JOURNAL OF CHEMICAL INFORMATION AND MODELING

ARTICLE

pubs.acs.org/jcim

Accelerating Two Algorithms for Large-Scale Compound Selection on GPUs

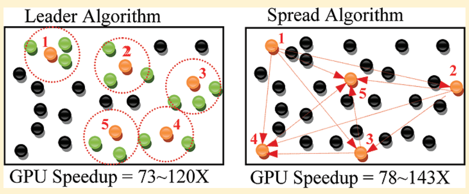
Quan Liao,[†] Jibo Wang,[‡] and Ian A. Watson*[,‡]

[†]ChemExplorer Co. Ltd., 965 Halei Road, Shanghai 201203, People's Republic of China

[‡]Lilly Research Laboratories, Eli Lilly and Company, Lilly Corporate Center, Indianapolis, Indiana 46285, United States

Supporting Information

ABSTRACT: Compound selection procedures based on molecular similarity and diversity are widely used in drug discovery. Current algorithms are often time consuming when applied to very large compound sets. This paper describes the acceleration of two selection algorithms (the leader and the spread algorithms) on graphical processing units (GPUs). We first parallelized the molecular similarity calculation based on Daylight fingerprints and the Tanimoto index and then implemented the two algorithms on GPU hardware using the open source Thrust library. Experiments show that the GPU leader algorithm is 73−120 times faster than the CPU version, and the GPU spread algorithm is 78−143 times faster than the CPU version.



INTRODUCTION

Compound selection procedures are widely used in drug discovery.[1,2] Pharmaceutical companies today often have significant numbers of proprietary compounds in their own inventories and purchase compounds from external vendors in order to increase diversity. Given the current availability of more than ten million commercially available compounds, identifying small subsets that most effectively complement an existing collection in a cost-effective way can be a computationally demanding process.[3] Technologies, such as combinatorial chemistry and high-throughput screening, have offered the possibility of synthesizing and screening up to 1 000 000 compounds in a matter of weeks. Again, considerations of cost often motivate for much smaller, carefully selected sets of compounds to be synthesized or screened.[4]

Many integrated compound selection strategies start with considerations of the lead- or drug-likeness of the candidate molecules. In this stage, property and structural filters are applied to eliminate compounds that would likely be reactive, be insoluble, or have other undesirable absorption, distribution, metabolism, elimination, and toxicity (ADMET) properties. After this initial filtering, candidate molecules are selected according to considerations of similarity or diversity, with the aim of maximizing coverage of chemical space and minimization of redundancy. Other applications include selection of desirable molecules for a given assay. These kinds of rational selection methods should in principle yield better success rates than random selection, as has been validated in comparison experiments.[5]

There are many existing compound selection approaches based on molecular similarity or diversity.[2] These methods differ in two main components, i.e., the molecular similarity calculation method and the selection algorithm.

Molecular similarity calculations require first computing some kind of representation of the molecules, such as topological indices, computed physical properties, two- and three-dimensional (2D and 3D) fingerprints, cell occupancy, or other methods. These computed values are then used in determining similarity metrics such as Tanimoto, cosine, Dice coefficients, Euclidean distance, or other measures.[6] Several works have already reported comparing the performance of different molecular descriptors or different similarity metrics for compound selection.[7−10] Matter et al.[8,9] compared several 2D and 3D descriptors and found that 2D fingerprints work better for designing a general library for lead discovery. Khalifa et al.[10] investigated the relative performance of 12 nonbinary similarity metrics for compound selection and found no obvious winner.

Many selection algorithms have also been reported, dependent or independent of the molecular similarity calculation methods. These selection algorithms can be cataloged into following types: (1) cluster-based selection methods,[11−13] which first identify clusters of compounds and then pick representative compounds of each cluster; (2) dissimilarity-based selection methods,[14,15] which select compounds in an iterative way so that the newly selected compound is most dissimilar to those already selected; (3) partition-based selection methods,[16] which put the molecules into a low-dimensional descriptor space and then partition the space into small cells. Compounds are selected evenly from each cell; (4) optimization-based selection methods,[17−19] which formulate the selection problem as a multiobjective optimization problem by defining some quantitative measure of diversity. Among these algorithms, many are only applicable to small or medium data sets because of their computational complexity. For example, some

Received: December 7, 2010
Published: April 28, 2011

clustering methods require $O(N^2)$ space to store the $N*N$ similarity matrix and $O(N^3)$ time for the clustering of N molecules. And many dissimilarity-based selection methods have a expected time complexity of $O(n^2N)$ for the selection of n diverse molecules from a collection of N molecules. On the contrary, there are also methods that can handle very large data sets. Clustering methods, such as the single linkage (SLINK),[20] leader[13] and Jarvis−Patrick,[21] do not need the store of the whole similarity matrix in memory and also have lower time complexity for the processing. Holliday et al.[22] described an efficient dissimilarity-based selection algorithm, and Higgs et al.[23] developed a spread algorithm; both can approximate a maximally diverse subset with an time complexity of $O(nN)$.

Even using the fastest compound selection methods, such as the leader[13] and spread[23] algorithms, the running time on large data sets can still be significant. It often takes tens of hours to process 1 000 000 compounds on a single central processing unit (CPU), with most of the run time consumed by the calculation of molecular similarities. To speed up the calculations, one can use a quick and rough estimation so that the calculation of similarities between obviously very dissimilar molecules can be skipped. For example, Li[24] introduced an interesting improvement of the leader algorithm. By sorting the molecular fingerprints with their counts of 1-bit, it is only necessary to calculate the similarities between each leader compound and their close neighbors within a small band. We also have CPU implementations of spread and leader algorithms that do not compare molecules if their heavy atom counts differ more than a specified amount.

Another way of speeding up these algorithms is to parallelize them on different kinds of parallel hardware, such as the parallelization of leader algorithm on connection machine[25] and the parallelization of spread algorithm on multiprocessor machine.[23]

Graphical Processing Units (GPUs) have recently become an attractive target for parallel implementations of algorithms useful in computational chemistry. In a previous study,[26] we demonstrated the power of GPU acceleration of molecular similarity calculations for support vector machine-based machine learning. Haque et al.[27] also published a GPU application for calculating chemical similarities based on text comparison of SMILES strings. Interestingly, our work and Haque's work use a very similar strategy to parallelize the Tanimoto similarity calculation with sparse vector representation of fingerprints, and both achieve comparable speedups. Motivated by these results, we now report the GPU acceleration of both the leader and the spread algorithms.

In the following sections, we begin with a description of the hardware and the software infrastructure for GPU programming and then describe the parallelization of molecular similarity calculations. Then we explain the leader and the spread algorithms and highlight the transfer of them to a full GPU implementation. Finally, we measure the performance of our implementations on several data sets and show two orders of magnitude speedups relative to the CPU implementations.

■ METHODS

GPU Hardware and CUDA API. GPUs were originally processors attached to graphics cards for accelerating graphics rendering. With continuing advances in hardware architecture, modern GPUs now yield much higher peak floating point performance than their CPU counterparts and have rapidly evolved for use in general purpose scientific and engineering computing. In this work, we used one newly developed NVIDIA GeForce GTX 280 GPU, which has 1 GB DDR3 RAM (global memory) and 30 multiprocessors (MPs). Each MP is equipped with 8 scalar processors (SPs), 16k registers, and 16 KB of high-bandwidth low latency memory (shared memory).

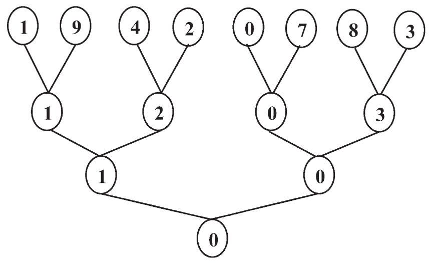


Figure 1. Example of reduction algorithm that determines the minimum of an array.

In order to expose all of the key hardware capabilities of GPUs, NVIDIA provided a programming interface called compute unified device architecture (CUDA).[28] In the CUDA programming model, a large problem is decomposed into a grid of thread blocks. Each block contains up to 512 threads which are concurrently executed on one MP, visiting the same shared memory and synchronizing at specified "barriers" (any threads must stop at this point until all other threads reach this point). One key to the performance of a CUDA program is to distribute the time-consuming calculations into at least thousands of threads so that the large number of cores on the GPU can be effectively utilized. Another important aspect of CUDA programming is the management of different kinds of memory. Since processors on the GPU are not able to directly access CPU memory and vice versa, data must be copied from CPU memory to GPU global memory before any GPU calculation, and the results must be copied back to the CPU memory at the end. It is important to minimize CPU−GPU data transfer because of the relatively slow transfer speeds. Since threads are executed in groups of 32 (called "warps") at the hardware level, it is vital to ensure threads in the same warp access data elements in consecutive locations (called "coalesced access") of the GPU global memory, otherwise the access will slow down by a factor of 2 or more. Accessing the on-chip shared memory is orders of magnitude faster than accessing global memory, so it is often helpful to copy data from global memory to shared memory so as to decrease repeatedly accessing global memory. For more details of the NVIDIA GPU architecture and the CUDA API, please see the CUDA programming guide.[28]

Thrust Library. Thrust[29] is an open source template library based on CUDA. The Thrust library includes many parallel algorithms for sorting, scanning, and other general data manipulation primitives and provides a flexible interface resembling the C++ standard template library (STL). It is developed by NVIDIA experts and closely follows the state of the art in GPU hardware development and algorithm optimizations. With this library, one can directly use highly efficient implementations of many general primitives without reinventing the wheel and focus on domain-specific parts. In this work, we mainly used two components in the Thrust library. Below is a simple introduction to them.

Reduction. Reduction is a common and important data parallel primitive, which uses a binary operation to reduce an input sequence to a single value. For example, the sum of an array is obtained by reducing the array with a plus operation and the

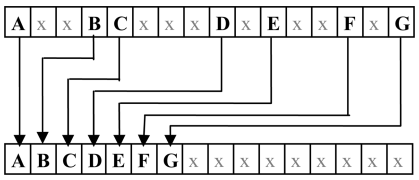

A x x B C x x x D x E x x F x G
A B C D E F G x x x x x x x x x

Figure 2. Example of stream compaction algorithm that collects favorable items from an array.

maximum/minimum of an array is obtained with a max/min operation (see Figure 1). It is easy to implement on a GPU but tricky to achieve optimum performance. Thrust provides an efficient implementation[30] of the reduction algorithm that is very easy to use.

Stream Compaction. Stream compaction[31] is another important primitive building block for parallel algorithms. It is often used to search wanted elements or remove unwanted elements in sparse data (see Figure 2). This allows parallel algorithms to maintain performance over several processing steps. In Thrust, the stream compaction algorithm is implemented in several functions, such as copy_if, remove_if, and partition.

Parallelization of Molecular Similarity Calculation. We use the Tanimoto index (T) to measure molecular similarity (S) or dissimilarity (distance, D) by a simple transformation $D = 1 - S$. In a previous study,[26] we introduced a GPU parallelization of the Tanimoto index calculation. The parallelization is coarse grained, with one row of similarities calculated during each GPU call. The entire fingerprint pool is stored in a column major matrix and kept in global memory. And the query fingerprint is copied to shared memory before calculating any row of the similarities.

Here we use a similar algorithm but based on a different type of molecular fingerprint. The original work was based on counted molecular fingerprints stored in sparse format, similar to Accelrys extended connectivity fingerprints (ECFP),[32] while this work is focused on binary fingerprints of fixed length. This kind of fingerprint is typified by Daylight fingerprints (DYFP),[33] which enumerate all unique paths up to a defined length in a molecular graph. These paths are then hashed into a fixed size bit string.

Whereas the fingerprints described in our previous paper contained a count of how many times a bit was found in a molecule, the fingerprints described here are simple binary (presence, or absence) type fingerprints. While such a representation must necessarily lose information when compared to a fully counted form, the binary representation can enable substantial computational efficiencies by being able to process multiple bits at a time, see below. In addition, the memory requirements for fingerprint storage are also dramatically reduced, which may lead to larger problems becoming feasible.

Given two binary fingerprints fp_a and fp_b, the calculation of Tanimoto index is as follows:

$$T = \frac{c}{a + b - c} \qquad (1)$$

where a and b are sum of 1-bit counts for fp_a and fp_b, and c is sum of 1-bit counts common in fp_a and fp_b. Since a and b can be precomputed for each fingerprint, the main workload is to calculate c. In our programs, each fingerprint is stored in dense format as an array of integers, and the calculation of c is to loop through the integer arrays na and nb using the bitwise AND operation (&) and then using a population count function

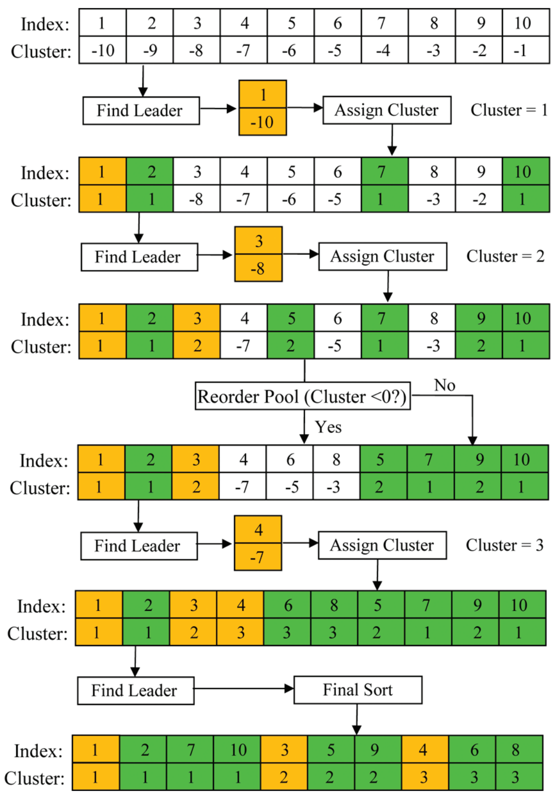

Index: 1 2 3 4 5 6 7 8 9 10
Cluster: -10 -9 -8 -7 -6 -5 -4 -3 -2 -1
Find Leader
1 -10
Assign Cluster
Cluster = 1
Index: 1 2 3 4 5 6 7 8 9 10
Cluster: 1 1 -8 -7 -6 -5 1 -3 -2 1
Find Leader
3 -8
Assign Cluster
Cluster = 2
Index: 1 2 3 4 5 6 7 8 9 10
Cluster: 1 1 2 -7 2 -5 1 -3 2 1
Reorder Pool (Cluster <0?)
No
Yes
Index: 1 2 3 4 6 8 5 7 9 10
Cluster: 1 1 2 -7 -5 -3 2 1 2 1
Find Leader
4 -7
Assign Cluster
Cluster = 3
Index: 1 2 3 4 6 8 5 7 9 10
Cluster: 1 1 2 3 3 3 2 1 2 1
Find Leader
Final Sort
Index: 1 2 7 10 3 5 9 4 6 8
Cluster: 1 1 1 1 2 2 2 3 3 3

Figure 3. Workflow for leader algorithm on GPU. The leaders selected are colored with orange color and the molecules already assigned to clusters are colored with green color.

(popc) to sum the 1-bit counts:

$$c = \sum_{i=1}^{n} \mathrm{popc}(na_i nb_i) \qquad (2)$$

For CPU serial code, a simple but efficient way for the popc function is to precompute all the 256 possible values of one byte {popc(0), ..., popc(255)}, store them in a constant table, and then do the table lookup for every byte. Depending on hardware, an analogous method with two-byte short integers may further improve performance. However, this method is not desirable for GPU processing since memory access is very expensive on the GPU. It is better to calculate the popc values on the fly. Fortunately, CUDA has already provided a optimized intrinsic function __popc(unsigned int) which is based on some simple logic and arithmetic operations.

Parallelization of Leader Algorithm. The leader algorithm is a fast clustering method and finds a set of leaders as cluster representatives, including following steps:

(1) Select the first unselected compound in the data set as a "leader".
(2) Calculate the similarity between every remaining compound and the "leader". If the similarity meets a predefined threshold, then assign it to that cluster.
(3) Repeat (1) and (2) until all compounds are assigned to a cluster.

The leader algorithm is sensitive to the order of compounds. If the compounds are rearranged and scanned in a different order, then the result can be different. In practice, we often sort the molecules by some kind of desirability criterion, such as lead-likeness or predicted activity before clustering, since leader oversamples from those compounds at the front of the list. In the leader algorithm, the final number of leaders cannot be known before the process begins. If it is required to obtain a given number of leaders/clusters, then the process can be repeated using different thresholds until the desired result is obtained.

For a typical leader clustering run, the similarity calculation occupies most of the total running time. So simply parallelizing the similarity calculation will significantly improve the performance. However, according to Amdahl's law, parallelization efficiency is limited by the fraction of the serial code; even 5% serial work will limit the overall speedup to no more than 20 times. In order to get the best possible speedup, we should not only parallelize the similarity calculation but also parallelize other steps that may initially seem negligible. For this purpose, we maintain two arrays on the GPU: an index array which stores the initial index of each molecule in the data set, and a cluster array which stores the cluster number of each molecule.

Our final GPU code for leader algorithm includes the following steps and is illustrated in Figure 3.

(1) Initialization. Copy the fingerprint matrix to the GPU global memory, and initialize the index array to $(1, ..., N)$, initialize the cluster array to $(-N, ..., -1)$.

(2) Find Leader. In the cluster array, unselected molecules are labeled by negative numbers in ascending order, and selected molecules are labeled by positive cluster numbers. So finding the first unselected molecule is equivalent to finding the smallest number in the cluster array, which can be done by a parallel reduction operation in the Thrust library.

(3) Assign Cluster. This is a simple extension to the parallelization of Tanimoto similarity. Each thread first calculates one similarity with the leader molecule and compares it with the predefined threshold. If the value is within the threshold, then it writes the current cluster number to the corresponding position in the cluster array.

(4) Reorder Pool. At each iteration (step 2 and 3), it is only necessary to compute the similarities between the leader and the unselected molecules which are scattered throughout the pool. While this would not cause major problems for CPU sequential code, it may decrease the efficiency for GPU code since some threads are idling. Noting that GPU threads in the same warp should visit contiguous memory locations, it is desirable to gather the unselected molecules (cluster number <0) into a subarray and submit only them to the next GPU calculations. This reordering procedure is done by the parallel stream compaction functions included in the Thrust library. Both the fingerprint matrix and the index/cluster arrays should be reordered at the same time. Ideally we can reorder the data after each run of step 3, but the reordering itself is not without cost. So we only do the reordering when there are enough gaps in the pool that removing them will significantly increase the efficiency. The frequency is empirically determined by experiment.

(5) Final Sort. The iterations (step 2−4) run entirely on the GPU, the only communication between CPU and GPU is to check if the selected leader is valid at each iteration. If the minimum of the cluster array is a positive number, then it means that all molecules have been selected and that the iterations should terminate. For the final process, we sort the cluster array as well as the associated index array by a parallel stable sort function in the Thrust library and then copy these two arrays back to the CPU.

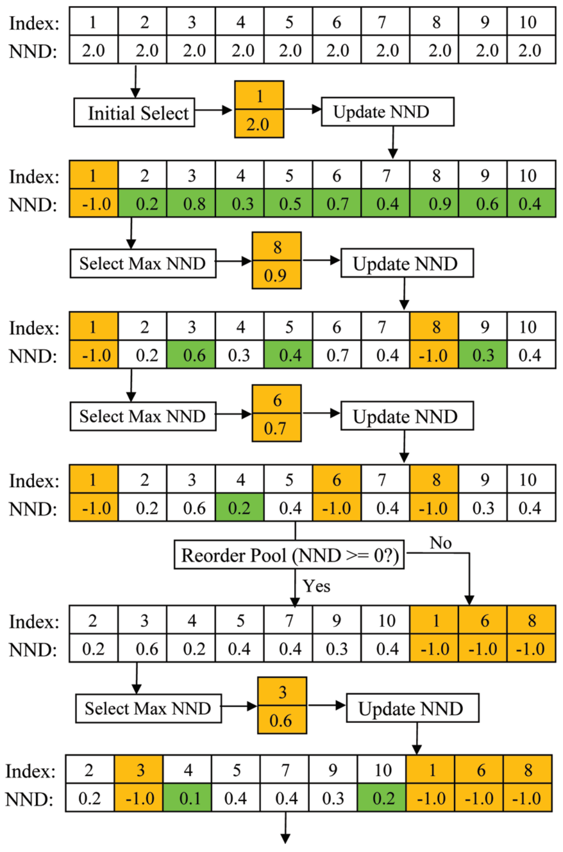

Index: 1 2 3 4 5 6 7 8 9 10
NND: 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0
Initial Select
1 2.0
Update NND
Index: 1 2 3 4 5 6 7 8 9 10
NND: -1.0 0.2 0.8 0.3 0.5 0.7 0.4 0.9 0.6 0.4
Select Max NND
8 0.9
Update NND
Index: 1 2 3 4 5 6 7 8 9 10
NND: -1.0 0.2 0.6 0.3 0.4 0.7 0.4 -1.0 0.3 0.4
Select Max NND
6 0.7
Update NND
Index: 1 2 3 4 5 6 7 8 9 10
NND: -1.0 0.2 0.6 0.2 0.4 -1.0 0.4 -1.0 0.3 0.4
Reorder Pool (NND >= 0?)
No
Yes
Index: 2 3 4 5 7 9 10 1 6 8
NND: 0.2 0.6 0.2 0.4 0.4 0.3 0.4 -1.0 -1.0 -1.0
Select Max NND
3 0.6
Update NND
Index: 2 3 4 5 7 9 10 1 6 8
NND: 0.2 -1.0 0.1 0.4 0.4 0.3 0.2 -1.0 -1.0 -1.0

Figure 4. Workflow for spread algorithm on GPU. The already selected molecules are colored with orange color, and the newly updated NND values are colored with green color.

Parallelization of Spread Algorithm. The spread algorithm[23] is a dissimilarity-based method (see Figure 4). The key of this method is the use of a nearest-neighbor distance (NND) between each candidate molecule and each previously selected molecule. It runs as following:

(1) Select the first molecule at random or by finding the molecule most distant (or closest) to the centroid or other methods.

(2) The NND for the remaining candidate molecules is checked to see if the newly selected molecule is the nearest neighbor to any candidate molecule, and if so, the NND for the candidate molecule is updated.

(3) Select the next molecule that has the largest NND to the already selected set of molecules.

(4) Repeat (2) and (3) until required molecules have been selected.

The parallelization of the spread algorithm on a GPU is quite similar to the parallelization of the leader algorithm. The algorithm also fully runs on the GPU by using two arrays in GPU memory: one is the index array, and the other is the NND array.

(1) Initialization. Copy the fingerprint matrix to the GPU and initialize the index array to $(1, ..., N)$ and the NND array to a number lager than any possible Tanimoto distance, such as 2.0. Then select the first molecule by any desirable method (here we simply use the first molecule in the pool).

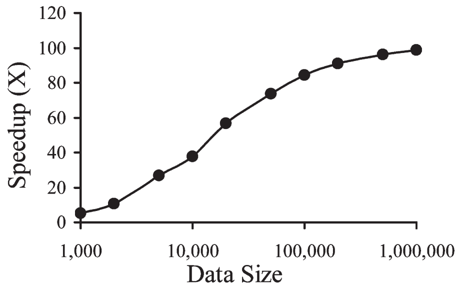


Figure 5. The speedup for Tanimoto similarity is dependent on data size.

(2) Update NND. Each thread calculates one distance between the newly selected molecule and one of the candidate molecules and checks if the distance value is smaller than the corresponding NND. If so, then update this NND element. The NND for this newly selected molecule is reset to a sentinel value such as −1.0, which is smaller than any possible NND value.

(3) Select Max NND. Find the largest value in the NND array by a reduction operation. The associated molecule is the one farthest from already selected molecules.

(4) Reorder Pool. The stream compaction operation is also very useful for the parallel spread algorithm in order to exclude the already selected molecules from the available molecule pool.

Repeat steps 2, 3, and 4 until the required number of molecules are selected. No final process is needed since the result (molecule index sequence) is generated immediately after each molecule is selected.

RESULTS AND DISCUSSION

Data sets. We test the performance of our GPU implementation of leader and spread algorithms on the PubChem Compound Database.[34] The chemical structures are downloaded from the PubChem FTP site.[35] Only normal organic chemicals are considered, and molecules with more than 50 heavy atoms or less than 7 heavy atoms are filtered out. This leads to a clean set with more than 25 million compounds, from which the data sets with different numbers of molecules are randomly selected. For each data set, 2048-bit linear path-based fingerprints (DYFP) were generated by using the default maximum path length of 7 and stored in a TDT format file.

All the experiments are running on a Dell Precision 490 workstation equipped with 4 Intel Xeon 5120 1.86 GHz processors, 2 GB RAM, and an NVIDIA GTX280 card with 1 GB VRAM.

Speedup of Tanimoto Similarity Calculation. According to our previous study,[26] the time for Tanimoto similarity calculation on a GPU (without copying the results back to CPU) is 66−124 times faster than on the CPU when using a sparse format counted fingerprint ECFP. Here we do similar comparisons based on the dense format binary fingerprint DYFP. Ten data sets with different randomly selected compounds, from 1000−1 000 000, were used. And for each set, the same rows of Tanimoto similarities were calculated on both the CPU and the GPU. The speedup of GPU processing over CPU processing is plotted in Figure 5. The speedup is largely dependent on the data size. For the set with 1000 compounds, the speedup is only 5.3 times. While for the set with 1 000 000 compounds, the speedup is 98.7 times. Considering only the large data sets with 100 000 or more compounds, the speedup on DYFP similarity is higher than 80 times, providing an excellent basis for the parallelization of leader and spread algorithms on the GPU.

Table 1. Similarity Matrix Construction Performance

method	data set	time (ms)	throughput (k/sec)
ECFP CPU[a]	4096	43 270	388
ECFP GPU[b]	32 768	57 670	18 619
DYFP CPU[a]	4096	7890	2126
DYFP GPU[b]	32 768	12 940	82 979
LINGO CPU[c]	4096	5460	3070
LINGO GPU[d]	32 768	11 637	92 270

[a] Run on Intel Xeon 5120 (1.86 GHz). [b] Run on NVIDIA GTX280 (240 SPs, 1.3 GHz). [c] Run on AMD Phenom II X4920 (2.8 GHz); data are derived from Table 2 of ref 27. [d] Run on NVIDIA Tesla T10 (240 SPs, 1.44 GHz); data are derived from Table 3 of ref 27.

It is also interesting to compare the relative speed of ECFP and DYFP as well as the LINGO similarity described in ref 27. We measured the time for each method to construct a similarity matrix on a random set of molecules. For GPU methods, times include the time to transfer the fingerprints to the GPU and the time to copy the similarity results back to the host. From the results in Table 1, we can see that DYFP is 5.5 times faster than ECFP on CPU and 4.5 times faster than ECFP on GPU. On the other hand, DYFP is only slightly slower than LINGO on comparable CPU and GPU hardware. Considering that LINGO is derived from a 1D SMILES string and may lose part of the 2D graph-based connection information, we prefer to use DYFP for a better compromise between speed and accuracy.

Speedup of Leader Algorithm. The comparison of the leader algorithm is based on four large data sets, containing 100 000, 200 000, 500 000, and 1 000 000 compounds, respectively. For each of the data sets, we use three similarity thresholds (0.9, 0.8, and 0.7) to generate different clustering results. The run times for both CPU and GPU implementations are listed in Table 2. The speedups range from 73− 120 times, increasing with the data size. As a comparison, we also parallelized the CPU code using Intel threading building blocks (TBB) library,[36] a C++ template library that enables fast parallel programming on multicore and symmetric multiprocessing (SMP) CPUs. The parallel CPU (pCPU) results on our quad-core machine are also listed in Table 2.

Speedup of Spread Algorithm. Comparison of the spread algorithm is based on the same 100 000, 200 000, 500 000, and 1 000 000 data sets as the previous section. Subsets with 20, 50, and 80% of the compounds are selected from each data set. The results in Table 3 show that the GPU version of the spread algorithm is 78−143 times faster than the CPU version. We also parallelized the CPU spread algorithm using the Intel TBB library,[36] and the results on quad-core machine are listed in Table 3.

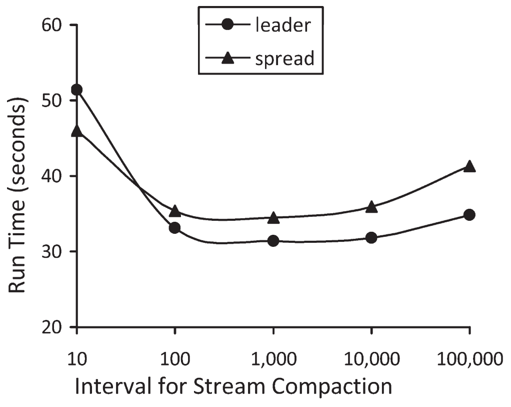
Effect of Stream Compaction. In both GPU leader and GPU spread algorithms, we use the stream compaction primitive to gather the unselected compounds into contiguous memory space. To determine the effect of stream compaction on run time, we repeated the leader and spread algorithms using different intervals for running stream compaction. Figure 6a and b gives the results on 100 000 and 1 000 000 data sets. When running stream compaction after every 10 iterations of compound selection, the overall run times are significantly longer than not using stream compaction at all. Since the time spent on stream compaction itself is more than the benefit from it. It is best

Table 2. Comparison of Leader Algorithm on CPU, Parallel CPU (pCPU), and GPU

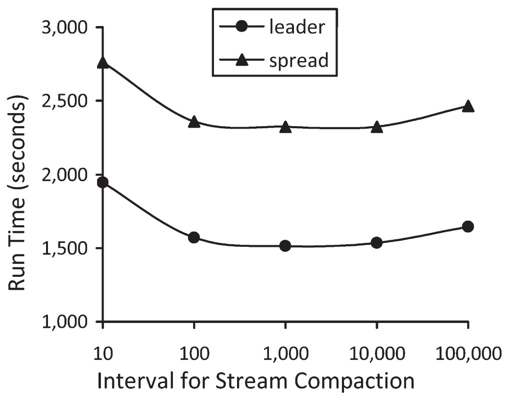
		number of clusters		run time (sec)				
data set	similarity threshold	CPU	GPU	CPU	pCPU	GPU	pCPU/CPU speedup (X)	GPU/CPU speedup (X)
100 000	0.9	93 141	93 124	2821	768	38.7	3.7	73
	0.8	78 719	78 733	2314	642	31.4	3.6	74
	0.7	60 000	60 000	1660	472	22.6	3.5	73
200 000	0.9	178 244	178 210	10 920	3026	119.3	3.6	92
	0.8	141 930	141 962	8202	2361	91.7	3.5	89
	0.7	101 209	101 209	5333	1601	57.6	3.3	93
500 000	0.9	407 767	407 618	60 393	17 071	572.1	3.5	106
	0.8	294 584	294 704	41 262	12 408	354.6	3.3	116
	0.7	193 110	193 115	23 049	7212	192.6	3.2	120
1 000 000	0.9	738 225	737 915	223 291	65 224	1967.0	3.4	114
	0.8	493 107	493 326	132 982	40 902	1218.6	3.3	109
	0.7	302 840	302 845	69 150	22 174	684.4	3.1	101

Table 3. Comparison of Spread Algorithm on CPU, parallel CPU (pCPU), and GPU

		run time (sec)				
data set	subset (%)	CPU	pCPU	GPU	pCPU/CPU speedup (X)	GPU/CPU speedup (X)
100 000	20	1319	371	16.9	3.6	78
	50	3084	866	34.5	3.6	89
	80	4384	1231	47.3	3.6	93
200 000	20	5252	1479	53.5	3.6	98
	50	12 350	3471	109.3	3.6	113
	80	17 439	4901	147.1	3.6	119
500 000	20	32 863	9210	291.3	3.6	113
	50	78 003	21 880	603.2	3.6	129
	80	110 579	30 992	786.5	3.6	141
1 000 000	20	132 009	36 979	1114.6	3.6	118
	50	301 426	84 443	2317.1	3.6	130
	80	428 627	120 030	2996.3	3.6	143



(a) 100K dataset



(b) 1M dataset

Figure 6. Run time dependent on the interval for stream compaction.

to run stream compaction with an interval of 1000−10 000 for both leader and spread algorithms, and for both the 100 000 and the 1 000 000 data sets. In such cases, the use of stream compaction can decrease overall run times by 5−20%. All results in Tables 2 and 3 are generated by using a default interval of 1000 for stream compaction.

Comparison of Selection Results. There are some differences between the selection results done on the GPU and CPU implementations with the same parameters. For example, the leader algorithm running on the GPU and the CPU resulted in a slightly different number of clusters when using the same distance threshold, as displayed in Table 2. The differences mainly come from the numeric difference in similarities (distances), which are single-precision float point data type in both CPU and GPU versions. In the leader algorithm, one compound may be larger than or equal to the similarity threshold in CPU calculation but smaller than the threshold in the GPU calculation and vice visa. In the spread algorithm, the minimal NND value in the CPU and in the GPU calculations may belong to different compounds, and these small numeric differences can change the rank order. Also, when two or more compounds have the same minimal NND, the CPU code will always select the first one, but the parallel reduction algorithm on the GPU is not guaranteed to do so. All these situations occurred with low frequency, and even when they occur, they do not affect the practical utilization of these algorithms. In reality, all we can say is that two equally valid sets of results have been generated, since we have no a priori means of determining which result (CPU or GPU) is more correct.

Presence of Previously Selected Set. In the applications of compound selection, it is not unusual to start from a set of previously selected molecules. For example, in the case of purchasing compounds, the current in-house collection would be used as the pre-existing set (Set A), and the external collection will be used as the working set (Set B). It is very natural to slightly adapt the leader and the spread algorithms (both on CPU and GPU) to handle such a situation by adding a preprocessing step. In the leader algorithm, when provided with Set A, each compound in Set A is used as a leader, and compounds in Set B will be filtered out if they are close to a leader. After that, the first remaining compound in Set B will be used as a leader, and the normal procedure begins. In the spread algorithm, when provided with Set A, each compound in Set A is serially compared with Set B so as to update the NND array of Set B. After that, the first selected compound from Set B would be the one most dissimilar to Set A.

Here we did not provide the speed comparison of CPU and GPU codes in the presence of a large Set A. Since a better solution for such cases is to do the preprocessing step on a large CPU cluster with hundreds to thousands of CPUs. Set B is split into suitably sized chunks, and each chunk is compared with the whole of Set A using the cluster. The results are then summarized into a single file that can be read by both the leader and the spread programs, which then run their normal procedures on the GPU, using this externally computed initialization data.

As we observed, many similarity related applications can be easily sped up using a CPU cluster by splitting into many independent jobs and then joining the results, a classic embarrassingly parallel task. While others, such as the leader and the spread algorithms, are not as easy to parallelize on such distributed memory systems. GPUs provided a good alternative for such applications. It is the role of the practitioner to select the most appropriate hardware for each task.

CONCLUSION

With the rapid increase in compounds involved in drug discovery, fast compound selection algorithms are required for different purposes. We have described GPU parallelization of two algorithms for large-scale compound selection, i.e., the leader and the spread algorithms. Based on the open source Thrust library which contains many important GPU primitives, both the leader and the spread algorithms are implemented fully on GPU hardware, and they achieve two orders of magnitude in speedup relative to the existing CPU implementations. Experiments based on four large PubChem data sets show the speedup is 73−120 times for the leader algorithm and 78−143 times for spread algorithm. A selection procedure on 1 000 000 compounds takes about 30 min on a GPU, while it often takes several days on a single CPU.

Both this and our previous results demonstrate how GPU-based parallelization of existing similarity-based algorithms can significantly extend the usefulness of these algorithms. We are already using these GPU implementations to deliver more timely results to practitioners of drug discovery.

ASSOCIATED CONTENT

Supporting Information. We provide CPU and GPU source code for leader and spread algorithms. This information is available free of charge via the Internet at http://pubs.acs.org.

AUTHOR INFORMATION

Corresponding Author

*E-mail: watson_ian_a@lilly.com. Telephone: 317-277-6704.

REFERENCES

(1) Olah, M. M.; Bologa, C. G; Oprea, T. I. Strategies for Compound Selection. *Curr. Drug Discovery Technol.* **2004**, *1*, 211–220.

(2) Gorse, A. -D. Diversity in Medicinal Chemistry Space. *Curr. Top. Med. Chem.* **2006**, *6*, 3–18.

(3) Lajiness, M.; Watson, I. A. Dissimilarity-Based Approaches to Compound Acquisition. *Curr. Opin. Chem. Biol.* **2008**, *12*, 366–371.

(4) Gillet, V. J. New Directions in Library Design and Analysis. *Curr. Opin. Chem. Biol.* **2008**, *12*, 372–378.

(5) Yeap, S. K.; Walley, R. J.; Snarey, M.; van Hoorn, W. P.; Mason, J. S. Designing Compound Subsets: Comparison of Random and Rational Approaches Using Statistical Simulation. *J. Chem. Inf. Model.* **2007**, *47*, 2149–2158.

(6) Maldonado, A. G.; Doucet, J. P.; Petitjean, M.; Fan, B. T. Molecular Similarity and Diversity in Chemoinformatics: From Theory to Applications. *Mol. Divers.* **2006**, *10*, 39–79.

(7) Brown, R. D.; Martin, Y. C. Use of Structure-Activity Data to Compare Structure-Based Clustering Methods and Descriptors for Use in Compound Selection. *J. Chem. Inf. Comput. Sci.* **1996**, *36*, 572–584.

(8) Matter, H. Selecting Optimally Diverse Compounds from Structure Databases: A Validation Study of 2D and 3D Molecular Descriptors. *J. Med. Chem.* **1997**, *40*, 1219–1229.

(9) Matter, H.; Pötter, T. Comparing 3D Pharmacophore Triplets and 2D Fingerprints for Selecting Diverse Compound Subsets. *J. Chem. Inf. Comput. Sci.* **1999**, *39*, 1211–1225.

(10) Khalifa, A. Al; Haranczyk, M.; Holliday, J. Comparison of Nonbinary Similarity Coefficients for Similarity Searching, Clustering and Compound Selection. *J. Chem. Inf. Model.* **2009**, *49*, 1193–1201.

(11) Downs, G. M.; Barnard, J. M. Clustering Methods and Their Uses in Computational Chemistry. In *Reviews in Computational Chemistry*; Lipkowitz, K. B., Boyd, D. B., Eds.; Wiley-VCH: New York, 2002, Vol. 18, pp 1−40.

(12) Willett, P.; Winterman, V.; Bawden, D. Implementation of Nonhierarchic Cluster Analysis Methods in Chemical Information

Systems: Selection of Compounds for Biological Testing and Clustering of Substructure Search Output. *J. Chem. Inf. Comput. Sci.* **1986**, *26*, 109–118.

(13) Hodes, L. Clustering a Large Number of Compounds. 1. Establishing the Method on an Initial Sample. *J. Chem. Inf. Comput. Sci.* **1989**, *29*, 66–71.

(14) Trepalin, S. V.; Gerasimenko, V. A.; Kozyukov, A. V.; Savchuk, N. Ph.; Ivaschenko, A. A. New Diversity Calculations Algorithms Used for Compound Selection. *J. Chem. Inf. Comput. Sci.* **2002**, *42*, 249–258.

(15) Snarey, M.; Terrett, N. K.; Willett, P.; Wilton, D. J. Comparison of Algorithms for Dissimilarity-Based Compound Selection. *J. Mol. Graphics Modell.* **1997**, *15*, 372–385.

(16) Mason, J. S.; Pickett, S. D. Partition-Based Selection. *Perspect. Drug Discovery Des.* **1997**, *7/8*, 85–114.

(17) Agrafiotis, D. K. Multiobjective Optimization of Combinatorial Libraries. *J Comput.-Aided Mol. Des.* **2002**, *16*, 335–356.

(18) Sharma, P.; Salapaka, S.; Beck, C. A Scalable Approach to Combinatorial Library Design for Drug Discovery. *J. Chem. Inf. Model.* **2008**, *48*, 27–41.

(19) Weber, L. Applications of Genetic Algorithms in Molecular Diversity. *Curr. Opin. Chem. Biol.* **1998**, *2*, 381–385.

(20) Sibson, R. SLINK: An Optimally Efficient Algorithm for the Single Link Cluster Method. *Comput. J.* **1973**, *16*, 30–34.

(21) Jarvis, R. A.; Patrick, E. A. Clustering Using a Similarity Measure Based on Shared Near Neighbors. *IEEE Trans. Comput.* **1973**, *22*, 1025–1034.

(22) Holliday, J. D.; Ranade, S. S.; Willett, P. A Fast Algorithm for Selecting Sets of Dissimilar Molecules from Large Chemical Databases. *Quant. Struct.-Act. Relat.* **1995**, *14*, 501–506.

(23) Higgs, R. E.; Bemis, K. G.; Watson, I. A.; Wikel, J. H. Experimental Designs for Selecting Molecules from Large Chemical Databases. *J. Chem. Inf. Comput. Sci.* **1997**, *37*, 861–870.

(24) Li, W. Z. A Fast Clustering Algorithm for Analyzing Highly Similar Compounds of Very Large Libraries. *J. Chem. Inf. Model.* **2006**, *46*, 1919–1923.

(25) Hodes, L.; Whaley, R. Clustering a Large Number of Compounds. 2. Using the Connection Machine. *J. Chem. Inf. Comput. Sci.* **1991**, *31*, 345–347.

(26) Liao, Q.; Wang, J.; Webster, Y.; Watson, I. A. GPU Accelerated Support Vector Machines for Mining High-Throughput Screening Data. *J. Chem. Inf. Model.* **2009**, *49*, 2718–2725.

(27) Haque, I. S.; Pande, V. S.; Walters, W. P. SIML: A Fast SIMD Algorithm for Calculating LINGO Chemical Similarities on GPUs and CPUs. *J. Chem. Inf. Model.* **2010**, *50*, 560–564.

(28) *NVIDIA CUDA Programming Guide 2.3*; NVIDIA Corporation: Santa Clara, CA, 2009; http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf. Accessed March 1, 2010.

(29) Hoberock, J.; Bell, N. *Thrust: A Parallel Template Library*, version 1.1; NVIDIA Corporation: Santa Clara, CA; http://code.google.com/p/thrust/. Accessed December 1, 2009.

(30) Harris, M. Optimizing Parallel Reduction in CUDA; NVIDIA Corporation: Santa Clara, CA; http://developer.download.nvidia.com/com-pute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf. Accessed December 1, 2009.

(31) Billeter, M.; Olsson, O.; Assarson, U. Efficient Stream Compaction on Wide SIMD Many-Core Architectures. In Proceedings of the Conference on High Performance Graphics 2009, New Orleans, LA, August 1—3, 2009; pp 159—166.

(32) Rogers, D.; Brown, R. D.; Hahn, M. Using Extended-Connectivity fingerprints with Laplacian-Modified Bayesian Analysis in High-Throughput Screening Follow-up. *J. Biomol. Screen.* **2005**, *10*, 682–686.

(33) *Daylight Fingerprints*; Daylight Chemical Information Systems. Inc.: Aliso Viejo, CA; http://www.daylight.com/. Accessed December 1, 2009.

(34) *The PubChem Project*; National Center for Biotechnology Information: Bethesda, MD; http://pubchem.ncbi.nlm.nih.gov. Accessed March 1, 2010.

(35) *The PubChem FTP Site*; National Center for Biotechnology Information: Bethesda, MD; ftp://ftp.ncbi.nih.gov/pubchem/Compound/. Accessed March 1, 2010.

(36) *The Intel Threading Building Blocks*, version 3.0; Intel Corporation: Santa Clara, CA; http://threadingbuildingblocks.org/. Accessed January 23, 2011.