2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

GPU-accelerated protein family identification for metagenomics

Changjun Wu
Xerox Innovation Group
Xerox Research Center
Webster, NY, USA
Email: changjun.wu@xerox.com

Ananth Kalyanaraman
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA, USA
Email: ananth@eecs.wsu.edu

***Abstract*—The clustering of putative protein/Open Reading Frame (ORF) sequences available from large-scale metagenomics survey projects is a core analytical function that has led to the identification and characterization of novel protein families of environmental microbial communities. The implementation of this function, however, is currently challenged not only by data size but also by data complexity. In this paper, we present a CPU-GPU implementation of a randomized graph clustering heuristic called *Shingling*, which was originally developed by Gibson et al. Our implementation uses the CPU and GPU for different stages of computation, using GPUs for the most time-consuming steps. Experimental results of a 2M ocean metagenomics data set obtained from the Sorcerer II Global Ocean Sampling project show that our new implementation is able to achieve a ∼7X speedup over our serial implementation without using asynchronous CPU-GPU communication, with the GPU part alone contributing to over ∼374X speedup in the accelerated part. Qualitative evaluation of the 2M data set shows that our method is able to improve sensitivity of clustering over existing methods, and is more successful in recruiting more sequences into the clustering without impacting the overall specificity. As a demonstration of a large scale run, we were able to cluster a real world homology graph, containing 11M vertices and 640M edges, and constructed from sequences of an ongoing Pacific Ocean metagenomics survey project, in about 94 minutes.**

***Keywords*-Protein family identification; Parallel graph clustering algorithm; Dense subgraph detection; GPGPU application;**

I. Introduction

Protein family identification is a fundamental problem in bioinformatics and computational biology. There are multiple accepted ways of relating protein sequences into families [2] — by full sequence similarity, local sequence or domain level similarity, functional and/or evolutionary relation, etc. [3], [19], [26]. However, in all these different approaches, once the pairwise relationship between proteins is determined, the family composition is typically determined based on a tight clustering formulation — each member of a family is related to most (if not all) of the other members in the family. This problem can be reformulated into a graph-theoretic problem: Given a protein graph $G(V, E)$ where V is a set of n proteins and $(p_i, p_j) \in E$ if and only if protein i is related to protein j, the problem becomes one of identifying tightly knit clusters (alternatively, densely connected components). Although information is sometimes available to assign edge weights in this graph based on the degree of pairwise relationship, the scope of this paper is restricted to unweighted inputs.

Identifying protein families is of fundamental importance to document the diversity of the known protein universe. It also provides a means to determine the functional roles of newly discovered protein sequences. This latter cause has become highly significant of late because numerous genome projects have been completed and as a result there is a sudden expansion of the protein universe. The most dominant contributor to this information-revolution has been the projects in metagenomics. Metagenomics [12], the sequencing and analysis of genetic content obtained from environmental samples, has rapidly emerged as an important area for discovery in environmental microbiology. Metagenomics has applications to bioenergy, environmental biotechnology, pharmaceuticals and agriculture [11]. For over the last several years, there have been hundreds of metagenomics projects [10], [17], [23], [26] resulting in an explosion of metagenomics survey sequences in repositories such as CAMERA [6] and IMG/M [13].

In a typical metagenomics project, DNA material is collected from a target environment of interest (e.g., ocean, acid mine, human gut) and passed through a shotgun sequencing facility [16]. The collected DNA represents a pool of microbes living in that environmental sample, and the shotgun sequencing approach shreds the DNA pool into millions of tiny "fragments", each measuring only a few hundred base pairs (bp). Each of these fragments can then be individually "sequenced" in a laboratory to have its nucleotide sequence determined. The resulting environmental sequence DNA data can be assembled, annotated for genetic regions and subsequently translated into six frames to result in Open Reading Frames (ORFs) or putative protein sequences. For convenience, the terms "proteins", "ORFs" and "sequences" are used interchangeably in the rest of this paper.

A. Related Work

One of the traditional approaches used for clustering ORF data has been to perform an all-against-all protein sequence alignment using an alignment heuristic such as BLAST [1],

978-0-7695-4979-8/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPSW.2013.185

and subsequently cluster them using heuristic approaches based on the resulting protein sequence homology graph. An excellent example of such a large-scale application is the Sorcerer II Global Ocean Sampling project [26], in which a collection of ∼28.6 million ORFs (∼17 million newly sequenced and remaining already processed) were analyzed to result in the identification of thousands of new protein families. However, this analysis took an aggregate of 10^6 CPU hours. The task was brute-force parallelized using 125 dual processors systems and 128 16-processor nodes each containing between 16GB-64GB of RAM.

In 2008, we proposed an alternative approach to performing such large-scale analysis [24]. This approach, which we refer to as the *pGraph-pClust* pipeline, differs from other methods in both phases of analysis:

1) *pGraph:* Homology detection is achieved by first identifying promising pairs of sequences based on a maximal-matching heuristic (suffix trees are used in our implementation to identify such pairs [14]), and subsequently performing the optimality-guaranteeing Smith-Waterman alignment algorithm [20] only on those identified pairs.
2) *pClust:* In order to process the large scale input graph, connected component detection is applied to the input graph to break down the large problem instance into subproblems of much smaller size. For each connected component, we developed an approach based on a well known graph clustering heuristic called *Shingling* developed by Gibson et al. in 2005 in the context of web communities [9] to report clusters.

Our experiments with the *pGraph-pClust* pipeline showed that the combination of techniques significantly improves the quality of the overall clustering, with a particular improvement in cluster sensitivity [24]. We were also able to efficiently parallelize the *pGraph* step to run on thousands of processors on a distributed memory computer [25]. In Daily et al. 2012 [7], an alternative approach based on work stealing is proposed to further enhance scaling to hundreds of thousands of cores.

While the homology detection phase is relatively easier to parallelize, the clustering phase presents a number of challenges for parallelization due to its inherent irregularity in computation and demands in data movement. In Rytsareva et al. [18], we report two very different approaches to parallelize *pClust* — one using shared memory OpenMP parallelization and another using the Hadoop MapReduce model for distributed memory machines. Both these implementations enhanced the problem size reach by a couple of orders of magnitude (to $\times 10^8$ edges). The OpenMP implementation was significantly faster than the Hadoop implementation due to the expensive disk I/O operations involved in the Hadoop platform.

B. Contributions

In this paper, we explore GPUs for parallelizing the clustering step in metagenomics protein family identification. The large number of processing cores on GPUs coupled with the general amenability of multiple stages within the serial clustering algorithm to benefit from data parallelism (described in Section III-B) makes it a promising case of application. However, there are also associated challenges imposed by the GPU model of computing for our clustering implementation — notably, the limited device memory availability, hierarchical memory organization, and overheads associated with data movement. To overcome these challenges, in this paper we present and evaluate a CPU-GPU parallel implementation, which takes advantage of the compute power of the GPUs and the larger memory available in RAM to the CPUs. More specifically, the CPU performs the inherently serial steps within the algorithm in addition to hosting the memory-intensive data structures in RAM, while off-loading the heavy duty computation to GPUs.

Experimental results on 2M sequence graph show that our new implementation is able to achieve a ∼7X speedup over our serial implementation even without using asynchronous CPU-GPU communication, with the GPU part alone contributing to over ∼374X speedup in the accelerated part. Qualitative evaluation over a 2M ocean metagenomics data set shows that our method is able to improve sensitivity of clustering over existing methods, and is more successful in recruiting more sequences into the clustering without impacting the overall specificity. As a demonstration of a large scale run, our GPU accelerated algorithm is able to cluster a real world homology graph, containing 11M vertices and 640M edges, and constructed from sequences from an ongoing Pacific ocean metagenomics survey project, in about 94 minutes.

The rest of the paper is organized as follows: Section II provides a brief introduction to the GPU model of computing. In Section III we define the clustering problem, present the serial algorithm and subsequently present our approach for CPU-GPU parallelization. Experimental results and evaluation of both performance and quality are presented in Section IV. Section V summarizes the findings and provide new directions for further research.

II. A brief introduction to GPU computing

Compute Unified Device Architecture (CUDA) developed by NVIDIA is a hardware and software platform to enable programmers to write general purposed program on GPUs. The GPU application usually contains a copy of host code and a copy of device code. The host code is executed on the CPU, and the device code is executed on the GPU. The memory on the CPU side is referred as the host memory, and the device memory is used to refer memory on the GPU side — including shared memory, const memory and

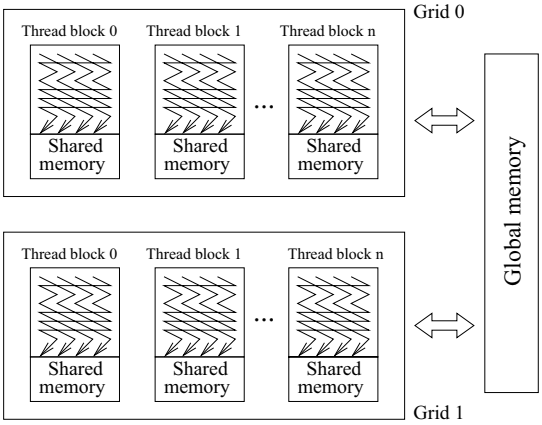

Figure 1. An overview of the hierarchical memory space in GPU.

global memory located on GPU. Data can be copied from host memory to device memory, and vice versa. However, the CPU and the GPU cannot directly access each other's memory space.

The parallelization of the device code is achieved through the execution of kernel functions on individual GPU thread. The threads are organized into thread blocks, and thread blocks are further grouped into grids. Figure 1 shows an overview of the hierarchical memory space in GPU. Each thread inside a thread block has its own local registers and per-thread local memory, and it executes an instance of the kernel. A thread inside a thread block is identifiable through a threadID, and a thread block is identifiable through a blockID inside each grid. Threads inside each thread block are executed concurrently, and they can cooperate with each other though barrier synchronizations or per-block shared memory. Thread blocks are scheduled independently, and the inter-block communication can be achieved through synchronizations on the global memory. The size of the per-block shared memory is much smaller comparing to the size of the global memory, but its memory latency is roughly 100X lower comparing to the latency of the global memory.

To execute the device code on GPU, thread blocks need to be physically mapped to streaming multiprocessors (called SMs in Fermi, and SMXs in Kepler). SMs are independent computation units, and they have their own control units, registers, execution pipelines, caches. One or more thread blocks can be mapped to a SM depending on the size of the thread block. Threads inside a SM are executed in a fixed sized group, called warp, and all the threads in a warp share the same instruction unit. Warp is designed as a single instruction multiple thread (SIMT) architectural unit, and it runs most efficiently if all the threads inside a warp execute same instructions. In case different instructions are programmed into the threads of a warp, the hardware will automatically handle the instruction divergence through multiple rounds of executions.

III. Methods

Notation: Let $S = \{s_1, s_2, \ldots s_n\}$ denote the set of n input protein sequences. Let $G = (V, E)$ represent the similarity graph of the set of sequences S, where each v_i corresponds to an individual sequence in S, and $(v_i, v_j) \in E$ if and only if s_i and s_j have a significant sequence similarity. The similarity relationship between two sequences is symmetric, therefore similarity graph $G = (V, E)$ is an undirected graph. $\Gamma(v_i)$ is used to denote the set of vertices which are adjacent to vertex v_i. In addition, the number of edges in G is denoted as m.

A. Problem Definition

Given a set of protein sequences S, the protein sequence clustering problem is defined as the problem of dividing the protein sequences into subgroups, and each sequence in the subgroup is expected to similar to majority of the sequences included in the same subgroup. In some extreme cases, all the sequences included in the same subgroup are expected to be related to all the other group members, and this extreme case corresponds the clique detection problem in the graph theory domain. However protein sequence clustering problem is a relaxed version of the clique detection problem, and it can also be mapped into a problem of detecting dense subgraphs. This related optimization problem is proved to be NP-Hard [8].

B. The Serial Shingling Algorithm

Definition 1: Given an integer constant s, a "shingle" [5] of a vertex u is defined as an arbitrary $s-$element subset of $\Gamma(u)$.

If two vertices are part of a dense subgraph, they can be expected to share a large fraction of their neighbors in common. Therefore, a brute-force way to detect vertices that are part of the same dense subgraph would be to compute the Jaccard Index (defined in Equation 1) of the two sets of neighbors for every pair of vertices.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

This pairwise neighbor comparison method leads to an expensive quadratical computation. To reduce the computation overhead, a min-wise permutation theory [4] based shingling approach is used to evaluate the similarity between neighbors. More specifically, it obtains random samples of size s (called shingles) from $\Gamma(u)$ for every vertex $u \in V$, and compares them against one another. If two vertices are part of the same dense subgraph, then by definition they should also share most of their neighbors and hence with a high probability of sharing a significant number of their shingles. There are two options of grouping vertices into the same cluster. One option is to group two vertices into the same cluster if they share at least one shingle, and this one shingle based approach can be too aggressive. A second,

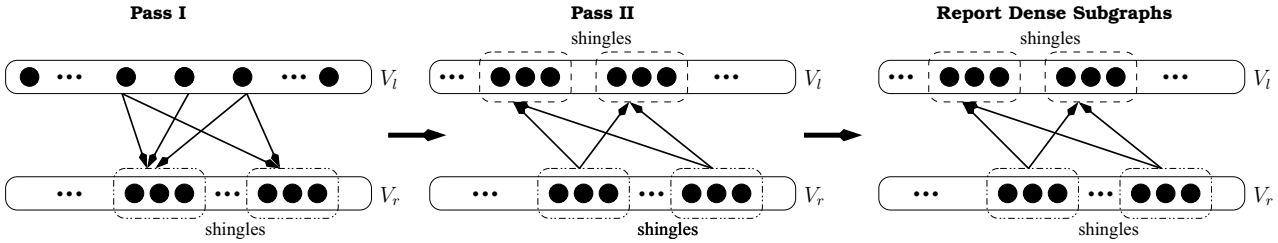

Figure 2. Illustration of the different steps in the Shingling heuristic. Typically, $V_l = V_r = V$.

albeit more conservation approach, is to group two vertices into the same cluster if *all* their shingles are identical. As a middle ground, an approach that groups vertices sharing a significant number of shingles (out of a fixed number c of shingles) is chosen.

Generation of c shingles for any vertex $u \in V$ that has at least s links is achieved as follows [9]: First, c random permutations of the vertices in $\Gamma(u)$ are obtained using a fixed set of c random number pairs $\{<A_j, B_j> | j \in [1, c]\}$. The top s elements within each permutation are then said to represent a *shingle*. The random permutation of $\Gamma(u)$ for a given random trial $j \in [1 \ldots c]$ is obtained as follows: Assume that every $v \in \Gamma(u)$ is associated with a unique integer id. Then a bijection from the set $\Gamma(u)$ to a new set $\Gamma^j(u)$ is computed by taking every $v \in \Gamma(u)$ and mapping it to an element $v^j \in \Gamma^j(u)$ such that $v^j = (A_j \times v + B_j)\%P$, where P is a big prime number. $\Gamma^j(u)$ becomes a random permutation of the original adjacency list $\Gamma(u)$, and its top minimum elements are selected as a shingle subsequently. A permutation thus obtained preserves the min-wise independent property that guarantees, with high probability, that vertices of a densely connected subgraph would also share significant number of shingles.

The serial shingling algorithm containing two passes of shingling followed by a cluster enumeration phase is illustrated in Figure 2. In what follows, we describe the implementation of the Shingling algorithm in our serial implementation *pClust* [24]:

Shingling Pass I: We first treat the input graph $G(V, E)$ in its equivalent bipartite graph representation $G(V_l, V_r, E)$ where $V_l = V_r = V$. The graph is made available as an adjacency list. The algorithm first generates c shingles for each vertex in V_l as described above. In our implementation, the sorting required to generate a shingle from $\Gamma^j(u)$ is implemented by performing an on-the-fly enumeration of $\Gamma^j(u)$ and alongside keeping track of an s-sized array that records the minimum s elements at any point of time through a simple insertion sort. The small values of s expected to be used in practice (typically under 10) justify a simple insertion sort-based approach. Let s_j denote a shingle generated for some vertex during the j^{th} random trial, and assume that it is in an integer representation obtained using a hash function. Since the same shingle s_j could have been generated by multiple vertices in V_l, a sorting is done to gather all vertices that generated each shingle. This shingle is done once for each random trial (so that shingles from different trials do not get mixed). Let $L(s_j)$ denote the set of vertices which generated a shingle s_j. The algorithm then outputs tuples of the form $<s_j, L(s_j)>$. Note that these tuples collectively define a new bipartite graph $G_I(S_1, V_l', E')$ in its adjacency list form, such that S_1 represents the set of distinct shingles generated during this phase, and $V_l' \subseteq V_l$ represents the subset of vertices that contributed to at least one shingle. Therefore, the output of this phase is G_I. We call the shingles in S_1 *first level shingles*.

Shingling Pass II: Using G_I as the new input, the algorithm executes the same series as steps as in Phase I. This generates a new bipartite graph $G_{II}(S_2, S_1', E'')$ in its adjacency list form, such that S_2 represents the new set of shingles generated during this phase (referred to as the *second level shingles*), and $S_1' \subseteq S_1$ represents the subset of first level shingles that contributed to at least one second level shingle in S_2.

Phase III - Reporting dense subsgraphs: The final set of clusters is generated from G_{II}. There are two ways in which this can be accomplished, depending on the stringency required for clustering.

1) Enumerate all connected components in G_{II}. Note that the nodes in each of the connected components of G_{II} represent the first level shingles only, and the second level shingles are used to union the first level shingles. For each connected component, report the set of all vertices in G that constitute the first-level shingles of that component as a "cluster". This formulation could produce potential overlaps between the output clusters, as the same input vertex can be part of two entire different shingles and different connected components.
2) Alternatively, initialize a union-find data structure [21] of size n, with all vertices in G in a cluster by itself initially. Next, enumerate all connected components in G_{II} and For each connected component, perform a union (using the union-find data structure) of the set of all vertices in G that constitute the first- and second-

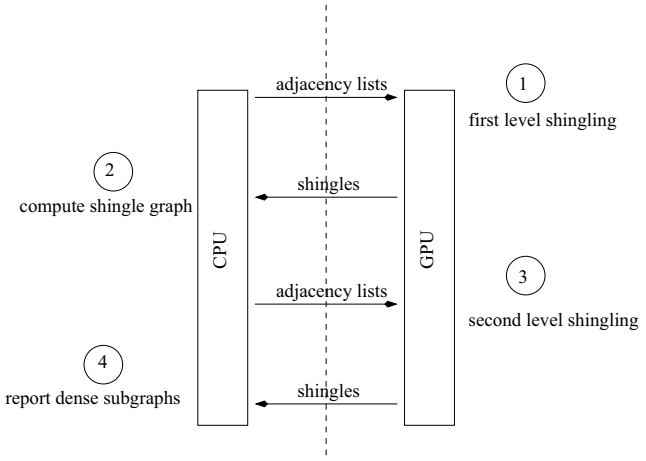

Figure 3. The computing framework of *gpClust*.

level shingles of that component as a "cluster". The clusters reported in this way represent a partition of the input vertices, and no vertex belong two different clusters.

In this paper, we use the second approach which does not allow any overlap between different clusters. The runtime complexity of the implementation is dominated by the shingle generation step in Shingling Phase I and II: $O(m \times c \times s)$. The peak memory complexity of the algorithm is $O(max\{m+n, |E'|\})$. For proofs, please refer to [18].

C. CPU-GPU Parallelization of the Shingling Algorithm

In general, graph algorithms are difficult to be parallelized on the GPU platform, as random memory access is heavily needed while accessing the graph data structure itself. The random memory access pattern will lead to non-coalesced memory access, thus will directly lead to the poor performance of an application. To understand the performance behavior of the shingling algorithm, we implemented a serial version of the shingling algorithm, and profiling results show that roughly 80% of the runtime is consumed by the *hashing* and *sorting* operations in the first and second level shingling steps. Therefore, to parallelize the shingling algorithm on GPU, it is important to focus on the parallelization of these two basic functions. Fortunately, these two functions have been well studied in the GPU community, and they can achieved through two efficient primitives *transform()* and *sorting()* [15] implemented in the Thrust [22] library. Thrust library is a high level parallel library implemented using CUDA C/C++, and it can be used to fast prototype a CUDA program. However, the data movement operations are implemented using synchronous mechanism, and the overhead of transferring data between the host and device memory is unavoidable in the current Thrust release.

From a practical point of view, an effective GPU application should take advantage of the strengths of both CPU and GPU. CPU is designed to minimize the latency experienced by one processor through on-chip caches and sophisticated control logic, while GPU is designed to maximize the overall throughput of the system through massive multithreading. Therefore CPU is very efficient to handle the sophisticated programming logic, while GPU is extremely efficient to perform the massive repetitive tasks. Figure 3 shows the computing framework of our parallel approach. In this framework, CPU is used to aggregate the data for the GPU, and GPU is responsible of the compute-intensive work. Our current implementation is implemented using the Thrust library, and data movement overhead between CPU and GPU is unavoidable because of the synchronous data movement operations implemented in current Thrust. Better performance could be achieved through asynchronous operations provided in CUDA C/C++.

The general idea of our parallel shingling algorithm on GPU is as follows: First, the input graph in an adjacency list format is loaded by the CPU from the I/O into the host memory. Subsequently, loaded adjacency lists are transferred from the host memory to the device memory for the first level shingling. The generated first level shingles are copied back from the device memory to the host memory to prepare a shingle graph for the second level shingling. At the end, the generated second level shingles are transferred back to the host memory to report dense subgraphs. In order to process the large-scale input graph on the relative small device memory, the input graph for the first and second level shingling can be partitioned into batches of adjacency lists, and subsequently moved to the device memory batch by batch.

There are two shingling steps happening on the GPU side, and they are identical from the algorithmic perspective. The only difference is the parametric settings and the input data. In the following, we describe the details of our shingling algorithm on the GPU. As observed in the serial shingling algorithm, shingle extraction operation applies exact the same set of operations to each adjacency list. Therefore, to match the SIMT architectural design of the GPU computing unit, our approach aggregates adjacency lists to the device memory and processes them all at once. As described in the serial shingling algorithm, multiple iterations of shingling are expected on the same adjacency list, and there is no data dependency between each iteration. Therefore, it is safe to transfer the generated shingles back to the host memory after each iteration for the immediate processing on the CPU side. Ideally, we would like to overlap the data communication stage with the shingle computation stage to eliminate the data transfer overhead. Eliminating the data latency will enable the device code and host code to run in parallel, thus leading to a better performance. Current data transfer module implemented in Thrust is synchronous, and therefore our current implementation is not capable of taking advantage of this level of parallelism.

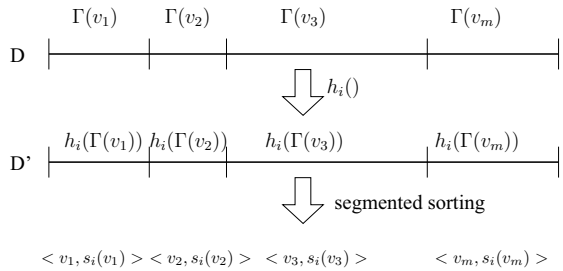

Figure 4. An iteration of shingling step on a batch of adjacency lists on the GPU platform.

Let $H = \{h_1, h_2 \ldots, h_c\}$ denote a set of random hash functions, and they can be implemented through c random number pairs $\{<A_j, B_j> | j \in [1, c]\}$ as described in section III-B. Let $h_i(\Gamma(v_i))$ denote a random permutation of $\Gamma(v_i)$ through h_i. A tuple $<v_m, s_i(v_m)>$ denotes a shingle generated from $h_i(\Gamma(v_m))$.

Algorithm 1 Shingling_on_GPU (D, s, c)

1. *DM* stands for the device memory
2. $H = \{h_1, h_2 \ldots, h_c\}$: a set of random hash functions
3. M_D: a memory space on the DM
4. M'_D: a memory space on the DM
5. R_D: a memory space to store shingles
6. $R \leftarrow \emptyset$
7. $M_D \leftarrow \{\Gamma(v_1), \Gamma(v_2) \ldots\}$
8. **for** $i \leftarrow 1$, c **do**
9. *// can be achieved through thrust::transform()*
10. **for all** $d_p \in M_D$ **do**
11. $M'_D[p] \leftarrow h_i(d_p)$
12. **end for**
13. *// can be achieved through thrust::sort()*
14. Segmented sorting on M'_D
15. **for** $j \leftarrow 1$, m **do**
16. $R_D \leftarrow$ top s elements in $h_i(\Gamma(v_j))$
17. **end for**
18. **end for**

Shingling on the GPU: An iteration of the shingle extraction step on a batch of adjacency lists is illustrated in Figure 4. A batch of adjacency lists is first loaded into an continuous memory space of the global memory on the device, and an auxiliary data structure on the device is used to mark the boundaries of each adjacency list. In case an adjacency list has to be split between two batches, a subsequent data aggregation on the CPU side will remember this case and the merge the different copies of shingles into one correct copy for the split adjacency list. Later a random hash function $h_i \in H$ is applied to each vertex in the batch. The *hashing* operation can be executed highly efficiently on the GPU platform owing to the efficient SIMT architectural design of the GPU computing unit. This random

Algorithm 2 GPU accelerated shingling algorithm

1. *HM* stands for the host memory
2. *DM* stands for the device memory
3. b_i: a batch of adjacency list
4. $graph_H = \{b_1, b_2 \ldots, b_k\}$: partitioned input graph on HM
5. $sglGraph_H = \{sb_1, sb_2 \ldots, sb_k\}$: partitioned input graph on HM
6. $sgls_D$: a memory space on DM
7. $sgls_H$: a memory space on HM
8. *// CPU initiate the task by loading graph into HM*
9. $graph_H \leftarrow$ CPU loads from disk I/O
10. **while** $graph_H \neq \emptyset$ **do**
11. DM $\leftarrow b_i$
12. $sgls_D \leftarrow$ Shingling_on_GPU (DM, s_1, c_1)
13. $sgls_H \leftarrow sgls_D$
14. **end while**
15. *// data aggregation on CPU*
16. $sglGraph_H \leftarrow$ CPU aggregates $sgls_H$ into a graph.
17. **while** $sglGraph_H \neq \emptyset$ **do**
18. DM $\leftarrow sb_i$
19. $sgls_D \leftarrow$ Shingling_on_GPU (DM, s_2, c_2)
20. $sgls_H \leftarrow sgls_D$
21. **end while**
22. *// final data aggregation on CPU*
23. CPU reports dense subgraphs from $sgls_H$

hashing operation will generate a random permutation of the original adjacency list. Based on this random permutation, a segmented sorting operation is applied to reorganize the permutations in each segment. At the end, the top s elements in each segment are selected as a shingle. To enumerate all the shingles through different random permutations, the shingling iteration is executed multiple times with different random hash function $h_i \in H$. Algorithm 1 summarizes the algorithmic steps of the shingle extraction operation on the GPU platform.

Role of the CPU: CPU is extremely efficient to handle the sophisticated programming logics, therefore the task of the CPU is to aggregate the data for the GPU. Initially, CPU is responsible for loading input graph from disk to the host memory, and later the graph is transferred either fully or in partial batches to the device memory (depending on its size). After GPU enumerates shingles, these shingle data are transferred back to the host memory for CPU to prepare the shingle graph for the second level shingling. At the end, all the generated second level shingles are copied back to the CPU, and a connected component detection is executed on CPU to report the dense subgraphs. To handle the case described earlier that an adjacency list has to be split between two job batches, CPU has to combine the shingle results for the split adjacency lists after it receives shingles

# Vertices	# Edges	Avg. degree	Largest CC size
1,562,984	56,919,738	73 ± 153	10,707

Table II
INPUT GRAPH STATISTICS FOR THE SIMILARITY GRAPH OF ∼2M SEQUENCES. CC STANDS FOR CONNECTED COMPONENTS.

from the GPU.

Putting it all together, the GPU accelerated shingling algorithm is described in Algorithm 2.

D. Implementation

The algorithm is implemented in C/C++ using CUDA Thrust parallel library version 1.5. All parameters described in the algorithm section were set to values based on preliminary empirical tests. The default settings are as follows: $s_1 = 2, c_1 = 200$ for the first level shingling, and $s_2 = 2, c_2 = 100$ for the second level shingling.

IV. EXPERIMENTAL RESULTS

There are no standard benchmark data for metagenomic protein families. Therefore to evaluate the quality of our algorithm, we took the predicted protein families from the Sorcerer II GOS project [26]. Section IV-C presents the extensive performance studies of our *gpClust* system, and some comparative quality studies with regard to the GOS clustering method is presented in the section IV-D.

A. Input Data

For our experiments, an arbitrary subset of predicted protein families of 2M sequences from GOS project are used as our benchmark. Based on the 2 million sequences, *pGraph* [25], a parallel software to construct sequence similarity graph is used to build the input graph for our *gpClust* algorithm. Table II shows some basic statistics of our 2M input graph.

B. Experimental Platform

Our experiment was performed on a commodity server equipped with a Tesla K20 card from NVIDIA. The Tesla K20 card has a total number of 2,496 CUDA cores and a 5GB per-board memory. The peak single precision performance of the GPU code is 3.52 teraflops. The CPU node is an 8-core Intel Xeon 2.0 GHz processor with 32 GB RAM running Red Hat Enterprise Linux 6.3.

C. Performance Study

In order to understand the behavior of *gpClust* system, we performed extensive studies on each component of our GPU shingling algorithm. Two dataset are used in our experiments. One dataset with 20K sequences is an arbitrary subset of the 2M sequences, and the other dataset is the 2M sequences itself. The medium sized graph with 20K sequences is chosen to test the scalability of our system. Additionally, the parallel homology graph construction software (*pGraph* [25]) is used to build the input graph for the two sets of sequences in our following experiments.

20K sequence graph: Within the 20K graph, 2,921 vertices are singleton vertices, and they will be ignored in the subsequent analysis as they do not affect the final result. The remaining 17,079 sequences formed a graph of 374,928 edges, and the average vertex degree is 44 ± 69. The serial and GPU implementation of the shingling algorithms complete the 20K graph in 392.32 seconds and 66.75 seconds respectively; thus leading to a ∼6X speedup. In order to better understand our system, a breakdown runtime of each component in our computing framework is reported in Table I. As noted in the table, 52.70 seconds are spent on the CPU side, while only 7.57 seconds are spent on the GPU side. A comparison between the CPU runtime in parallel and CPU runtime in serial further confirmed that 80% of the runtime in the serial implementation is spent on the two levels of shingling. The observation further confirmed our system design choice by moving the most compute-intensive task to the GPU side in order to increase the performance. For this small sized graph, our system looks inefficient since the CPU consumes the 79% of the total runtime - CPU consumes 52.70 seconds, and GPU consumes 7.57 seconds. A careful examination reveals that the shingling task that used to take 339.63 seconds on the GPU in just 7.57 seconds, and this leads to a ∼45X speedup on the GPU accelerated part. Also an extra 4.82 seconds data transfer overhead incurred while transferring the shingling results back from the device memory to the host memory.

2M sequence graph: There are 1,562,984 non-singleton vertices in the 2M sequence graph, and the average vertex degree is 73 ± 153. This graph is sparser than the 20K sequence graph. The overall system speedup is ∼7X, and the GPU accelerated speedup is ∼374X. This extremely high speedup is credited to the powerful GPU node used in our system. Also the increased performance regarding to the 20K sequence graph is contributed by significant increased shingling workload in the 2M graph, as the SIMT architectural GPU is extremely efficient executing these repetitive operations in large scale. The more workload can be executed in parallel on GPU, the better speedup it will contribute to the overall system. Also the data transfer overhead of 108.19 seconds from GPU to CPU can be eliminated through asynchronous data transfer.

D. Comparative Quality Study

An arbitrary subset of predicted protein families from GOS project was used as our benchmark data to perform the qualitative assessment, and the total number of sequences in the benchmark is roughly 2 million. To compute the protein family relationship, the GOS team used a $k-$neighbor linkage (k=10) [26] based graph heuristic to cluster the

#Input graph	#Non-singleton vertices	#Edges	Runtime of each component in *gpClust*						Serial runtime	Total speedup	GPU speedup
			CPU	GPU	Data$_{c\to g}$	Data$_{g\to c}$	Disk I/O	Total runtime			
20K	17,079	374,928	52.70	7.57	1.26	4.82	0.40	66.75	392.32	5.88	44.86
2M	1,562,984	56,919,738	2685.06	447.97	5.99	108.19	28.77	3275.98	23,537.80	7.18	373.71

Table I
SERIAL RUNTIME AND THE RUNTIME OF EACH COMPONENT IN *gpClust* (IN SECS). DATA$_{c\to g}$ DENOTES THE OVERHEAD OF MOVING DATA FROM THE HOST MEMORY TO THE DEVICE MEMORY, AND THE OVERHEAD FROM THE DEVICE MEMORY TO THE HOST MEMORY IS DENOTED AS DATA$_{g\to c}$. GPU SPEEDUP MEANS THE SPEEDUP CONTRIBUTED BY THE GPU ON THE ACCELERATED PART.

sequences, and those reported clusters were further expanded into predicted protein families through profile-sequence and profile-profile matching techniques. The expansion further combined the related sequences or clusters into loosely defined clusters, which represent the membership in protein families. The reason behind this expansion is that sequence-sequence based matching is less sensitive comparing to the profile-based matching techniques, and protein family is a relatively loosed defined term compared to the conventionally accepted clustering concept. The reported clusters from the GOS and our approach are basically "core sets" of the protein families.

To perform a comparative study, reported clusters of our approach and the GOS approach are compared against the benchmark respectively. For notational simplicity, the clusters reported by our approach are labeled as "*gpClust* partition", and the "GOS partition" is used to denote the clusters reported by the GOS $k-$neighbor approach. In the GOS study, only clusters of size $\geq$ 20 are reported, therefore we only use clusters of size ≥ 20 from our *gpClust* approach for the qualitative assessment. To examine the membership quality reported by both approaches, we defined the following measurements. If s_i and s_j are grouped into the same cluster in one approach, and the relationship is also preserved in the benchmark data, then it is marked as a true positive (**TP**). Similarly, we define false positive (**FP**), false negative (**FN**) and true negative (**TN**), and formal definitions are as follows.

Let s_i and s_j be any two sequences in S. Let $g_p(i)$ denote the group that owns s_i in partition p. To compare a test partition ("t") against the benchmark partition ("b"), we classified every such pair (s_i, s_j) into one of the four classes:

1) **TP**: If $g_t(i) = g_t(j)$ and $g_b(i) = g_b(j)$;
2) **FP**: If $g_t(i) = g_t(j)$ and $g_b(i) \neq g_b(j)$;
3) **FN**: If $g_t(i) \neq g_t(j)$ and $g_b(i) = g_b(j)$;
4) **TN**: If $g_t(i) \neq g_t(j)$ and $g_b(i) \neq g_b(j)$.

Using the above definitions, we derived the following measurements:

$$\text{Positive predictive value (\textbf{PPV})} = \frac{\text{TP}}{\text{TP+FP}} \qquad (2)$$

Approach	PPV	NPV	SP	SE
gpClust vs. Benchmark	97.17%	92.43%	99.88%	17.85%
GOS vs. Benchmark	100.00%	90.62%	100.00%	13.92%

Table III
QUALITATIVE COMPARISON OF THE *gpClust* PARTITION AND THE GOS PARTITION AGAINST THE BENCHMARK FOR THE 2M SEQUENCES.

$$\text{Negative predictive value (\textbf{NPV})} = \frac{\text{TN}}{\text{FN+TN}} \qquad (3)$$

$$\text{Specificity (\textbf{SP})} = \frac{\text{TN}}{\text{FP+TN}} \qquad (4)$$

$$\text{Sensitivity (\textbf{SE})} = \frac{\text{TP}}{\text{TP+FN}} \qquad (5)$$

Based on the above defined measurements, comparative quality results of different partitions against the benchmark can be found in Table III. Ideally, we would expect PPV = NPV = SP = SE = 100%, and this means that the evaluated approach reported exactly the same clustering membership as the benchmark. As expected, the reported clusters are "core sets" of the benchmark, therefore the PPV is expected to high, while the sensitivity is expected to be low. In our comparison, the PPVs reported from both approaches are almost 100%, but the sensitivities are relative low for both approaches — 17.85% for the *gpClust* approach and 13.92% for the GOS approach. This high PPV and low SE scenario implies that reported clusters from both approaches are sub-partitions of the protein families in the benchmark. The low sensitivity is expected because both partitions are sequence-sequence matching based approaches, which are less sensitive comparing to the profile-matching based approaches used in the benchmark data. Note the *gpClust* approach outperforms the GOS approach in sensitivity, and this higher sensitivity is contributed by the high configurable s and c parameters used in our approach based on the size of the input graph.

$$\textit{density} = \frac{\text{\#(Edges in a cluster)}}{\text{Total number of possible edges}} \qquad (6)$$

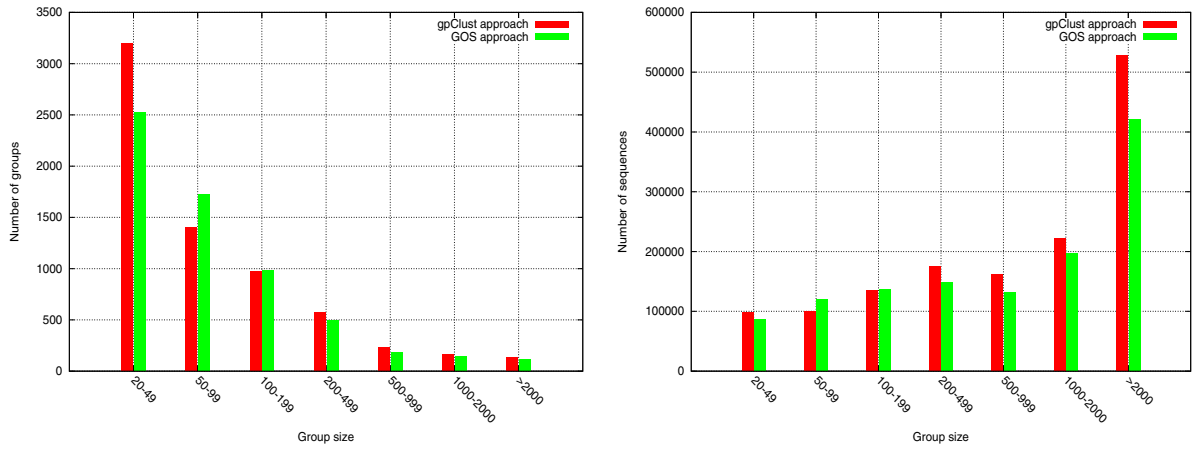

Figure 5. (a) Distribution of the dense subgraphs by their size in the *gpClust* and GOS approaches respectively. (b) Sequence distribution among the different group size bins in *gpClust* and GOS approaches.

Partition	# Groups	# Seqs. included	Group size	
			Largest	Average
Benchmark	813	2,004,241	56,266	$2,465 \pm 4,372$
GOS	6,152	1,236,712	20,027	201 ± 650
gpClust	6,646	1,414,952	19,066	213 ± 721

Table IV
TABLE SHOWING THE STATISTICS OF DIFFERENT PARTITIONS FOR THE 2 MILLION SEQUENCES.

To evaluate the quality of reported clusters, "density" — defined in Equation 6 is used to measure the intra-connectivity of a cluster. The highest expected *density* of a cluster is 1, and it corresponds to a clique. The higher *density* score of a cluster implies a better intra-connectivity among the members in the cluster. According to this definition, the average density of the GOS partition and our *gpClust* partition is 0.40 $\pm$ 0.27 and 0.75 $\pm$ 0.28 respectively. The average density of the benchmark partition is only 0.09 $\pm$ 0.12. The low average *density* of the benchmark data explained the low sensitivities reported in above comparisons, as the protein family is a loosely defined concept in terms of the connectivity of their members. In addition, the higher *density* of our reported clusters demonstrates the high quality of our results comparing to the GOS $k-$neighbor approach. In the GOS approach, two vertices are included into a cluster if they share a fixed number (k) of neighbors, and this clustering strategy makes sense if and only if all the clusters in the input graph are of the same fixed size k; otherwise GOS approach will falsely group potentially unrelated vertices into the same cluster. Furthermore, the choice of k could potentially influence the clustering results for different inputs. A careful examination of the 2M sequence graph shows that the assumption of fixed size clusters does not hold. Also to be noted, the density alone cannot be used as the criterion to evaluate the quality of the reported clusters. For example, if each vertex from the input graph is reported as an individual cluster by itself, then the average density of the reported clusters is 1. However, this high density does not imply the high quality of the reported clusters.

Statistics of reported clusters from both approaches are shown in Table IV. In total, the GOS approach reported 6,152 clusters, and the *gpClust* approach reported 6,646 clusters. A careful examination shows that GOS approach grouped some highly-connected clusters into a relatively loosely-connected cluster due to the limitation of the fixed size k, and that is one of the reasons why the GOS approach reported less number of clusters than our approach. The average group size of the reported clusters in the GOS approach and our approach is 201 ± 650 and 213 ± 721. The high standard deviation of group size in GOS clusters further shows the limitation of the GOS approach in terms of reporting clusters of the correct size.

In Figure 5(a), we compare the group size distribution in the *gpClust* and GOS partitions. As can be observed, both partitions show roughly the same distribution. The *gpClust* partition also reported the "small" clusters of size under 20, and they are not shown in the plot. Figure 5(b) shows how the sequence included in the individual partitions are distributed among different group sized bins. Overall, they have the similar distribution.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a GPU accelerated dense subgraph detection approach, called *gpClust* to compute the "core sets" of protein families in metagenomics data. This approach combined the strengths of both CPU and GPU by off-loading the repetitive compute-intensive workload to

the GPU, while keeping the logic sophisticated data aggregation tasks on the CPU. Based on an arbitrary set of 2M metagenomic sequences from the GOS project, our approach reported a $\sim$7X total speedup and a $\sim$374X GPU accelerated speedup, and further performance could be achieved through asynchronous data transfer primitives provided by CUDA C/C++. As a demonstration of a large scale run, we were able to cluster a real world homology graph, containing 11M vertices and 640M edges, and constructed from sequences of an ongoing Pacific Ocean metagenomics survey project, in about 94 minutes. Additionally, comparative quality studies with regard to the GOS approach confirmed the high quality of our reported clusters.

ACKNOWLEDGMENT

Our sincere thanks to Dr. Steven Hallam and his lab at the University of British Columbia, Vancouver, BC, for providing us with the Pacific Ocean metagenomics data set and helping us with scientific interpretation.

A.K. was funded for this research in parts by DOE award DE-SC-0006516 and NSF grant IIS 0916463.

REFERENCES

[1] S.F. Altschul, W. Gish, W. Miller *et al.* Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[2] R. Apweiler, A. Bairoch and C.H. Wu. Protein sequence databases. *Current Opinion in Chemical Biology*, 8(1):76–80, 2004.

[3] A. Bateman, L. Coin, R. Durbin *et al.* The Pfam protein families database. *Nucleic Acids Research*, 32:D138–141, 2004.

[4] A.Z. Broder, M. Charikar, A. Frieze and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60:630–659, 2000.

[5] A.Z. Broder, S. Glassman, M. Manasse and G. Zweig. Syntactic clustering of the web. *WWW6/Computer Networks*, 29:1157–1166, 1997.

[6] CAMERA - Community Cyberinfrastructure for Advanced Microbial Ecology Research & Analysis. *http://camera.calit2.net*. Last date accessed 6/12/2012.

[7] J. Daily, S. Krishnamoorthy and A. Kalyanaraman. Towards Scalable Optimal Sequence Homology Detection. *Proc. ParGraph'11 - Workshop on Parallel Algorithms and Software for Analysis of Massive Graphs*, 2012

[8] U. Feige, D. Peleg and G. Kortsarz. The dense k-subgraph problem. *Algorithmica*, 29(3):410–421, 2001.

[9] D. Gibson, R. Kumar and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Proc. VLDB Conference*, pp. 721–732, 2005.

[10] S.R. Gill, M. Pop, R.T. DeBoy *et al.* Metagenomic analysis of the human distal gut microbiome. *Science*, 312(5778):1355–1359, 2006.

[11] J. Handelsman. Metagenomics: Application of genomics to uncultured microorganisms. *Microbiology and Molecular Biology Reviews*, 68(4):669–685, 2004.

[12] J. Handelsman, M.R. Rondon, S.F. Brady *et al.* Molecular biological access to the chemistry of unknown soil microbes: a new frontier for natural products. *Chemistry & Biololgy*, 5(R):245–249, 1998.

[13] V.M. Markowitz, N.N. Ivanova, and E. Szeto *et al.* IMG/M: a data management and analysis system for metagenomes. *Nucleic Acids Research*, 36(D):534–538, 2008.

[14] A. Kalyanaraman, S.J. Emrich, P.S. Schnable and S. Aluru. Assembling genomes on large-scale parallel computers. *Journal of Parallel and Distributed Computing*, 67:1240–1255, 2007.

[15] D. Merrill and A. Grimshaw. High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(2):245–272, 2011.

[16] E.W. Myers, G.G. Sutton, A.L. Delcher *et al.* A Whole-Genome Assembly of *Drosophila*. *Science*, 287:2196–2204, 2000.

[17] D.B. Rusch, A.L. Halpern, G. Sutton *et al.* The Sorcerer II Global Ocean Sampling Expedition: Northwest Atlantic through Eastern Tropical Pacific. *PLoS Biology*, 5(3):e77, 2007.

[18] I. Rytsareva, T. Chapman, and A. Kalyanaraman. Parallel algorithms for clustering biological graphs on distributed and shared memory architectures. *International Journal of High Performance Computing and Networking*, In Press, 2013.

[19] O. Sasson, A. Vaaknin, H. Fleischer *et al.* ProtoNet: hierarchical classification of the protein space. *Nucleic Acids Research*, 31(1):348–352, 2003.

[20] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[21] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[22] J. Hoberock and N. Bell. Thrust: A Parallel Template Library. *http://www.meganewtons.com/*, 2010.

[23] J.C. Venter, K. Remington, J.F. Heidelberg *et al.* Environmental genome shotgun sequencing of the Sargasso Sea. *Science*, 304(5667):66–74, 2004.

[24] C. Wu, and A. Kalyanaraman. An efficient parallel approach for identifying protein families in large-scale metagenomic data sets, *Proceedings ACM/IEEE conference on Supercomputing*, pp.1–10.2008

[25] C. Wu, A. Kalyanaraman, and W.R. Cannon. pGraph: Efficient parallel construction of large-scale protein sequence homology graphs. *IEEE Transactions on Parallel and Distributed Systems*, 23(10):1923 – 1933, 2012.

[26] S. Yooseph, G. Sutton, D. B. Rusch *et al.* The Sorcerer II Global Ocean Sampling Expedition: Expanding the Universe of Protein Families. *PLoS Biology*, 5(3):e16, 2007.