2010 Ninth International Workshop on Parallel and Distributed Methods in Verification/2010 Second International Workshop on High Performance Computational Systems Biology

Fast Parallel Markov Clustering in Bioinformatics using Massively Parallel Graphics Processing Unit Computing

Alhadi Bustamam*†, Kevin Burrage*‡ and Nicholas A. Hamilton*
*Institute for Molecular Bioscience, The University of Queensland, Australia
† Department of Mathematics, University of Indonesia
‡COMLAB, Oxford University, UK

Email: alhadi.bustamam@uqconnect.edu.au, kevin.burrage@comlab.ox.ac.uk, n.hamilton@imb.uq.edu.au,

Abstract

Markov clustering is becoming a key algorithm within bioinformatics for determining clusters in networks. For instance, clustering protein interaction networks is helping find genes implicated in diseases such as cancer. However, with fast sequencing and other technologies generating vast amounts of data on biological networks, performance and scalability issues are becoming a critical limiting factor in applications. Meanwhile, Graphics Processing (GPU) computing, which uses a massively parallel computing environment in the GPU card, is becoming a very powerful, efficient and low cost option to achieve substantial performance gains over CPU approaches. This paper introduces a very fast Markov clustering algorithm (MCL) based on massive parallel computing in GPU. We use the Compute Unified Device Architecture (CUDA) to allow the GPU to perform parallel sparse matrix-matrix computations and parallel sparse Markov matrix normalizations, which are at the heart of the clustering algorithm. The key to optimizing our CUDA Markov Clustering (CUDAMCL) was utilising ELLACK-R sparse data format to allow the effective and fine-grain massively parallel processing to cope with the sparse nature of interaction networks datasets in bioinformatics applications. CUDA also allows us to use on-chip memory on the GPU efficiently, to lower the latency time thus circumventing a major issue in other parallel computing environments, such as Message Passing Interface (MPI).

Here we describe the GPU algorithm and its application to several real world problems as well as to artificial datasets. We find that the principle factor causing variation in performance of the GPU approach is the relative sparseness of networks. Comparing GPU computation times against a modern quad-core CPU on the published (relatively sparse) standard BIOGRID protein interaction networks with 5156 and 23175 nodes, speed factors of 4 times and 9 were obtained, respectively. On the Human Protein Reference Database, the speed of clustering of 19599 proteins was improved by a factor of 7 by the GPU algorithm. However, on artificially generated densely connected networks with 1600 to 4800 nodes, speedups by a factor in the range 40 to 120 times were readily obtained. As the results show, in all cases the GPU implementation is significantly faster than the original MCL running on CPU. Such approaches are allowing large-scale parallel computation on off-the-shelf desktop machines that were previously only possible on super-computing architectures, and have the potential to significantly change the way bioinformaticians and biologists compute and interact with their data.

1. Introduction

Recently, the Markov clustering algorithm (MCL) [1], which originally was developed for the general problem of graph clustering, has been adopted in a wide range of applications including in bioinformatics applications [2]–[4]. The algorithm has also been reviewed intensively [5]–[7] and has been shown to be robust and reliable compared to some other clustering algorithms. As applications of MCL expand and the size of datasets increase, there is a strong need for a fast and reliable implementation of MCL. Hence, the parallel implementation of the MCL algorithm is now an important challenge in order that MCL performance may be improved.

Previously, we developed a parallel MCL implementation in an MPI environment with preliminary results showing a good performance improvement [8]. However, the increasing popularity of massive parallel implementation using many cores graphic cards processors (GPUs) has created a new efficient and effective way to do massive parallel computing. Recent publications in bioinformatics applications have shown large performance improvements when using GPUs. A 5- to 6-fold of GPU speedup over a general-purpose CPU is often attained, while in several cases a more than 100-fold speed up is reported by authors (see CUDA Zone websites [9] for examples).

978-0-7695-4265-2/10 $26.00 © 2010 IEEE
DOI 10.1109/PDMC-HiBi.2010.23

Table 1: Feature Comparison of the NVIDIA GPUs

Feature	9800GTX	GTX260M	GTX285
#Streaming Processors	128 cores	112 cores	240 cores
Processor Clock	1688 MHz	1375 MHz	1476 MHz
Off-chips Memory Size	512 MB	1 GB	2 GB
Memory Bandwidth	70.4 GBps	61 GBps	159 GBps
Peak Performance	648 GFLOPS	462 GFLOPS	1063 GFLOPS

There are many important bioinformatics applications that have been developed using GPU such as parallel sequence alignment [10] [11], biosequence database scanning [12], phylogenetics [13], features detection in proteomics [14], and molecular dynamic simulation [15]. An extensive review, especially with regard to drug discovery applications recently appeared in [16]. In the current paper we present a GPU implementation of MCL. We benchmark the implementation on a number of problems and show significant speed-ups over CPU computing in all cases. Thus, we believe GPU implementations of MCL will significantly improve performance over CPU implementations and hence will extend the range of applications of MCL.

2. The rise of multi-core CPUs and many-core GPUs computing

A new era of computing power is now arising due to advances in multi-core CPUs and many-core GPUs. In the 90s, the rate of processor-performance grew exponentially due to the improvement in gates-per-die, clock speed and instruction-level parallelism (ILP). However, around 2003, the improvement in clock speed started to hit physical limits due to power consumption and heat effects, and the increasing of demand for sophisticated ILP also became restrictive. This left the gate count remaining as the major basis for performance improvement. As a result, many manufacturers have reconfigured their improvement strategy to focus on gate count instead of pushing clock rate, in particular to making more cores [17] [18].

The new direction of developing multi-core CPUs and many-core GPUs brings the processor chips into being parallel systems. Higher performance can be achieved by populating the cores with multiple floating-point arithmetic logic units (ALUs) where each ALU performs the same operation on distinct pieces of data. This process can be done efficiently using a *single instruction, multiple data* (SIMD) approach. The common implementation of SIMD processing uses *explicit* short-vector instruction, that provided a SIMD width of four, with the instructions that control the operation of four ALUs. This SIMD width of four model is commonly used in multi-core CPU design as a balance between providing increasing throughput and retaining high-single threaded performance [19].

The modern many core GPUs' design employs both multi-threaded and SIMD to maintain high efficiency. The GPU from NVIDA's 8-series, for example, uses *implicit* sharing and instructions across multiple threads with identical program counters (PCs). The GPU then employs significantly wider SIMD processing (widths ranging from 32 to 64) that support a rich set of operations. The wider SIMD processing enables a GPU to pack many cores densely with ALUs. For example, the NVIDA GeForce 8800 Ultra contains 128 single precision ALUs operating at 1.5GHz. These ALUs are organized into 16 processors (each ALU performs one-32 bit multiply-add per clock) and yield a peak rate of 384 GFlops, compared to a high-end 3-GHz Intel Core 2 CPU contains four cores (two 4-width vector instruction per clock) that is capable of 96 Gflops of peak performance [19]. The more recent GPU cards from NVIDIA we use here give an even greater peak performance (see Table 1).

3. CUDA programming model on GPU

Traditional methods of parallel programming such as message passing interface (MPI) often have limited scaling ability due to serialization and synchronization phases that increase with core count. Hence there is the need for an approach to scale up with core-count without the need to restructure the application architecture every time a new core count is targeted [17] [18].

With the advance of GPU architecture, several major graphics card manufactures have develop language tools to make sophisticated parallel programs in many-cores GPU readily expressible with a few abstractions. In 2007, NVIDIA released a scalable parallel programming model using the C language on NVIDA's GPU cards called *compute unified device architecture* (CUDA). CUDA provides a set of extensions to the standard ANSI C programming language which enable the programmer to do heterogeneous computation using both CPU and GPU. The serial portions of applications can be run on the CPU (called host) and the parallel portions can be massively executed on the GPU (called device/kernel) [20].

Since that release, commodity graphics hardware have become a cost-effective parallel platform to solve many

general problems [21]. In particular, the economical manufacture of GPUs in large numbers with broad availability in the personal computer market today gives the benefit of GPU accelerators for both general and specific programming purposes [16] [22]. Recently, GPUs specifically for scientific applications became available with the release of GPUs for supercomputing systems such as TESLA from NVIDIA [23]. Hence CUDA is now emerging as a new development platform for general purpose high performance computing on GPUs.

4. Markov clustering algorithm and CUDA implementation

Markov clustering [1] is an important bioinformatics algorithm for determining cluster information in graph networks. Recently MCL, which originally was developed for general graph clustering, has been adapted to a wide range of bioinformatics applications, such as protein-protein interaction networks [4] [8]. That MCL is effective, fast, and often more tolerant and robust to noise, has made it an attractive algorithm in the extraction of complexes from interaction networks [5] [7].

4.1. MCL algorithm

MCL uses two simple algebraic operations, *expansion* and *inflation*, on the stochastic (Markov) matrix associated with a graph. The Markov matrix M associated with a graph G is defined by normalizing all columns of the adjacency matrix of G. The clustering process simulates random walks (or flow) within the graph using expansion operations, and then strengthens the flow where it is already strong and weakens it where it is weak using inflation operations. By continuously alternating these two processes the underlying structure of the graph gradually becomes apparent, and there is convergence to a result with regions with strong internal flow (clusters) separated by boundaries within which flow is absent [1].

The MCL **expansion** operator takes the pth power of the matrix M as

$$Exp(M) = M^p. \tag{1}$$

By default $p = 2$. For the MCL inflation operation: given a matrix $M \in \mathbb{R}^{m\times n}, M \geq 0$ and a number $r \in \mathbb{R}, r > 0$, the **inflation** operator $\Gamma_r : \mathbb{R}^{m\times n} \rightarrow \mathbb{R}^{m\times n}$ to M with power coefficient r is defined by:

$$(\Gamma_r M)_{ij} = (M_{ij})^r / \sum_{k=1}^{m} (M_{kj})^r; i = 1...m, j = 1...n. \tag{2}$$

$\Gamma_r M$ is called the inflation matrix of M with a power coefficient r. This inflation process automatically normalizes and creates a new Markov matrix result [1].

The iteration of expansion and inflation processes results in an idempotent matrix with clusters in blocks inside. This final idempotent matrix is chosen when no more significant changes occur in the values of the matrix elements on the current expansion and inflation iteration compared to the previous iteration. The idempotent condition is numerically achieved when the global chaos, according to Equations 3 and 4, of the kth column of matrix M in the current iteration is less then a minimum threshold value e (by default $e = 10^{-3}$), for all k.

$$(chaos)_k = \frac{max((\Gamma_r M)_{ik}, i = 1...m)}{\sum_{i=1}^{m} (\Gamma_r M)_{ik}^2}. \tag{3}$$

$$glb\_chaos = max((chaos)_k, k = 1...n). \tag{4}$$

4.2. Sparse Matrix Representation for efficient GPU processing

Network datasets were encoded using a sparse matrix format, as follows, in the parallel Markov clustering implementation using CUDA. There have been several parallel sparse matrix-vector multiplication (SpMV) algorithms mostly using Compressed Sparse Row (CSR) format [24] and more recently using ELLPACK-R format [25] (described below). In this paper we focused our implementation on using ELLPACK-R format since the evaluation in [25] [26] suggested that the ELLPACK-R structure should give several advantages over the CSR structure when implemented in CUDA. These advantages include coalesced global memory access, non-synchronized execution between different blocks of threads, the reduction of the waiting time or unbalance between threads of one warps, and homogeneous computing within the threads in the warps. However, the ELLPACK-R format consumes more memory space compared to CSR format. We leave a CUDA implementation of MCL in GPU using CSR-format for further investigation in subsequent research.

4.2.1. ELLPACK-R Sparse Matrix Format. Suppose n is the total number of rows and L is the maximum number of non-zero elements per row in a matrix. The ELLPACK-R format requires two arrays of dimension $n \times L$. One, denoted val, to store non-zero array elements, and another, col, to store column indices. An additional vector, rL, of dimension n is used to store the actual length of each row [25] [26]. The vector rL is used here as an effective key factor for searching and distributing each matrix column element in SpMV with CUDA in GPU [25] [26]. For example, let the

matrix M with total nonzero elements $NZ = 10$ be

$$\mathbf{M} = \begin{pmatrix} 0 & 0 & 4 & 0 & -1 \\ -2 & 3 & 0 & 5 & 0 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & -5 & 0 & 2 \\ 1 & 0 & -1 & 0 & 0 \end{pmatrix}. \quad (5)$$

The ELLPACK-R representation format of matrix M in Equation 5 is:

$$\mathbf{val} = \begin{array}{|r|r|c|}\hline 4 & -1 & 0 \\ \hline -2 & 3 & 5 \\ \hline 5 & 0 & 0 \\ \hline -5 & 2 & 0 \\ \hline 1 & -1 & 0 \\ \hline \end{array}, \mathbf{col} = \begin{array}{|c|c|c|}\hline 2 & 4 & 0 \\ \hline 0 & 1 & 3 \\ \hline 1 & 0 & 0 \\ \hline 2 & 4 & 0 \\ \hline 0 & 2 & 0 \\ \hline \end{array}, \mathbf{rL} = \begin{array}{|c|}\hline 2 \\ \hline 3 \\ \hline 1 \\ \hline 2 \\ \hline 2 \\ \hline \end{array}.$$

Furthermore, for performance analysis in Section 5, the sparse density (SpD) ratio is defined as the ratio of total number of non-zero elements (NZ) in the sparse matrix M compared to total number of elements of a fully dense matrix with dimension $n \times n$. Hence, in this example the SpD of matrix M is $\frac{10}{25}$ or 25%.

4.3. Parallel MCL Implementation with CUDA

4.3.1. CUDA Sparse Matrix-Matrix Multiplication. CUDA sparse matrix-matrix multiplication (SpMM) is one of the core modules at the heart of our CUDA-MCL implementation (see more detail in Section 4.3.2). We adopted CUDA sparse-vector multiplication (SpMV) implementation using ELLPACK-R sparse format on GPU from [25] [26] to perform massively parallel CUDA SpMM processes in our CUDA-MCL implementation. To illustrate, suppose we have matrix M with dimension of $n \times n$. The parallelization of the sparse matrix-matrix multiplication $M^2 = MM$ using CUDA threads in GPU is described as follows. Firstly, we assign each CUDA thread i to extract all non-zero elements of column-k of sparse matrix M into column-vector v_k in parallel. Then we launch CUDA SpMV kernel on matrix M against vector v_k to produce vector $u = Mv_k$. The SpMV were performed in parallel by assigning each thread i to compute scalar vector multiplication of row i of matrix M against vector v_k to find element i of vector u independently. The resultant column vector u then was transformed into column-k of ELLPACK-R format of matrix M^2. Furthermore, to completely execute SpMM for $M^2 = MM$ in CUDA, we launch the CUDA SpMV kernel from host program (CPU) for each column-k of matrix M for all $k = 1, ..., n$.

Listing 1: CUDA-MCL algorithm

```
//*************************************
//* chaos = max energy on each iteration.
//* energy = energy tolerant to stop MCL.
//* val, col, rL = the ELLPACK-R form of M.
//* maxL = max number of entries by rows
energy=ETOL; //default=1e-3.
chaos=1.0
SH=1; //SH=1 used shared memory
{copy sparse matrix M to device}
while (chaos>energy)
{ chaos=0.0 ;
  for (int k=0;k<N;k++)
  { CUDA_expansion_kernel(k);
    if (SH==1)
      {CUDA_inflation_SH_kernel(k);
       //* compute chaos
       CUDA_chaos_SH_kernel;}
    else
      {CUDA_inflation_GB_kernel(k);
       //* compute chaos
       CUDA_chaos_GB_kernel;}
   //** restore result into column-k M2
   CUDA_ellpackr_restore_kernel(k);
 }
 //** copy global chaos to host
 copy_chaos_to_host();
 //** copy matrix M2 into M in device
 copy_M2_to_M()
 //** synchronize threats
 cudaThreadSynchronize();
 iters++;
 if (iters>=MAX_ITERS) chaos=0.0;
}
{Interpret M as clusters}
```

4.3.2. CUDA-MCL implementation. The most demanding computing time in original MCL algorithm are in the matrix-matrix multiplication processes of the MCL Expansion module (Equation 1), and also the vector reduction processes both in the MCL Inflation module (to compute column-vector sum for Markov matrix normalization – Equation 2) and in the MCL Chaos module (to compute local and global chaos for MCL stopping criteria – Equation 3 and Equation 4, respectively). So the key factor in improving the original MCL algorithm is to exploit all of these MCL Expansion, Inflation and Chaos modules in parallel [8]. Hence, the CUDA-MCL implementation consists of three main massively parallel threads CUDA kernels: (1) *Expansion kernel* to compute parallel MCL expansion processes; (2) *Inflation kernel* to compute parallel MCL inflation processes; and (3) *Chaos kernel* to compute parallel local and global chaos. In Listing 1 we can see the CUDA-MCL algorithm, which incorporates two different kernels for using GPU memory (global or shared memory modules) for the Inflation and Chaos kernels. As an illustration, to do the CUDA-MCL process in GPU, the dataset which is stored in the ELLPACK-R format including val, col, and rL values firstly is copied from CPU (**host**) into GPU (**device**) and is stored in the device global memory. The Expansion kernel uses global memory to compute $M^2 = MM$ then the inflation

kernel stores the results into global or shared memory to do parallel reduction and normalize the matrix resulting in a new Markov matrix input. The Chaos kernel then computes global chaos also using either global memory or shared memory.

Listing 2: CUDA-MCL: Expansion Kernel to compute vector $u = Mv$

```
//************************************
int tid=threadIdx.x;
int bid=blockIdx.x;
int row=blockDim.x*blockIdx.x+threadIdx.x;
if (row<ncols)
{ //**extract column-vector v from data
  int max=rowL[row]; int i=0;
  while ( i < max )
   {  int pos=i*ncols+row;
      if (pos<matSize){
      if(column[pos]==col)
      {  v[row]=data[pos];
         i=max;}
      else
      {  v[row]=0.0;i+=1;}
 }}
__syncthreads();
//***EXPANSION OPERATION
//***SpMV with ELLPACK-R sparse format
float svalue=0.0;
float tvalue=0.0;
for (int i=0;i<max;i++)
 {  float rowval=0.0;
    float colval=0.0;
    int idxcol=0;
    int idxrow=0;
    int pos=i*ncols+row;
    if (pos<matSize)
      { rowval=data[pos];
        idxcol=column[pos];
        colval=v[idxcol];
        svalue+=rowval*colval;
 }}
__syncthreads();
//***store expansion results in vector u
   u[row]=svalue;
 }}
```

In the CUDA-MCL Expansion kernel, the CUDA SpMM algorithm as described in Section 4.3.1 is used to perform the CUDA-MCL expansion process (see Listing 2) using global memory. Meanwhile, for the Inflation and Chaos kernels, we adopted the parallel reduction *type-5* algorithm from NVIDIA SDK [9]. In both of the Inflation and Chaos kernel with shared memory module, the partial sum on each block is computed locally in each block using shared memory, then the results are sent to global memory to find the global sum over all blocks. Furthermore, to compute the final result, the second reduction phase is done via global memory by the last active block. Finally, the thread-0 of the last active block stores the final result in global memory, to enable this result for all others threads (see Listing 3 for MCL Inflation kernel using shared memory). The results from all blocks are synchronized using CUDA _threadfence()

Table 2: Random Full Datasets

No	Name	#nodes	#interactions
1.	$RFULL1$	1600	2,560,000
2.	$RFULL2$	3200	10,240,000
3.	$RFULL3$	4800	23,040,000

Table 3: Random Sparse Datasets *

No	Name	#nodes	#interactions
1.	$RSPARS1$	1600	768,000
2.	$RSPARS2$	3200	3,072,000
3.	$RSPARS3$	4800	6,912,000

{*} *the sparse density ratio (SpD) is 30%*

Table 4: PPI Datasets (from BioGRID [27], HPRD [28])

No	Name	Source	#nodes	#interactions
1.	$PPI1$	$BioGRID$	5,156	51,050
2.	$PPI2$	$HPRD$	19,599	58,450
3.	$PPI3$	$BioGRID$	23,175	137,104

memory fence function (adopted from from CUDA 2.3 Threadfence SDK toolkit) and then the last active block is determined using CUDA _atomic_inc() atomic function. Meanwhile, in Inflation and Chaos kernels with the global memory module, the partial sum on each block is computed directly via global memory. The thread-0 in each block then stores the partial sum results in an intermediate array (with dimension the same as the total number of blocks in kernel). The _syncthreads() function is then used to synchronize all the partial results. To find the final result, thread-0 in each block then does the second reduction phase on the intermediate array, so each block will have the same final result independently.

To find the best memory approach for the CUDA-MCL, firstly we implemented both global and shared memory kernel in both Inflation and Chaos kernel since in both of them we massively used parallel reduction processes. Basically, global memory kernels are easier to implement and to synchronize compared to shared memory kernel. However the global memory kernel has more latency time than the shared memory kernels. On the other hand, the shared memory kernel uses the very low latency on-chip shared memory, but it has to be accessed, computed, and synchronized locally on each thread-block and needs a second parallel reduction step using global memory for global synchronization of all blocks to get the final computation results [20]. Moreover, to check the optimum numbers of threads per block (TPB) (block size), in order to have the

most effective block size of each kernel to achieve the scalable, fastest and most robust performance, we tested three different sizes of TPB including 128 TPB, 256 TPB, and the maximum one, 512 TPB. The NVIDIA GPUs used here only allow the CUDA kernel to launch a maximum of 512 TPB. The outcome of this testing is described in Section 5.

Listing 3: CUDA-MCL: Inflation Kernel for column-vector u using Shared Memory

```
//************************************
__shared__ float sdata[BLOCK_SIZE];
int blockSize=BLOCK_SIZE;
int tid = threadIdx.x;
int bid= blockIdx.x;
int i = blockIdx.x*blockDim.x + threadIdx.x;
// cached input data
float total=0;
float infval=pow(input[i],r);
sdata[tid]=(i<ncols)? infval:0;
__syncthreads();

// do first reduction in shared mem
if (blockSize>=512)
   {if(tid<256){sdata[tid]+=sdata[tid+256];}
   _syncthreads();}
if (blockSize>=256)
   {if(tid<128){sdata[tid]+=sdata[tid+128];}
   __syncthreads();}
if (blockSize>=128)
   {if(tid<64){sdata[tid]+=sdata[tid+64];}
   __syncthreads();}

#ifndef __DEVICE_EMULATION__
if (tid < 32)
#endif
{ if(blockSize>=64){sdata[tid]+=sdata[tid+32];
      EMUSYNC;}
  if(blockSize>=32){sdata[tid]+=sdata[tid+16];
      EMUSYNC;}
  if(blockSize>=16){sdata[tid]+=sdata[tid+8];
      EMUSYNC;}
  if(blockSize>=8){sdata[tid]+=sdata[tid+4];
      EMUSYNC;}
  if(blockSize>=4){sdata[tid]+=sdata[tid+2];
      EMUSYNC;}
  if(blockSize>=2){sdata[tid]+=sdata[tid+1];
      EMUSYNC;}
}
// write result for this block to global mem
if(tid == 0) srdtn[blockIdx.x] = sdata[0];
__syncthreads();

// do second reduction
// to have the same result in each blocks
if(tid==0){
  total=0.0;
  for (int jj=tid;jj<nblocks;jj++)
      total+=srdtn[jj];
  sdata[tid]=total;
}
__syncthreads();
// normalize each column matrix result
// to setup new Markov for next MCL iteration
input[i]=infval/sdata[0];
}
```

5. Performance comparison results and discussion

5.1. Datasets

For performance testing, several random full network datasets (RFULL) and random sparse network datasets (RSPARS) were created (see Table 2 and Table3). The RFULL and RSPARS were created using random matrix generator rand() and sprand() functions in MATLAB, respectively. Then, we add a loop at each of the network nodes with as its weight a random permutation number generated by randperm() function in MATLAB, in order to boost the random walks or flows in the networks to make one node (with higher loop weight) becomes more attractive than the others. The higher its loop weight, the more attractive this node is then to others. We select the 30% sparse density ratio in RSPARS datasets to evaluate the behaviour of CUDA-MCL performance in the sparse datasets compared to RFULL datasets. We do the preliminary performance test with a focus on random sparse network datasets before moving to the real PPI datasets since the PPI datasets structure are generally relatively sparse.

For the real dataset tests, we extracted several protein-protein interaction datasets from public domain websites, including the BioGRID [27] and human protein reference database (HPRD) [28]. BioGRID is one of the freely available online curated biological interaction datasets, compiled comprehensively for protein-protein and genetic interaction from major organism species and available in wide variety of standardized formats. HPRD consists of a protein database directed toward understanding human protein function. For instance, HPRD has been used to develop a human protein interaction networks based on protein-protein and subcellular localization data [29]. The HPRD datasets were manually curated from published literature using bioinformatics analysis on protein sequences by biologist experts. HPRD datasets are also available online with various standardized data format as well. For our CUDA-MCL performance tests, we use three PPI datasets (as shown in Table 4), including the PPI1 and PPI3 datasets (from BioGRID) and the PPI2 dataset (from HPRD) with dataset size ranging from small (PP1), medium (PPI2) and large (PPI3).

5.2. Testing

To test the CUDA-MCL performance we used a wide range of CPU and GPU pair systems including desktop and laptop computers. We employed three different CPU-GPU pair machines: (1) **SC-9800GTX**, an unbranded desktop PC with a classical single core Intel Pentium 4 CPU paired with 128-core NVIDA 9800GTX GPU; (2) **ASUS-GTX260M**, a branded ASUS G51VX-RX05 laptop with a modern dual-core Intel Centrino CPU paired with 112-core

Table 5: Feature Comparison of Testing Machines

Feature	SC-9800GTX	ASUS-GTX260M	QC-GTX285
Computer System	Desktop	Laptop	Desktop
GPU Model	9800GTX	GTX260M	GTX285
GPU Cores	128 cores	112 cores	240 cores
CPU Model	Pentium 4 3.0 Ghz	Centrino P7350 2.0 Ghz	Phenom II 655 3.4 Ghz
CPU Cores	Single-core	Dual-core	Quad-core
RAM Size	4 GB DDR2-667Mhz	4 GB DDR2-400Mhz	4 GB DDR3-1066Mhz GB
Operating System	Windows XP 32 bit	Windows Vista 64 bit	Windows 7 64 bit
CUDA Version	CUDA 2.3	CUDA 2.3	CUDA 2.3

NVIDIA GTX260M mobile GPU; and (3) **QC-GTX285**, an unbranded desktop PC with a powerful modern AMD Phenom II quad-core CPU paired with a powerful 240-core NVIDA GTX285 GPU (see Table 5 for more detail). Such machines represented the old and currently available computer systems on the market and have been used here to check the different and scalable CUDA-MCL performance on different representatives of CPU-GPU pairs such as a weak CPU against a strong GPU (SC-9800GTX), a strong CPU against a less strong GPU (ASUS-GTX260M) and a very strong CPU against a very strong GPU (QC-GTX285).

Firstly, we want to test the preliminary speed-up performance of the CUDA implementation of MCL algorithm between global memory and shared memory kernels using three different numbers of TPB. This is in order to select the best CUDA-MCL kernel model and the optimum number of TPB for further performance tests. We conducted our preliminary tests on the SC-9800GTX machine with 128-core GPU using the random datasets in Table 2. The results in Figure 1 show that up to $235\times$ and more than $300\times$ speed-ups were achieved in the global and shared memory kernels respectively, on the largest dataset. Hence, a 15-35% speed-up in performance was produced by shared memory kernel compare to global memory kernel. The shared memory kernel also gave a more stable speed-up trend and also scalable performance improvement with increasing dataset size, while the global memory kernel produced a more fluctuating speed-up especially in the kernel with 512 TPB. Moreover, among the three TPB implementations, the 256 TPB model always gave a consistent speed-up improvement along with the highest speed-up gain, especially in the shared memory kernel. Meanwhile, 128 TPB in most cases produced a lower speed-up compare to others. These tests led us to select the shared memory kernel with 256 TPB model as our core model for the CUDA-MCL implementation in further tests.

Once the shared memory/256 TPB architecture was selected, we conducted further performance test on the ASUS-GTX2560 and QC-GTX285 machines. We wanted to check possible CUDA-MCL speed-ups on the current representative laptop and desktop machines in the market today. Since the ASUS-GTX260M machine consists of a less powerful 112-core GPU against a quite stronger new generation dual core mobile CPU, we would expect that this system would produce a quick simulation time on both GPU and CPU, but give a moderate speed-up performance. Meanwhile, the QC-GTX285 machine is actually our main focus test since it has larger VRAM and closely represents the actual desktop computer system in market today. With the most powerful 240-core GPU and quad-core CPU here, QC-GTX285 will obviously produce fastest simulation times on both GPU and CPU. The larger VRAM in this GPU should also benefit the computing capacity.

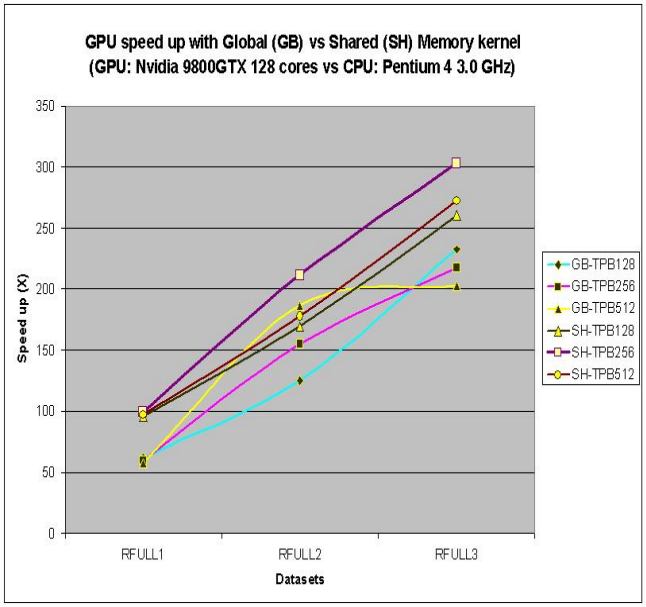

Figure 1: Speed-up on Desktop SC-9800GTX Machine using Global vs Shared Memory Kernel

We ran the tests on both machines using both random full datasets (Table 2) and random sparse datasets (Table 3). Figure 2 shows the speed-up performance results of our CUDA MCL implementation on ASUS-GTX260M and QC-GTX285 machines, using shared memory kernel and 256 TPB model for the RFULL and RSPARS datasets. The results show a very consistent and scalable speedup improvement in all cases on both machines for both random full and sparse network datasets. On the ASUS-GTX260M machine we achieved speed-up by a factor of up to 31× and 22×, while on the QC-GTX285 machine the speed-up increased by a factor of up to 124× and 120×, for both RFULL and RSPARS datasets, respectively. Interestingly, in Figure 2 we can see that the speedup over CPU for random sparse datasets was about 75% of that for random full datasets on ASUS-GTX260M machine, while on QC-GTX285 machine this speed-up was more fluctuative between 70% to 95%.

Finally, to conclude our analysis, we applied the CUDA-MCL implementation to real PPI datasets, with the dataset sizes are ranging from a small (PPI1), medium (PPI2) and large (PPI3) data sizes (see Table 4). In Figure 3 we can see that with the dataset from BioGRID we achieved a speed up by a factor of 4 on PPI1 and of 9 on PP3. Meanwhile, a speed up of a factor of 7 was achieved on the HRPD dataset, PPI2. On the real PPI datasets, we have a very sparse networks in all cases. These very sparse conditions give a drop on of CUDAMCL performance compared to the random full dense or random sparse artificial datasets we generated. Nevertheless, the speed-ups are still a significant improvement in all real PPI dataset cases. As an illustration, on PPI3 dataset we are able to do the clustering with CUDA-MCL on NVIDIA GTX285 GPU within 10 minutes compared to 1 hour and 23 minutes with the original MCL algorithm on quad-core AMD Phenom II 655 3.4GHz CPU.

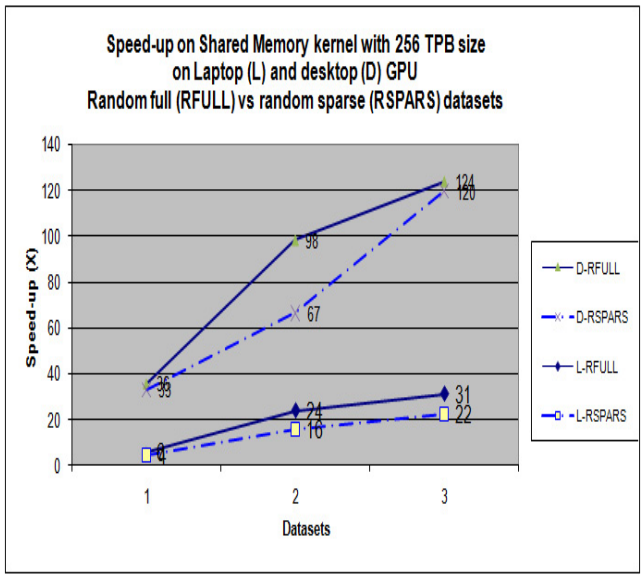

Figure 2: Speed-up on Laptop ASUS-GTX260M and Desktop QC-GTX285 Machines for Artificial Network Datasets

6. Conclusions and Future Work

In this paper, we proposed and evaluated a new approach to the Markov clustering algorithm using GPU computing with CUDA. We proposed our implementation based on an available SpMV package using ELLPACK-R sparse matrix format [25] to compute the parallel expansion processes. We also integrated into our parallel inflation process the parallel reduction method *type-5* from NVIDIA. Our implementations have been tested on a wide range of dataset sizes showing that acceleration factors of up to 300× may be obtained, with the sparseness of the networks being the principle factor effecting the speed-up.

It was also shown that the shared memory implementation increased the speed-up by around 15-35% compared to the global memory implementation of parallel reduction in Inflation and the Chaos kernel. Meanwhile, the 256 TPB model produced more consistent and scalable performance improvement when increasing datasets size. However, on the real PPI datasets with a very sparse nature of the PPI networks, the CUDA-MCL performance dropped but the speed-up achieved were still very significant. Given the general trend in our other tests of improved speed-up on larger datasets we would expect improved performance in larger real world problems.

Due to the relatively large memory usage of the CUDA-MCL implementation using ELLPACK-R sparse data format, we plan to evaluate another approach of Parallel MCL implementation on GPUs using CUDA with CSR sparse matrix format, since the CSR format stores less array elements compared to ELLPACK-R one. However, the dynamic structure on CSR format will likely increase the complexity of parallelization in CUDA and perhaps will reduce the speed-up. Hence, it will be an interesting research problem to increase the CUDA-MCL capability using CSR format without penalizing its speed-up.

As another extension of CUDA-MCL capability, we have also considered hybrid CUDA and openMP implementations (hybrid CUDA/OpenMP) which enable the exploitation of multi-core CPU and many-core GPUs in multi-GPU cards using OpenMP and CUDA, respectively. OpenMP might be used to manage each thread in a multi-core CPU to split the parallel task locally into each GPU unit in multi-GPUs. Furthermore, CUDA could be employed in all GPU units to exploit the power of their massively parallel threads to do each local task in each GPU unit in parallel. This

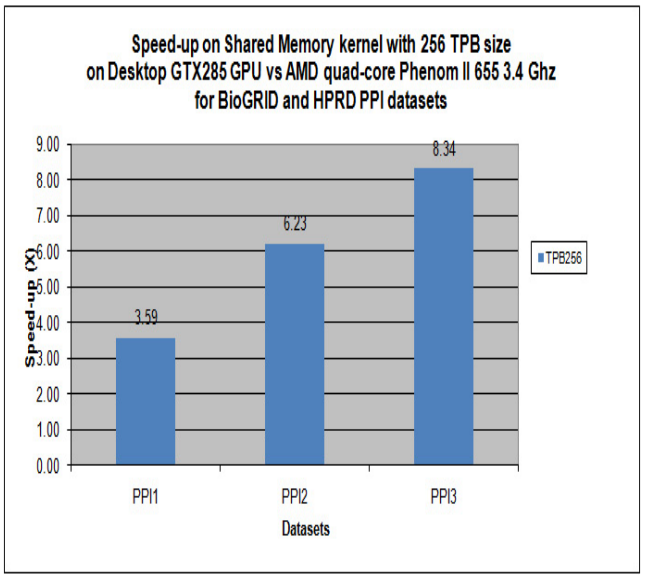

Figure 3: Speed-up on Desktop QC-GTX285 Machine for PPI datasets

approach will give us more accessibility, computing power and capacity with more cores and more VRAM from all of GPUs cards, all locally in a single desktop machine. These CUDA/openMP approaches can be considered as a cheaper, more scalable, efficient hybrid method to solve the large dataset and memory problems of traditional hybrid openMP/MPI in supercomputers or distributed computing machines.

Acknowledgment

This work has been supported by AUSAID scholarship and ARC Center of Excellence in Bioinformatics at IMB the University of Queensland.

References

[1] S. V. Dongen, "Graph clustering via a discrete uncoupling process," *SIAM J. Matrix Anal. Appl.*, vol. 30, no. 1, pp. 121–141, 2008.

[2] A. Enright, S. van Dongen, and C. Ouzounis, "An efficient algorithms for large scale protein families," *Nucleic Acids Research*, vol. 30, pp. 1575–1584, 2002.

[3] T. Harlow, J. Gogarten, and M. Ragan, "A hybrid clustering approach to recognize of protein families in 114 microbial genomes," *BMC Bioinformatics*, vol. 5, p. 45, 2004.

[4] S. Wong and M. A. Ragan, "MACHOS: Markov clusters of homologous subsequences," *Bioinformatics*, vol. 24, no. 13, pp. i77–i85, 2008.

[5] S. Brohée and J. van Helden, "Evaluation of clustering algorithms for protein-protein interaction networks," *BMC Bioinformatics*, vol. 7, p. 488, 2006.

[6] R. Sharan, I. Ulitsky, and R. Shamir, "Network-based prediction of protein function," *Molecular System Biology*, vol. 3, no. 88, 2007.

[7] J. Vlasblom and S. J. Wodak, "Markov clustering versus affinity propagation for the partitioning of protein interaction graphs," *BMC Bioinformatics*, vol. 10, no. 99, September 2009. [Online]. Available: http://www.biomedcentral.com/1471-2105/10/99

[8] A. Bustamam, M. S. Sehgal, N. Hamilton, S. Wong, M. A. Ragan, and K. Burrage, "An efficient parallel implementation of markov clustering algorithm for large-scale protein-protein interaction networks that uses MPI," in *Proceeding of The 5th IMT-GT International Conference on Mathematics, Statistics, and their Applications (ICMSA)*, ser. Computational Mathematics, I. M. Arnawa, Muhafzan, Maiyastri, and S. Bahri, Eds., June 2009, pp. 94–101.

[9] NVIDIA. (2009) CUDA zone. Online. [Online]. Available: http://www.nvidia.com/object/cuda_home.html

[10] S. A. Manavski and G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for smith-waterman sequence alignment," *BMC Bioinformatics*, vol. 9 (Suppl 2), p. S10, 2008. [Online]. Available: http://www.biomedcentral.com/1471-2105/9/S2/S10

[11] S. Jung, "Parallelized pairwise sequence alignment using CUDA on multiple GPUs," *BMC Bioinformatics*, vol. 10, no. Suppl. 7, p. A3, 2009. [Online]. Available: http://www.biomedcentral.com/1471-2105/10/S7/A3

[12] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig, "Bio-sequence database scanning on a GPU," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006. [Online]. Available: http://www.hicomb.org/papers/HICOMB2006-01.pdf

[13] M. A. Suchard and A. Rambaut, "Many-core algorithms for statistical phylogenetics," *Bioinformatics*, vol. 15, no. 11, p. 13701376, April 2009.

[14] R. Hussong, B. Gregorius, A. Tholey, and A. Hildebrandt, "Highly accelerated feature detection in proteomics data sets using modern graphics processing units," *Bioinformatics*, vol. 25, no. 15, pp. 1937–1943, 2009.

[15] M. S. Friedrichs, P. Eastman, V. Vishal, M. Houston, S. Legrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, and V. S. Pande, "Accelerating molecular dynamic simulation on graphics processing units," *Journal of Computational Chemistry*, vol. 30, no. 6, pp. 864–872, 2009.

[16] J. W. Pitera, "Current developments in and importance of high-performance computing in drug discovery," *Current Opinion in Drug Discovery & Development*, vol. 12, no. 3, pp. 388–396, 2009. [Online]. Available: http://www.biomedcentral.com/content/pdf/cd-1002727.pdf

[17] C. Boyd, "Data-parallel computing," *ACM Queue*, vol. 6, no. 2, pp. 30–39, 2008.

[18] T. P. Chen and Y.-K. Chen, "Challenges and opportunities of obtaining performance from multi-core CPUs and many-core GPUs," *Acoustics, Speech, and Signal Processing, IEEE International Conference on*, vol. 0, pp. 613–616, 2009.

[19] K. Fatahalian and M. Houston, "GPUs: A closer look," *ACM Queue*, vol. 6, no. 2, pp. 18–28, 2008.

[20] NVIDIA Coorporation, *NVIDIA CUDA Programming Guide, Version 2.3.1*, August 2009.

[21] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[22] Y. Liu, B. Schmidt, and D. L. Maskell, "MSA-CUDA: Multiple sequence alignment on graphics processing units with CUDA," in *ASAP*. IEEE, 2009, pp. 121–128.

[23] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[24] N. Bell and M. Garlandy, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Coorporation, Tech. Rep. NVR-2008-004, December 2008.

[25] F. Vázquez, E. Garzón, J. Martínez, and J. Fernández, "Accelerating sparse matrix vector product with GPUs," in *the 9th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE), Gijón, Asturias, Spain*, 2009.

[26] F. Vázquez, G. Ortega, J. Fernández, and E. Garzón, "Improving the performance of the sparse matrix vector product with GPUs," in *10th IEEE International Conference on Computer and Information Technology (CIT 2010), University of Bradford, Bradford , United Kingdom*, 2010.

[27] C. Stark, B. Breitkreutz, T. Reguly, L. Boucher, A. Breitkreutz, and M. Tyers, "Biogrid: a general repository for interaction datasets," *Nucleic Acids Research*, vol. 34, p. D535, 2006.

[28] T. S. K. e. a. Prasad, "Human protein reference database - 2009 update," *Nucleic Acids Research*, vol. 37, pp. D767–72, 2009.

[29] S. Peri, J. D. Navarro, R. Amanchy, and et al., "Development of human protein reference database as an initial platform for approaching systems biology in humans," *Genome Research*, vol. 13, pp. 2363–2371, 2003.