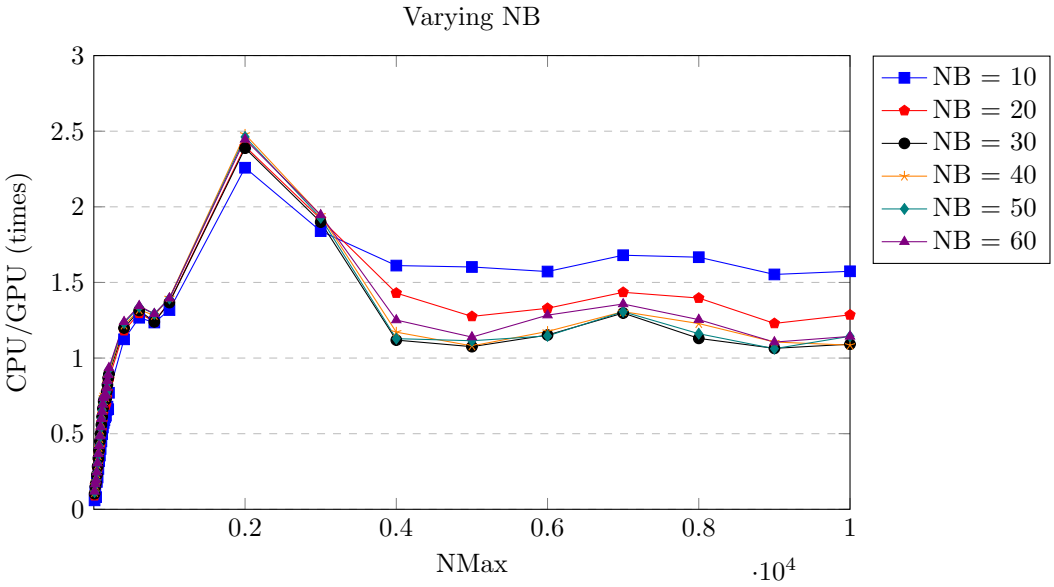
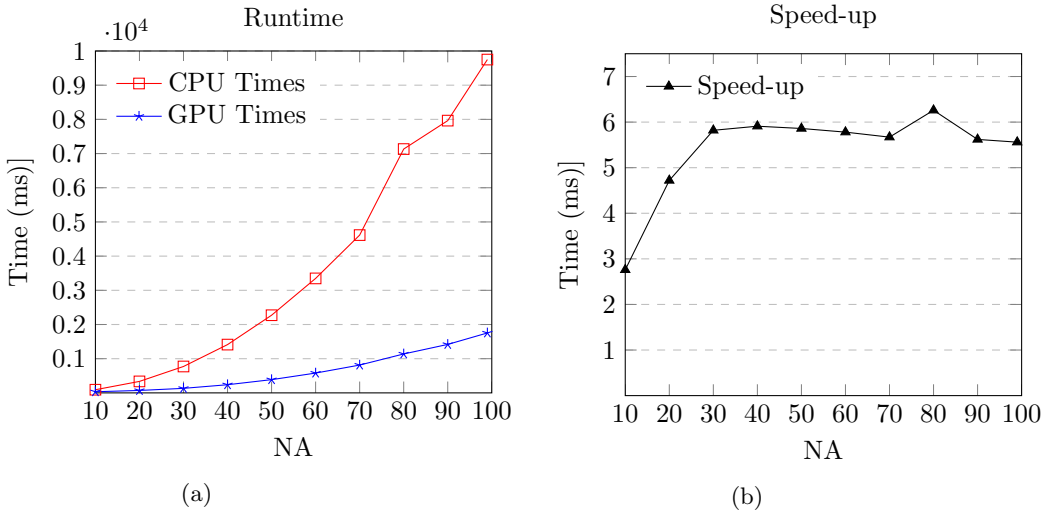


Figure 5.6: Calculate similarity scores step: varying NB and NA = 20



(a) (b)

Figure 5.7: A single query search in a database of 300,000 molecules

Chapter 6

Conclusion

Extensive experiments have demonstrated the strong relationship between structural similarities resulting from the atom mapping method and the corresponding molecule's biological activities (Artymiuk et al. 1992).

In this project, atom mapping method was successfully implemented in MATLAB, C/C++ and CUDA/C. Massively parallel implementation with CUDA C achieves up to 150 time speed-up for the calculating atom matching matrices step. Artymiuk et al. (1992) reported a speed-up of up to 40 times. The whole search operation including many I/O operations has speed-up of up to 6 times even for a single query search. A typical molecule with 20 non hydrogen atoms only takes about 70 second search time in a database of 300,000 structures. Previously, a search in the 180,000 structures database has a response time in the range 20-60 min, the precise time depending primarily upon the size of the target structure (Willett 1995). The implementation of atom mapping using CUDA/C is clearly superior to the previously reported work.

This project demonstrated the huge potential of parallel computing in scientific fields. With recent developments by NVIDIA, CUDA/C is a programmable language that allows scientists to work with GPU efficiently. Many calculations that are previously considered "super-computing applications" will be able to take advantage of GPU's massive parallelism to significantly shorten response time. For example, modeling macro-molecules such as proteins or DNA can be done at atomic level in less time. These enhancement will have tremendous implications to science and medicine.

Appendices

Appendix A

Testing

A.0.4 Unit testing

Each main step in the algorithm is written as an individual modular function. For each implementation (MATLAB, C/C++, CUDA/C), each function was tested with a set of input against the expected outputs (Table A.1, A.2, A.3).

Number	Input Array (A)	Input Array (B)	Expected output	Results	Comments
1	2 2	2 2	2 2	Y	NA = NB
	2 2	2 2	2 3		
2	2 2 2	2 2	2 2	Y	NA >NB
	2 2 2	2 2	2 2		
3	2 2 2	2 2 2 2	3 3	Y	NA <NB
	2 2 2	2 2 2 2	3 3		
4	2 1.5 3	1 2 3	3 1	Y	Threshold
	4 1 0	0 4 1	1 3		= 0.4

Table A.1: Testing plan for atom matching function

A.0.5 Overall correctness testing

MATLAB is an ideal tool for building the first implementation which serves as a golden standard for correctness testing for all other implementations. In this case, MATLAB implementation was written in sequential logic comparing one molecule to only one other molecule. No parallelism was done to ensure the simplicity and ease to debugging. A set of 10 standard compounds was used for correctness testing. The top Similarity scores of comparing one molecule to the set of 10 standard compounds are listed in Table **??**.

Number	Input Array	Expected output	Results	Comments
1	2 2 2 2 2 2 2 2 2	2	Y	All elements are the same
2	1 2 3 4 5 6 7 8 9	(9+5+1)/3 = 4.3	Y	All different elements

Table A.2: Testing plan for calculate similarity function

Number	Input Array	Expected output	Results	Comments
1	1 2 3 a c b	1 2 3 a c b	Y	Already sorted inputs
2	3 2 1 a c b	1 2 3 b c a	Y	Unsorted inputs
3	1 1 1 a c b	1 1 1 a c b	Y	All key inputs are equal
4	1 2 1 a c b	1 1 2 a b c	Y	Not all key inputs are equal

Table A.3: Testing plan for sort function

Rank	Similarity score	Chemical Formula	Compound ID	Compound Name
1	1.0000	C18H12N4O3	CHEMBL6328	
2	0.9938	C18H16ClN3O3	CHEMBL265667	
3	0.9938	C16H8Cl3N3O3	CHEMBL268097	
4	0.9938	C17H11BrClN3O3	CHEMBL6352	
5	0.9216	C17H12ClN3O3	CHEMBL267864	
6	0.9216	C17H12ClN3O3	CHEMBL6329	
7	0.9206	C17H12ClN3O5S	CHEMBL266457	
8	0.8464	C17H13N3O3	CHEMBL6363	
9	0.8407	C17H13N3O3	CHEMBL6362	
10	0.0576	C6H6	CHEMBL277500	Benzene

Table A.4: Result of a query molecule (CHEMBL6328) with database of 10 molecules

Appendix B

Submission's structure and execution

The submitted file consists of one copy of the report and six folders containing code and necessary data:

1. MATLAB-implementation: containing all necessary MATLAB functions and test samples.
2. CPU-implementation: containing all necessary functions and bat files for CPU implemetation
3. GPU-implementation: containing all necessary kernels and bat files for CPU implemetation
4. PerformnaceComparisionCPUGPU: containing all codes that were used for benchmarking CPU and GPU implementation
5. Testing samples: a number of "mol" files for users to test atom mapping method
6. CommonFiles: contains all supported functions that are used in both CPU and GPU implementations
7. Data-processed: text files containing names and distance matrices of about 300,000 chemical structures. This was used to performance bench-marking between CPU and GPU implementation

All folders have pre-compile executable files and a "Read-Me,txt" file explaining the structure and how to run codes.

Appendix C

Discussion

The project is a success with all main objectives fulfilled. The strong emphasis of correctness and quick response time were achieved. All implementations were correct when checking against the golden standard MATLAB implementation. A single query search of a typical molecule with 20 non hydrogen atom in a database of 300,000 molecules merely takes about 70 seconds.

Possible future work would include a batch processing program that allows users to search for multiple queries simultaneously. Batch processing can achieve significantly higher speed-ups by amortising expensive initial set-up cost when data are copied from CPU to GPU. In addition, speed-up of atom matching step could also be further improved by transforming input matrices into a new set-up to exploit the locality of memory. This approach, however, requires significantly more complicated code and data which makes it hard to debug.

Appendix D

Project Management

I am a trained chemist with four year of industry experience. During my previous work, I realised the huge potential of computer science application in chemistry. I entered this MSc program with an aim of eventually working in high performance computing (HPC) which is ery well-suited for scientific applications.
For this summer project, I wanted to apply parallel computing with CUDA/C knowledge to solve a chemistry related problem. I took a course in C/C++ and attended lectures in Parallel and Distributed Computing by Dr. Dan Ghica as an preparation. I discussed with Dan about the project at the end of the second semester while starting my self-training in CUDA. We worked out a project schedule (Table D.1). During the summer project, I have weekly discussion with Dan. The software is then used in a real database at Unilever for a research project.

Week	Date	Main Task 1 (60% time)	Main Task 2 (40% Time)
W1	8-12 Jun	CUDA self-training	Literature review
W2	15-19 Jun	Reading up algorithm	Writing project proposal
W3	22-26 Jun	MATLAB implementation	CUDA self-training
W4	29 Jun-3 Jul	CUDA implementation	Bug Fixing
W5-8	6-31 Jul	Optimising CUDA code	Testing and Bench-marking
W9-10	3-14 Aug	Testing and Bench-marking	Write up literature review
W11	17-21 Aug	Presentation and Demonstration	
W12-14	24 Aug-10 Sep	Prototype testing at Unilever	Write-up report

Table D.1: Working plan

List of Figures

List of Tables

Bibliography

Artymiuk, P. J., Bath, P. A., Grindley, H. M., Pepperrell, C. A., Poirrette, A.R.and Rice, D. W., Thorner, D., Wild, D. J. Willett, P. and Allen, F.: 1992, Similarity searching in databases of three-dimensional molecules and macromolecules, *Journal of Chemical Information and Modeling* **32**, 617–630.

Kirk, D.B.and Hwu, W.: 2013, *Programming Massively Parallel Processors: A Hands-On Approach*, Elsevier Inc., MA, USA.

Pepperrell1, C., Taylor, R. and Willett, P.: 1990, Implementation and use of an atom-mapping procedure for similarity searching in databases of 3-d chemical structures, *Tetrahedron Computer Methodology* **3**(6C), 575–593.

Pepperrell1, C. and Willett, P.: 1991, Techniques for the calculation of three-dimensional structural similarity using inter-atomic distances, *Journal of Computer-Aided Molecular Design* **5**, 455–474.

Pepperrell1, C.A.and Poirrette1, A., Willett, P. and Taylor, R.: 1991, Development of an atom mapping procedure for similarity searching in databases of three-dimensional chemical structures, *Pesticide Science* **33**, 97–111.

Sheridan, P.R.and Miller, D. M., Underwood, J. and Kearsley, K.: 1996, Chemical similarity using geometric atom pair descriptor, *Journal of Chemical Information and Modeling* **36**, 128–136.

Stumpfe, D. and Bajorath, J.: 2011, Similarity searching, *WIREs Computational Molecular Science* **1**, 260–282.

Willett, P.: 1988, Similarity and clustering in chemical information systems, *Journal of Chemometrics* **2**(3), 229.

Willett, P.: 1995, Searching for pharmacophoric patterns in databases of three-dimensional chemical structures, *Journal of Molecular Recognition* **8**, 290–303.