ABSTRACT

(l, d) planted motif problem is defined as: Given a sequence of n DNA sequences, each of length L, find M, the set of sequences(or motifs) of length l which have at-least one d-neighbor in each of the n sequences. Planted motif problem is an important and well-studied problem in computational biology. Motif finding is useful for developing methods to obtain transcription factor binding sites, sequence classification, in developing methods for building phylogenetic trees etc. The planted motif problem is difficult to solve especially for challenging instance sizes (15,5), (17,6), (19,7), and (21,8). The challenging instances are computationally intensive and require large amount of memory. Several serial implementations have been proposed for solving this problem. The time required by these methods for solving large challenge instances is prohibitively expensive. In this paper, we propose a parallel implementation on GPU that solves the challenge instance (21,8) in 1.1 hours. We are not aware of any sequential or parallel method that will solve this challenge instance in better time. Additionally, to the best our knowledge we are not aware of any previous implementation of a parallel method to solve the planted motif problem on GPU.

1. INTRODUCTION

Motif finding is an important and well-studied problem in computational biology [18] [6]. Motif finding is useful for developing methods to obtain transcription factor binding sites, sequence classification, in developing methods for building phylogenetic trees etc. Finding motif is a computationally expensive and challenging task. Many variants of motif finding problem can be found in the literature. One set of variants concentrates on finding repeated patterns in a single sequence, and the other set concentrates on finding patterns that appear in multiple sequences. The planted motif problem (PMP) falls in the second category.

An (l, d) planted motif problem can be defined as “Given a sequence of n DNA sequences, each of length L, find M, the set of sequences(or motifs) of length l which have at-least one d-neighbor in each of the n sequences”. A d-neighbor of an l-mer (sequence of length l) p is defined as an l-mer that is at a Hamming distance of d or less from p. In the rest of the paper, we refer to l as enumeration length and d as enumeration distance.

A number of approaches have been proposed to solve the motif finding problem including PMP. Some of these approaches find approximate motifs [12], [2], [14] and others find exact motifs[9], [16], [17], [9], [15], [11], [3], [13], [10], [8]. These approaches can be classified into two types: iterative approaches and combinatorial approaches. Iterative approaches like Gibbs sampling and expectation maximization are based on position weight matrices while combinatorial approaches like MITRA, WINDOWER are based on hamming distances. Planted motif problem defined in this paper is based on hamming distances.

Most approaches to solve PMP are serial in nature and are difficult to parallelize. We had recently proposed a new parallel approach to solve PMP called BitBased approach[7]. BitBased is a simple, easily parallelizable approach. It outperforms all the approaches proposed so far to solve the planted motif problem. In this paper, we show how to implement BitBased on GPU architecture. Iterative approaches like Gibbs sampling [19] and MEME [4] have been implemented on GPU while there are no combinatorial approaches implemented on GPU currently.

BitBased is an enumeration based approach to solving planted motif problem. It uses n’ bit arrays, n’ ≤ n, of size 4^l each to find the planted motifs. Each bit in the bit array corresponds to an l-mer. The key idea of BitBased is to enumerate all the l-mers in the input sequences to find their d-neighbors and set the bits corresponding to the d-neighbors in the bit arrays. It then uses the bit arrays to find the planted motifs. It can be noticed that BitBased has high memory requirement. To reduce memory requirement one can use the iterative BitBased approach at the expense of increasing the execution time. Iterative approach works by virtually partitioning the bit arrays into chunks such that a chunk fits in the available memory. We then make multiple passes of the original algorithm to find motifs. The number of passes is determined by the number of virtual partitions. A small chunk size results in increased number of virtual partitions, and thus increasing the overall time to find motifs.
GPUs are becoming increasingly popular in the world of parallel computing. GPUs, which were once used only for graphics, are now being used for different types of applications to achieve high performance. With the advent of CUDA, the task of programming for GPU has become much simpler. A GPU is a massively parallel, multi-threaded, manycore processor with hundreds of cores and huge computation power. It can execute thousands of threads concurrently. The programmer must carefully design her application to map to GPU and effectively utilize the hardware.

In this paper we parallelize the BitBased approach[7] for GPU. Though BitBased approach is easily parallelizable, it is challenging to effectively implement it on GPU. The reason being the high memory requirement. We have seen that BitBased uses bit arrays to find planted motifs and that the bit arrays are of size 4lbits each. And moreover the access to the bit arrays is very scattered. For example, to solve a (15,5) instance, BitBased needs bit arrays of size 128MB each. Such amount of memory is only available on GPU’s global memory. But global memory has very high latencies especially when the access pattern is scattered. In such cases it is highly recommended to use GPU’s shared memory. But the shared memory is too small (16KB for Tesla C1060 and S1070) to accommodate the bit arrays. So we use iterative BitBased approach and partition the bit arrays into chunks that fit in shared memory. We then optimize the approach by decreasing the register usage which increases the occupancy of the GPU. We also do reordering of shared memory to avoid bank conflicts.

We have implemented BitBased on NVidia Tesla C1060 which has one GPU device and NVidia Tesla S1070 which has four GPU devices. Tesla C1060 has 30 multi-processors with 8 streaming processor cores each while Tesla S1070 has 960 cores. We tested the (15,5), (17,7), (19,7), (21,8) challenging instances. Tesla C1060 took 8 seconds, 1.52 minutes, 19.7 minutes and 4.5 hours respectively and Tesla S1070 took 3 seconds, 23.9 seconds, 5 minutes and 69 minutes respectively. These are the best timings obtained for planted motif problem so far. We also compare with the results on multicore architecture. We found that a single GPU shows up to 13 to 14 times speed-up and 4 GPU devices shows up to 40 to 60 times speed-up compared to single core CPU.

2. THE BITBASED APPROACH

BitBased approach is a simple, easily parallelizable approach to solving PMP. It is based on exhaustive enumeration of l-mers in the input sequences. Let $S = \{S_i | 0 \leq i \leq n - 1\}$ be the set of n input sequences. An l-mer in $S_i$ starting at location $j$, $0 \leq j \leq L - l$ is represented as $S_i^j$. The set of d-neighbors of all the l-mers in $S_i$ is represented by $N_i^{l,d}$. It is easy to see that the set of planted motifs is $M = \bigcap_{j=0}^{n-1} N_i^{l,d}$. Therefore, to find the planted motifs we first need to generate the set of $N_i^{l,d}$, $0 \leq i \leq n - 1$, and then find the motifs, i.e. l-mers that are present in all $N_i^{l,d}$, $0 \leq i \leq n - 1$. The main issue here is the memory requirement. To overcome this issue consider (15,5) instance. For a 15-mer, there can be 853584 number of 5-neighbors. For a sequence of length 600, the size of $N_i^{l,d}$ is 500200224 integers which requires approximately 2GB of memory for a single sequence. To reduce the memory requirement we use bit arrays of size $4^l$. Each bit in the array corresponds to an l-mer. For example, when $l = 4$ bit 0 represents AAAA, bit 1 represents AAC, bit 255 represents TTTT assuming A=0, C=1, G=2, T=3. For (15,5) instance we now require only $4^{15}$ bits i.e. 128MB of memory for each input sequence. The memory requirement can further be reduced using the approaches mentioned in sections 2.1.1, 2.1.2 and 2.2.

2.1 The basic BitBased approach

The basic BitBased approach consists of two phases, setting bits and finding motifs. In setting bits phase, $N_i^{l,d}$, $0 \leq i \leq n - 1$, is generated. $N_i^{l,d}$ is represented using bit arrays. A bit array $B_i$ is assigned to each input sequence $S_i$, $0 \leq i \leq n - 1$. Each l-mer in sequence $S_i$ is enumerated to generate all its d-neighbors and the bits are set in the bit array $B_i$ at the indexes corresponding to the d-neighbors. The index corresponding to an l-mer can be obtained by replacing A by 0, C by 1, G by 10 and T by 11. For example the index corresponding to the 4-mer GACT is 1000111. After setting bits phase, a bit array $B_i$ has a bit set only if the l-mer corresponding to its index is present in $N_i^{l,d}$.

In finding bits phase, the equivalent to $M = \bigcap_{j=0}^{n-1} N_i^{l,d}$ is performed. We perform logical AND operation on the bit arrays to generate a single bit array which can be used to obtain the planted motifs. The final bit array $B$ is obtained by $B = B_0 \land B_1 \land \ldots \land B_{n-1}$. If a bit is set at index $j$ in $B$ only if the bit is set at index $j$ in all the bit arrays $B_i$, $0 \leq i \leq n - 1$. In other words, the l-mer corresponding to the index $j$ is present in all $N_i^{l,d}$, $0 \leq i \leq n - 1$ making the l-mer a planted motif. Therefore the planted motifs are nothing but the l-mers corresponding to the indexes in $B$ in which a bit is set.

To reduce the memory requirement further, we use two modifications to the basic approach: Increment motifs and filtering motifs. These modifications, if applicable, not only reduce the memory requirement but also improve the performance.

2.1.1 Increment Motifs

This modification is based on the observation that given the set of motifs for $(l−1, d)$ instance their d-neighbors and corresponding distances in all the n sequences, we can find the motifs for $(l, d)$ instance in $O(n)$ time. Let $p$ be a motif for $(l−1, d)$ instance. Let $(j_0, j_1, \ldots, j_{n-1})$ and $(d_0, d_1, \ldots, d_{n-1})$ be the locations of d-neighbors in n sequences and their distances respectively. We can say that $p[R; R ∈ \{A, C, G, T\}$ and ‘′ is append operation, has a d-neighbor in sequence $S_i$ if it satisfies any of the following conditions: 1. residue at location $j_i + l$ is $R$. 2. $d_i < d$. For each motif $p$ for $(l−1, d)$ instance, we find if $p[A; p[C; p[G; p[T$ is a motif for $(l, d)$ instance using the above conditions. Therefore to find $(l, d)$ motifs, we can first find $(l′, d)$ motifs and then use the above logic incrementally to find $(l, d)$ motifs. With decreasing values of $l′$, the number of $(l′, d)$ motifs increase exponentially and hence the time spent in increment motifs. Therefore the value of $l′$ must be carefully chosen.

2.1.2 Filter Motifs

Instead of setting bits and finding motifs for all n sequences, this modification first finds the motifs for n′ sequences where
n’ ≤ n. These motifs are called candidate motifs. These candidate motifs are then filtered to find the final planted motifs. This is done by checking each of the candidate motifs if it is present in all the remaining n − n’ input sequences. This modification reduces the memory requirement because we now require only n’ buffers instead of n buffers. By decreasing the value of n’, not only the space requirement decreases but also the time decreases. The reason being that the time taken by BitBased approach is dominated by setting bits phase. By reducing n’ we need to set the bits for fewer sequences and hence reducing the time taken. But if the value of n’ is chosen to be too low, then the time spent in filtering motifs increases and so the overall time. So it is important to choose an optimum value for n’.

2.2 The Iterative BitBased Approach

This is a crucial modification to the basic BitBased approach and also is the basis for implementing BitBased on GPU. As we have seen previously, BitBased has high memory requirement. It might not always be possible to satisfy such requirement. In such cases, we can use the iterative BitBased approach. Iterative BitBased approach solves the planted motif problem with much less memory requirement but at the expense of increase in time due to the increase in number of operations. Iterative approach works by reusing the available memory to accomplish the required task, which is to find planted motifs. Let \( l_{\text{max}} = \max \{ i \mid i \text{ \# bits of memory can be allocated} \}. \) We virtually partition the bit array of size 4\(^i\) into 4\(^i-l_{\text{max}}\) chunks, each chunk of size 4\(^{l_{\text{max}}}\) bits. In 4th iteration, the l-mers of input sequences are enumerated in such a way that the bits are only set in the ith chunk. After finding motifs in ith chunk the same memory is then reused for the \((i+1)\)th iteration. Note that when bit array of size 4\(^i\) is partitioned into 4\(^{i-l_{\text{max}}}\) chunks, the first \( l-l_{\text{max}}\) residues corresponding to the index to in a chunk are all the same. For example, when we partition 4\(^{17}\) bits into 16 partitions, all the 17-mers corresponding to the indexes in the chunk start with AA, second chunk starts with AC, and so on. To effectively enumerate the l-mers, we reduce the enumeration length from l to \( l_{\text{max}} \) as shown in algorithm 1. Note that the more number of chunks the bit array is partitioned into, the less is the enumeration length.

Algorithm 1 IterativeApproach

Input: \( n, l, l_{\text{max}} \)
Output: \( M’, \) the set of \((l, d)\) planted motifs

1. Let \( l_{\text{aff}} = l - l_{\text{max}} \)
2. \( M = \emptyset \)
3. for \( idx = 0 \) to \( 4^{l_{\text{aff}}} - 1 \) do
4. get the sequence \( p \) of length \( l_{\text{aff}} \) that corresponds to \( idx \)
5. setting the bits in \( id\text{x}_{\text{idx}} \) chunk
6. for \( i = 0 \) to \( n - 1 \) do
7. for \( j = 0 \) to \( L - l + 1 \) do
8. get distance \( d’ \) between \( p \) and \( S_{\text{idx}}^{l_{\text{aff}}} \{ j \} \)
9. generate \( N_{l_{\text{max}}, d-d’}^{l_{\text{aff}}} \{ j + l_{\text{aff}} \} \)
10. for each \( l_{\text{max}}\)-mer \( q \) in \( N_{l_{\text{max}}, d-d’}^{l_{\text{aff}}} \{ j + l_{\text{aff}} \} \) do
11. get index \( \text{idx}’ \) corresponding to \( q \)
12. set \( B_i[\text{idx}’] = 1 \)
13. end for
14. end for
15. end for
16. {finding motifs in \( id\text{x}_{\text{idx}} \) chunk}
17. \( B = B_0 \land B_1 \land ... \land B_{n-1} \)
18. for \( i = 0 \) to \( 4^{l_{\text{aff}}} - 1 \) do
19. if \( B[i] = 1 \) then
20. \( r \) be the \( l_{\text{max}}\)-mer corresponding to \( i \)
21. Append \( r \) to \( p \) and add the appended sequence to \( M \)
22. end if
23. end for
24. clear all the bit arrays \( B_0 \) to \( B_{n-1} \)
25. end for

3. OVERVIEW OF GPU

GPU is a massively parallel, multi-threaded, manycore processor. Each GPU device is an array of streaming multiprocessor which in turn consists of a number of scalar processor cores. GPU is capable of running thousands of threads concurrently. It is able to do so by employing SIMT (single-instruction multiple-thread) architecture. The threads are created, scheduled and executed in groups called warps. All the threads in a warp share a single instruction unit. The threads in a GPU are extremely light weight and they can be created and executed with zero scheduling overhead.

CUDA is a parallel programming model that enables programmers to develop scalable applications to be executed on GPU. It exposes a set of extension to C and C++. A CUDA program is organized into sequential host code which is executed on CPU and calls to functions called kernels which are executed on GPU. A kernel contains the device code that is executed by the GPU threads in parallel. CUDA threads can be grouped into thread blocks. Using CUDA one can define the number of blocks and the number of threads per block that can execute a kernel.

3.1 Memory organization

The device RAM is virtually and physically divided into different types of memory: global, local, constant and texture memory. Apart from device RAM the threads can also access on-chip shared memory and registers as shown in figure 1. Global memory and texture memory have highest latency compared to the other types of memory. A thread has exclusive access to its local memory. All the threads in a block can access on-chip shared memory. All the threads across all thread blocks have access to global, texture and constant memory. Constant and texture memories are read only while global is both read and write.

3.2 Performance considerations

A CUDA program should be properly designed taking advantage of the resources for better performance. Since GPU uses a SIMT architecture in which all the threads in a warp use a single instruction unit, the best results can be achieved when all the threads in a warp execute without diverging. When threads diverge they are executed serially, thus decreasing performance.

Global memory has very high latency. But by coalescing the global memory accesses, high throughput can be achieved. For example if the threads in a warp access contiguous ad-
issue is that BB has high memory requirements. As we have not straightforward to implement it on the GPU. Though BitBased is a easily parallelizable approach, it is non-parallelizable thus reducing the effective bandwidth. In order to avoid this, the programmer should try to make sure that the threads access separate addresses then 32 transactions are issued.

Shared memory is divided into equally sized blocks called banks. If two threads in a half warp access the same bank, this would result in bank conflict and the accesses are serialized thus reducing the effective bandwidth. In order to avoid this, the programmer should try to make sure that the threads access different banks.

The memory latencies can be hidden by executing other warps when a warp is paused. So to keep the hardware busy there should be enough active warps. Occupancy is the ratio of number of active warps per multi-processor to the maximum possible number of active warps. If the occupancy is too low, then the memory latency cannot be hidden resulting in performance degradation. So the programmer should try to increase the occupancy to effectively use the hardware.

4. PARALLELIZING BITBASED ON GPU

Though BitBased is a easily parallelizable approach, it is not straightforward to implement it on the GPU. The main issue is that BB has high memory requirements. As we have seen in section 2, it requires $4^l$ bits of memory for each bit array. Such high amount of memory is only available on the global memory. But global memory has a drawback of high latency. Furthermore, the access pattern of the bit arrays is very scattered making it difficult to use the coalescing feature of the global memory. So to avoid using global memory, we partition the bit arrays into smaller chunks that fit in shared memory. This is similar to the iterative approach discussed in section 2.2. The only difference is that instead of iterating, we assign the task of each iteration to a GPU thread block.

Let $t$ be the number of threads in each block. To solve $(l, d')$ instance we first find $l'$ and $n'$ as explained in [7]. Let $l_s = \max \{ 1 \mid 4^n$ bits of memory can be allocated on shared memory $\}$. The bit arrays are partitioned into chunks of $4^n$ bits of memory. Each chunk is assigned to a single block. Thus the number of blocks is $4^{l'-l_s}$. The threads in each block enumerate the $l$-mers in such a way that they generate the $d$-neighbors only in the chunk of bit arrays assigned to the block. We use the same logic as in iterative approach. Note that the enumeration length here is $l_s$.

The $t$ threads in a block are responsible for setting bits in the chunk of bit arrays assigned to the block. The $l$-mers are distributed among the $t$ threads. The consecutive $l$-mers are assigned to consecutive threads. After all the threads have finished enumerating the $l$-mers and setting bits, the threads enter the find Motifs phase. After finding the candidate motifs, they must be filtered by checking if they are present in the remaining $n - n'$ input sequences. We perform this step in a separate kernel called FilterMotifs to avoid divergence of threads. So a thread, after finding a candidate motif instead of performing the filtering phase, it writes it to the global memory so that the candidate motif can be accessed in the FilterMotifs kernel. To write on to global memory, we use a variable called glndex. When a thread finds a candidate motif, it first atomically increments glndex and then writes the candidate motif to the global memory at the index returned by the atomic operation. This is to avoid different threads in different blocks writing to the same index in global memory.

After finding the candidate motifs, filtering them is straightforward. Let $c$ be the number of candidate motifs. For the FilterMotifs kernel, we need $c/t$ blocks. The $c$ candidate motifs are equally distributed among the blocks. Within the block, the candidate motifs are further distributed among the threads. Each thread is assigned a candidate motif and it checks if the candidate motif has $d$-neighbors in the remaining $n - n'$ input sequences which were not considered during FindCandidateMotifs kernel. If a thread finds that the candidate motif is a planted motif, it writes to the global memory using the same logic explained previously. We improve this implementation by using two modifications: Bit representation and repartitioning and reordering.

4.1 Bit Representation

As we have seen in section 3, each multiprocessor has a limited number of registers. This implementation is limited by the number of registers. Since each thread consumes large number of registers, the number of threads per block is less and hence the occupancy of GPU. To improve the occupancy and performance, we need to reduce the registry usage as much as possible. Each input sequence of length $L$ has $L-l+1$ $l$-mers. If the input sequence is represented using a character array then an $l$-mer requires $l$ bytes of memory. Instead we can represent an $l$-mer using an integer, 2 bits for each residue [1] [15]. For example, the 4-mer CGGA can be represented using an integer whose binary representation is 01101000. By doing so, an $l$-mer, $l \leq 16$, would need only 4 bytes and $l \leq 32$ would need 8 bytes of memory. So we convert the input character array into an integer array, the integer at index $i$ represents the $l$-mer starting at location $i$ in the input sequences. By converting into input array, GPU threads only need to read one integer rather than $l$ bytes. This would not only reduce the registry usage by also reduce the I/O time as only an integer need to be read. We use texture binding to read the input sequences.

4.2 Repartitioning and reordering
We have seen in section 3 that the shared memory is organized into banks. Successive 32-bit words are assigned to successive banks. We implement a bit array using a 32-bit integer array. Therefore successive integers are assigned to successive banks. Each thread executing the kernel enumerates $l_s$-mers in the input sequence and may set the bits in any of the integer and therefore in any bank resulting in bank conflicts. In order to avoid bank conflicts we repartition the integer array and then reorder the integer array. The integer array, which was once partitioned to fit in the shared memory, is repartitioned into 16 chunks (as there are 16 banks in Tesla). The $i$th thread in a half warp enumerates the $l_s$-mers to set the bits in $i$th chunk. We then reorder the integer array such that the $i$th thread in a half warp would only access the integers in the $i$th bank. Therefore there will be no bank conflicts after reordering the integer array.

In addition to avoiding the bank conflicts, repartitioning and reordering has another advantage. Partitioning a bit array into chunks reduces the enumeration length. Because we partition the integer array into 16 chunks, the enumeration length reduces from $l_s$ to $l_s - 2$. Note that the maximum enumeration distance is equal to the enumeration length. For example, when $l_s = 4$, the maximum enumeration distance is 4. So the maximum enumeration distance also decreases by 2. Thus we only need to enumerate to generate $(l_s - 2)$-neighbors instead of $l_s$-neighbors. This would reduce the registry consumption of each thread and hence we can increase the number of threads per block. Having more threads per block would increase the occupancy resulting in better performance.

5. EXPERIMENTAL RESULTS

We have implemented BitBased on Nvidia Tesla C1060 and Nvidia Tesla S1070 both running at 1.3GHz. C1060 has 30

Table 1: Comparison with multicore

<table>
<thead>
<tr>
<th>GPU devices</th>
<th>time (seconds)</th>
<th>speed-up</th>
<th>time (seconds)</th>
<th>speed-up</th>
<th>time (seconds)</th>
<th>speed-up</th>
<th>time (seconds)</th>
<th>speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 core CPU</td>
<td>16 cores CPU</td>
<td>1 core CPU</td>
<td>16 cores CPU</td>
<td>1 core CPU</td>
<td>16 cores CPU</td>
<td>1 core CPU</td>
<td>16 cores CPU</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>13.5</td>
<td>1.4</td>
<td>91.2</td>
<td>13.6</td>
<td>1.6</td>
<td>19.7</td>
<td>14.3</td>
</tr>
<tr>
<td>2</td>
<td>4.4</td>
<td>24.5</td>
<td>2.5</td>
<td>46.1</td>
<td>26.8</td>
<td>3.1</td>
<td>9.9</td>
<td>28.5</td>
</tr>
<tr>
<td>3</td>
<td>3.2</td>
<td>33.6</td>
<td>3.4</td>
<td>31.1</td>
<td>39.7</td>
<td>4.6</td>
<td>6.62</td>
<td>42.6</td>
</tr>
<tr>
<td>4</td>
<td>2.7</td>
<td>40</td>
<td>4.1</td>
<td>23.9</td>
<td>51.7</td>
<td>6.0</td>
<td>5</td>
<td>56.4</td>
</tr>
</tbody>
</table>

Figure 2: (a) The integer array is partitioned into 16 chunks so that the $i$th thread in a half warp only accesses $i$th chunk. (b) The integer array is reordered such that the $i$th thread in a half warp only accesses $i$th bank.
multiprocessors with 8 scalar processor cores each. S1070 has four GPU devices with 240 cores each. We have tested our code with 20 input sequences of length 600 each. We tested it on random sequences with motifs planted at random positions in the 20 sequences. We have used \( n' = 6 \) for all our experiments. C1060 and S1070 both have a shared memory of 16KB per processor. As we have described in section 4 we need to find the value of \( l_s \) where \( l_s = \max \{ \lfloor i \div 4n' \rfloor \} \) bits of memory can be allocated on shared memory. We have found that 6 is the most suitable value for \( l_s \). Table 1 shows the performance results obtained on 1 to 4 GPUs.

We have also experimented the approach using 1 to 120 multiprocessors on Tesla S1070 with only one active block for each multiprocessor and the load is distributed equally among the multiprocessors. It can be seen from Figure 3 that the approach scales well with the number of multiprocessors. It can be seen clearly that the approach scales well with the increase in number of multiprocessors.

![Figure 3](image_url)  
**Figure 3**: Plot showing the speed-up of the approach with respect to number of multiprocessors.

We have also collected the results using different number of multiprocessors on Intel based multicore architectures. The BitBased approach was implemented on a 4 quadcore 2.67 GHz Intel Xeon X5550 machine with a total of 16 cores using 1GB memory. The basic BitBased approach was used for (15,5) and lower instances and iterative BitBased approach was used for (17,6) and higher instances. Table 1 shows the results obtained on the multicore machine. It shows the speed-up obtained on GPU with respect to 1 core CPU and 16 cores GPU. The actual results for multicore are discussed in [7]. It can be seen that a single GPU device is 13 to 14 times faster than a single core of Xeon X5550 machine. It performs better than 16 core Xeon machine. 4 GPU devices are 40 to 60 times faster than single core CPU and 4 to 6 times faster than 16 core CPU.

![Figure 4](image_url)  
**Figure 4**: Plot showing the speed-up of the approach with respect to number of GPU devices.

5.1 Comparison with multicore

The BitBased approach was implemented on a 4 quadcore 2.67 GHz Intel Xeon X5550 machine with a total of 16 cores using 1GB memory. The basic BitBased approach was used for (15,5) and lower instances and iterative BitBased approach was used for (17,6) and higher instances. Table 1 shows the results obtained on the multicore machine. It shows the speed-up obtained on GPU with respect to 1 core CPU and 16 cores GPU. The actual results for multicore are discussed in [7]. It can be seen that a single GPU device is 13 to 14 times faster than a single core of Xeon X5550 machine. It performs better than 16 core Xeon machine. 4 GPU devices are 40 to 60 times faster than single core CPU and 4 to 6 times faster than 16 core CPU.

6. CONCLUSION

We presented an efficient parallel approach for solving the planted motif problem on GPU. This approach is modification of a BitBased approach that was originally proposed for Intel based multicore architectures. The BitBased approach had to be modified for GPU architecture. The proposed implementation solves the challenge instance (21,8) of planted problem in 1.1hrs. We are not aware of any sequential or parallel method that will solve this challenge instance in better time. Additionally, to the best our knowledge we are not aware of any previous implementation of a parallel method to solve the planted motif problem on GPU.

7. REFERENCES


