2009 International Conference on Parallel and Distributed Computing, Applications and Technologies

Theoretical and Empirical Analysis of a GPU based Parallel Bayesian Optimization Algorithm

Asim Munawar, Mohamed Wahib
Grad. School of Information Science & Technology
Hokkaido University, Sapporo, JAPAN
Email: asim@uva.cims.hokudai.ac.jp

Masaharu Munetomo and Kiyoshi Akama
Information Initiative Center
Hokkaido University, Sapporo, JAPAN
Telephone: +81-11-7063760

***Abstract*—General Purpose computing over Graphical Processing Units (GPGPUs) is a huge shift of paradigm in parallel computing that promises a dramatic increase in performance. But GPGPUs also bring an unprecedented level of complexity in algorithmic design and software development. In this paper we describe the challenges and design choices involved in parallelization of Bayesian Optimization Algorithm (BOA) to solve complex combinatorial optimization problems over nVidia commodity graphics hardware using Compute Unified Device Architecture (CUDA). BOA is a well-known multivariate Estimation of Distribution Algorithm (EDA) that incorporates methods for learning Bayesian Network (BN). It then uses BN to sample new promising solutions. Our implementation is fully compatible with modern commodity GPUs and therefore we call it gBOA (BOA on GPU). In the results section, we show several numerical tests and performance measurements obtained by running gBOA over an nVidia Tesla C1060 GPU. We show that in the best case we can obtain a speedup of up to 13x.**

***Index Terms*—General Purpose computing over GPU (GPGPU); Estimation of Distribution Algorithms (EDAs);**

I. Introduction

Bayesian Optimization Algorithm (BOA)[1] belongs to a class of algorithms known as Estimation of Distribution Algorithms (EDAs)[6]. EDAs are an outgrowth of Genetic Algorithms (GAs). Conventional GAs maintain a population of probable solutions and then they apply genetic operators like selection, mutation, and crossover to find the next population. This process continues until the algorithm finds an acceptable solution. EDAs replace the crossover step of conventional GAs with variation. In variation some kind of statistical inference from the existing population is used to construct the next population. In the case of BOA, variation starts by constructing a Bayesian Network (BN) [2] as a model of promising solutions after selection. New candidate solutions are then generated by sampling the constructed Bayesian network. Finally, new solutions are incorporated into the population, eliminating some old candidate solutions, and the next iteration is executed unless a termination criterion is met. Figure 1 shows the important steps in a single iteration of BOA.

BOA employs BN to encode the problem structure. Construction of BN is a computation-intensive task and according to Jiri et al. [8] takes more than 95% of the total execution time. Parallel implementation of BOA (pBOA) is often used to reduce the total execution time. Almost all the parallel implementations of BOA suggested in the last decade were designed for clusters or MPPs. However, in this decade other architectures like multicores and Graphical Processing Units (GPUs) are competing as an affordable and energy efficient alternatives to conventional parallel computing paradigms. Such architectures are becoming more and more common and their importance cannot be ignored anymore.

GPU is a highly parallel multithreaded and manycore processor originally designed for computer graphics. With the addition of programmable stages and higher precision arithmetic, GPUs are now commonly used for applications that were traditionally handled by a CPU. The use of GPU for non-graphics applications is known as General Purpose processing using GPU or GPGPU in short. GPUs are designed such that more transistors are devoted to data processing rather than data caching and flow control, making GPUs less general purpose than CPUs. However, a carefully designed algorithm on GPU can achieve unprecedented speedups. Compute Unified Device Architecture (CUDA) [4] programming model by nVidia is very well suited to expose the parallel capabilities of GPUs through industry standard programming languages like C.

The main motivation behind this paper is to harness the computational power of modern day GPUs to reduce the total execution time for solving complex combinatorial optimization problems using BOA without compromising the output fidelity. Utilizing memory bandwidth and massive data parallelism in an efficient way are the major design constraints. Although nVidia's Tesla C1060 is capable of 933 GFLOPs/s of processing performance and has a memory bandwidth of 102 GB/s, reaching peak performance or near peak performance is not an easy task. The objective of this research is to have a GPU compatible pBOA, where the algorithm is designed to take maximum advantage of the GPU processing power and memory bandwidth. The primary contribution of this research is the modification in the conventional pBOA algorithm to make it suitable for GPU architecture. By writing this paper, we also want to encourage more research in the area of EDAs over modern parallel architectures. We believe that such architectures will become more and more common in the coming years, making similar implementations inevitable in the very near future. We have not only designed and implemented the proposed algorithm, but in the results section (Sect. V) we empirically demonstrate the speedups achieved by solving different well known optimization problems. This

978-0-7695-3914-0/09 $26.00 © 2009 IEEE
DOI 10.1109/PDCAT.2009.32

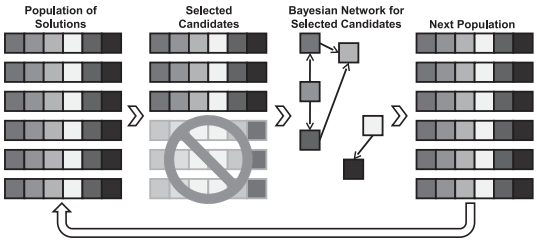

Fig. 1. Bayesian Optimization Algorithm (BOA).

paper is an extension of earlier work done by Asim et al. [7].

In order to make this paper self-contained, Sect. II describes the relevant architectural features of GPU and CUDA framework. Section III describes the BOA and pBOA algorithms in detail. Section IV explains gBOA, the proposed implementation of pBOA over nVidia GPU using CUDA. Section V shows the empirical results obtained and compares them with the results obtained over other state-of-the-art CPU architectures. Section VI concludes this paper and gives some guidelines for future work.

II. General-Purpose computing on GPU (GPGPU)

GPU is emerging as one of the most powerful parallel processing devices. GPU is especially well-suited to address problems that can be expressed as data-parallel computations with high arithmetic intensity (i.e. ratio of arithmetic operations to memory operations). Applications that process large data sets can use a data-parallel programming model to speed up the computations. In 3D rendering, large sets of pixels and vertices are mapped to parallel threads. In the recent years, many algorithms outside the field of image rendering and processing were also accelerated by data-parallel processing.

Although GPUs can offer unprecedented performance gain, implementation of an algorithm over a GPU to take full advantage of this new technology involves a significant complexity of parallelizing across the multiple cores. Memory management over a GPU makes things even more challenging. CUDA is a parallel computing architecture developed by nVidia. CUDA is the compute engine in nVidia's CUDA compatible GPUs, and is accessible to software developers through industry standard programming languages like C. CUDA is widely used for programming nVidia's GPUs for general purpose processing.

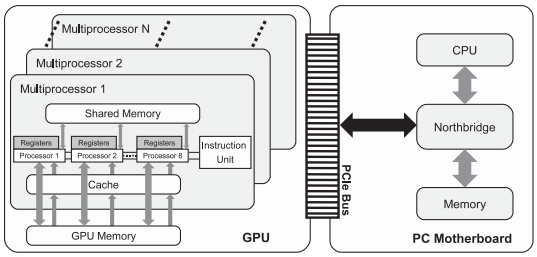

Fig. 2. Hardware architecture of GPU mounted on the PC motherboard.

Figure 2 illustrates the architecture of nVidia Tesla GPU (used for this research) [5]. The GPU runs its own specified instructions independently from the CPU but it is controlled by the CPU. A thread is the computing element in the GPU. When a GPU instruction is invoked, blocks of threads are defined to assign one thread to each data element. Arrangement of blocks and threads in a GPU is shown in Fig. 3. All threads in one block run the same instruction on one streaming MultiProcessor (MP), which gives the GPU an SIMD/SIMT architecture. Each MP includes 8 stream processor (SP) cores, an instruction unit, and on-chip memory that come in three types: registers, shared memory, and cache (constant and texture). As shown in Fig. 2, threads in each block have access to the shared memory in the MP, as well as to a global memory in the GPU. When an MP is assigned to execute one

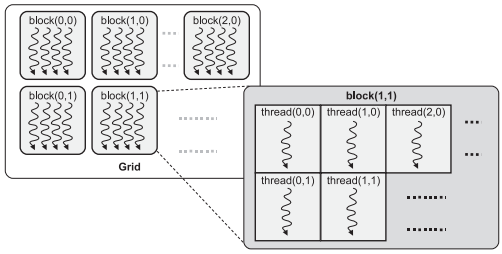

Fig. 3. Hierarchy of computing structure in a GPU.

or more thread blocks, the instruction unit splits and creates groups of parallel threads called warps. The threads in one warp are managed and processed concurrently. Unlike SIMD architecture, threads in the same warp can follow different instruction paths. However, this phenomenon drastically reduces the performance and should be minimized for an efficient implementation.

III. Past Work: Parallel BOA (pBOA)

BOA uses BN to encode the structure of a problem. In the chromosome of length n each gene is treated as a variable and is represented by a node in the dependency graph. For each variable X_i it defines a set of variables P_{X_i} it depends on, so the distribution of individuals is encoded as:

$$p(X) = \prod_{i=0}^{n-1} p(X_i|P_{X_i})$$

A directed edge from X_j to X_i in the network implies that X_j belongs to the parent nodes P_{X_i} of X_i. To reduce the space of networks, the number of incoming edges into each node is limited to k. Bayesian Dirichlet (BD) metric [3] is used to measure the quality of the network. A special case of BD metric called K2 metric is used when no prior information about the problem is available. The search for an optimal BN is an NP hard problem, so some kind of search algorithm is necessary for reaching an optimal result in a reasonable amount of time. Different algorithms can be used for building BN. A simple greedy algorithm with only one edge addition

```
1 : Start with an empty network B;
2 : Generate the permutation array perm;
3 : for i := 0 to (n - 1) do in parallel
4 :  while any edge ending in the variable Xi can be added do
5 :   for possible new edge (variable Xj having perm[j]<perm[i]) do
6 :    Compute local increase of K2 metrics after adding edge (Xj,Xi);
7 :   endfor
8 :   Add the edge(X_j,X_i) with highest improvement to the network B;
9 :  endwhile
10: endfor
```

Fig. 4. Parallel algorithm for Construction of BN [8].

in each step is commonly used for the purpose [10], [8]. The algorithm starts with an empty network B and for each edge that can be added (new edge is added only if the edge meets the limit of the incoming edges and does not introduce a cycle in the graph), it computes the BD metrics for the BN. The edge giving the highest improvement is then added to the network B. This process is repeated until no more addition is possible. After this step the variable (nodes) are ordered in a topological order and then the nodes whose parents are already determined are generated using the conditional probabilities. This is repeated until all the variables are generated.

A. Parallel Bayesian Optimization Algorithm (pBOA)

Although BOA can be parallelized in different ways, we will discuss a widely used parallel implementation proposed by Jiri et al. [8]. As 95% of the overall execution time of BOA goes into the calculation of BD metrics and BN construction, therefore, we will concentrate only on the parallelization of the BN construction step of BOA. Jiri et al. [8] suggest that BD metric is separable and can be written as a product of n factors, where the i^{th} factor expresses the influence of edges ending in the variable X_i. It is possible to use up to n processors, each processor corresponds to one variable and it examines only edges leading to this variable (it has its own local copy of parent population). As the addition of edges is parallel, so an additional mechanism is required to keep the network acyclic. Jiri et al. propose to predetermine the topological ordering of nodes in advance. At the beginning of each generation, the random permutation of numbers $\{0, 1, ..., n-1\}$ is created. Each processor generates the same permutation. The direction of all edges in the network should be consistent with the ordering, so the addition of an edge from X_j to X_i is allowed only if $perm[j] < perm[i]$. The variable X_i with $perm[i] = 0$ has no predecessor and is forced to be independent. To compensate this phenomenon a new permutation needs to be used for each generation. The algorithm for parallel computation of BD is given in Fig. 4.

B. Related Work

Jiri et al. [8] and other pBOA algorithms in literature were designed either for clusters or shared memory multicore processors. GPU, on the other hand, is a completely different environment, and therefore, using a conventional pBOA algorithm over a GPU will seriously underutilize the resources. An algorithm designed for such an environment must be able to exploit the memory hierarchy and SIMD/SIMT features of a GPU. To the best of author's knowledge, there is no significant research done for implementation of BOA over commodity GPUs.

A simple implementation of pBOA would assign BD metric calculation of the i^{th} variable to the i^{th} thread of the GPU. Therefore, the total number of threads will be equal to the total number of variables. Although this may work for the shared memory multicore architectures, but in case of GPUs this implementation will not be able to use coalesced reads from global memory of the device and therefore will create some serious memory bottle-necks. Moreover, it will not be able to use the SIMD features of GPU as each thread in a warp will follow a unique instruction path. Also the temporary memory required by each thread will not fit in the shared memory of the thread block and this will further reduce the performance.

IV. Proposed Implementation (gBOA)

We propose a GPU compatible parallel BOA (BOA on GPU *or* gBOA). In order to avoid the problems discussed above, we will have to further breakdown the calculation of BD metrics. While designing an algorithm for GPU, we must keep the following facts about the CUDA and GPU architecture into consideration:

1) GPU is good at dealing with 1000's of threads at the same time with each thread doing a very small part of the overall job.
2) Use of shared memory and registers wherever possible.
3) Coalescing global memory access wherever possible.
4) Avoiding braches in the Threads of the same warp.
5) Try to maximize occupancy. Occupancy can be maximized by choosing an appropriate value thread block size, shared memory size and number of registers used.

In the proposed approach, we divide the BD metric calculations into smaller problems in two distinct steps. In the first step we break down the BD metrics calculation in N parts. Where, N is the total number of variables and the i^{th} part

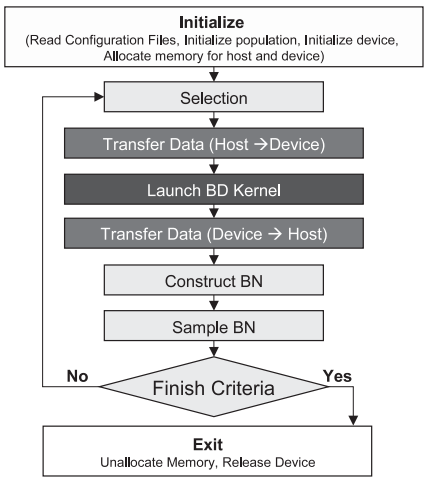

Fig. 5. Host side controller of the proposed implementation.

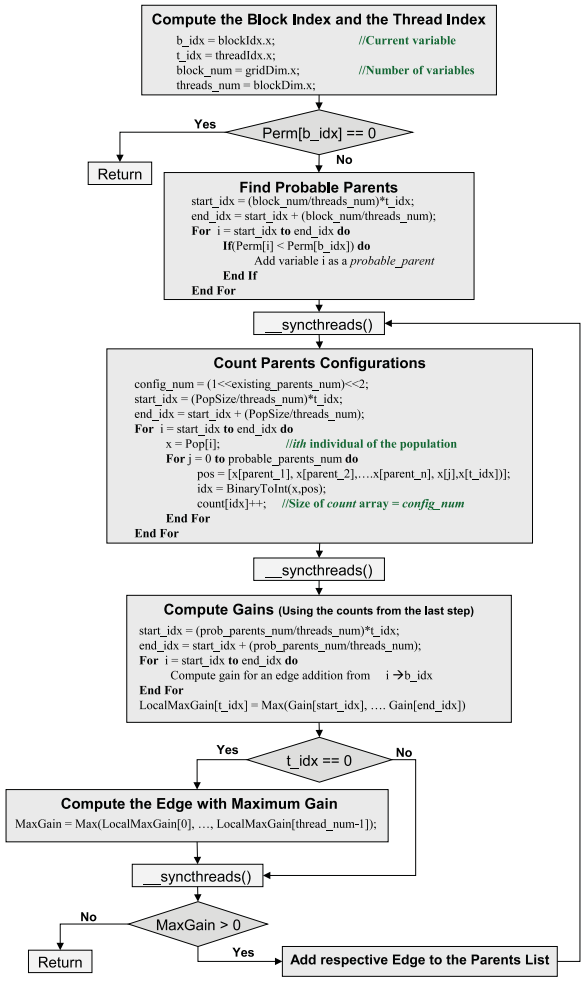

Fig. 6. Bayesian Dirichlet (BD) Kernel.

represents the calculations for edges ending in the i^{th} variable. Each division of this step is assigned to one thread block of CUDA. The first step is similar to the parallelization suggested by Jiri et al. [8]. In the second step, we further break the computation for each block into smaller tasks. Each of these tasks is then assigned to a single thread inside that thread block. Therefore, in our implementation the block number represents the variable (node) number that it processes, and within each block there are a number of threads that solve a very small part of the overall computation. Overall flow of the implementation is shown in Fig. 5. It is clear from the figure that *BD kernel* is launched once for each generation. BD kernel shown in Fig. 6, lies at the heart of gBOA. Therefore, the next section expounds on each part of the BD kernel in detail. Compliance to the SIMD architecture of GPU and the efficiency of memory usage is also discussed in detail.

A. Bayesian Dirichlet (BD) kernel

As shown in Fig. 5, BD kernel is called once for each generation. Before the kernel is deployed the population is transferred from the host memory to global memory of the device. The array containing the permutations is also transferred to the device. The maximum allowed number of incoming edges is also passed as an argument to the kernel. The output of the kernel for the i^{th} block is an array of the nodes that will become parents of the variable X_i in the final BN graph. This data is then transferred from device to host and is used to formulate the complete BN. As discussed earlier, the total number of blocks should be equal to the total number of variables as each block deals with only one variable at one time. In order to fill the pipeline of the GPU processing elements the number of threads per block is set to the warp size supported by the GPU. Different sections of the *BD kernel* can be listed as follows:

1) *Computing the Indexes*: Block and thread indexes can easily be computed using the inbuilt variables of CUDA. As each block deals with only one variable therefore block index (b_idx) is the same as the variable number. The total number of blocks ($block\_num$) gives the total number of variables in the problem. Thread index ($thread\_idx$) gives the index of the current working thread within the block, while thread number ($thread\_num$) gives the maximum number of threads available per block. These threads work together to compute the section of the BN graph which deals with the edges ending in variable number b_idx.
   After this step, thread with $t\_idx = 0$ checks if the permutation value of the current node (b_idx) is 0. If the condition is true the block exits the execution. As soon as this block of threads finish execution the next thread block is assigned to this multiprocessor automatically by CUDA.
2) *Finding probable parents*: finds the variables that can become potential parent of current variable. This means the variables that can have an edge ending in b_idx. This is done by locating all the variables that have permutation value less than the permutation value of b_idx. Each thread checks its share of the variables as shown in Fig. 6. The data structure to store the probable parents resides inside the shared memory. While the access to the permutation array, which resides in the global memory, is coalesced. This step is SIMD compatible with only one conditional statement that may cause the threads to diverge for a single instruction.
3) *Count parent configurations*: This is the most time consuming task in the BD metric calculation. In this step the algorithm needs to go through the whole population and count the binary patterns at specific locations of each individual. If the number of existing parents for a node is P_e, then the total number of possible configurations can be $config\_num = 2^{P_e+2}$. The additive 2 is to take care of the location of current variable and the location of the probable parent. This step is fully SIMD compatible as each thread in a block will follow the same instruction pattern. In ideal case all the memory

Problem Size	Avg Execution Time (secs) + Standard Deviation			Speedups	
	sBOA	pBOA	gBOA	sBOA/gBOA	pBOA/gBOA
32	10.8 ± 1.3	1.9 ± 0.4	1.5 ± 0.1	7.20	1.27
64	26.1 ± 2.1	5.5 ± 0.9	2.89 ± 0.4	9.03	1.90
96	49.7 ± 4.3	10.8 ± 1.2	5.01 ± 0.7	9.92	2.15
128	85 ± 5.8	17.9 ± 2.2	7.8 ± 0.9	10.80	2.29
160	139.3 ± 9.0	27.8 ± 3.7	11.1 ± 1.3	12.50	2.50
192	206.6 ± 14.3	39.3 ± 5.0	15.58 ± 1.9	13.26	2.52
224	275.6 ± 17.6	56.8 ± 5.8	21.88 ± 2.7	12.60	2.59

TABLE I
GBOA RUNNING OVER NVIDIA C1060 GPU VS. SERIAL IMPLEMENTATIONS OF SBOA [10] & PBOA [8] OVER INTEL I7.

reads to the population structure (residing in global device memory) can be coalesced. Moreover, $count$ data structure that stores the counting results resides in the shared memory. All the threads are synchronized after this step.

4) *Compute gains*: uses the $count$ data structures from the last step to calculate the $gain$ achieved by addition of an edge from a probable parent to the current variable. Again the task is divided among the threads available in the block. Time consuming section of this step is SIMD compatible. This step is memory efficient as it does not access the global memory of the device.
5) *Compute the MaxGain*: This step finds the edge that gives the maximum gain. Only the thread with $t\_idx = 0$ iterates through an array of $LocalMaxGain$ which is of size $thread\_num$. Other threads do nothing. This step does not access the global memory and is SIMD compatible as threads with $t\_idx \neq 0$ do not execute any instruction. If there is an improvement in the $MaxGain$ then the respective probable parent is shifted to parents list and the loop continues. On the other hand, if $MaxGain = 0$ the processing stops. We also need to check that the number of parents does not exceed the maximum number of allowed incoming edges.

As discussed above the proposed implementation for gBOA is efficient in memory usage and it exploits the SIMD capability of the device. The algorithm is robust and extendable. Given a huge global memory, gBOA can solve problems with up to 2^{16} variables (maximum allowed number of blocks in CUDA).

V. RESULTS

Results given in this section were collected over a system with nVidia Tesla C1060 GPU mounted on a motherboard with Intel®Core™i7 920@ 2.67GHz as the host CPU. C1060 has 4GB of device memory, 30 streaming MultiProcessors (MPs), and the total number of processing cores is 240. The maximum amount of shared memory per block is 16KB and clock rate is 1.30GHz. The compute capability of the device is 1.3. We are using Fedora Core 8 as the operating system and CUDA SDK/Toolkit ver. 2.1 with nVidia driver ver. 180.22. Other tools used for optimization and profiling include *CudaVisualProfiler ver. 1.1* and *CudaOccupancyCalculator*. C1060 is dedicated to computations only. The system has a separate GeForce 8400 GS GPU acting as a display card.

For making the kernel call we keep the number of blocks equal to the number of variables in the problem. Number of threads/block is kept constant at 480 throughout the experiments. The maximum number of incoming edges is also kept constant at 3. We keep the population size constant at 4800 individuals and in each generation 50% of the good individuals are selected for BN construction. We have used sum of 5-Bit trap functions as the objective function. Sum of trap functions are considered to be a GA difficult problem and is widely used for benchmarking purpose. The maximum number of generations is set to 200. The execution stops only when the maximum number of generations is reached. In all the implementations we are using the compiler directives to maximize the execution speed.

In the current implementations we have used the shared memory to store the temporary data structures for each thread block. These data structures are used to store the list of nodes with lower permutation value than the current node ($List_{DS}$), the gain for adding an edge ($Gain_{DS}$) and the data structure for counting different patterns in the population ($Count_{DS}$). Shared memory is 100-150 times faster than the global memory but is limited to 16KB/thread block. This limits the size of the problems that can be solved using the proposed implementation. Global device memory can be used to store these data structures for larger problems.

A. Theoretical Analysis

Complexity to construct the Bayesian network using the greedy search driven by BD metric is $O(k2^k n^2 N + kn^3)$, where n is the length of a chromosome, k is the limit of incoming edges into each node and N is the size of parent population [8]. Complexity of pBOA for using m processors ($m \leq n$) is $O(\lceil n/m \rceil (k2^k nN + kn^2))$ [8]. Theoretical complexity for the proposed implementation can be given by $O(\lceil n/m \rceil (k2^k n \lceil N/t \rceil + \lceil k/t \rceil n^2))$, where m is the total

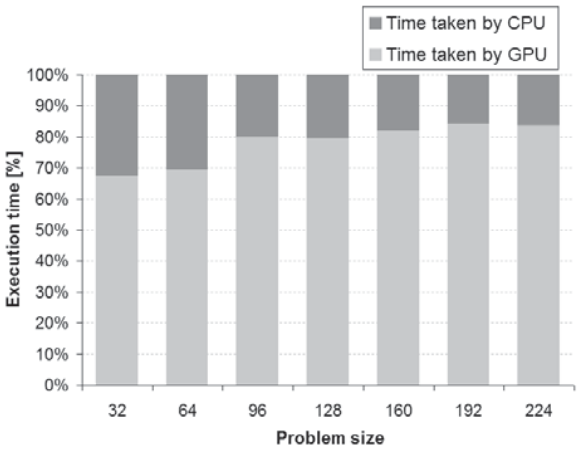

Fig. 7. gBOA time complexity profile.

number of MPs ($m \leq n$) in the GPU and t is the total number of threads/blocks. Therefore, theoretically the proposed implementation should get a speedup equivalent to the total number of threads in all the blocks.

B. Empirical Analysis

Table I compares the execution time for gBOA with time taken by serial implementation of simple BOA [10], [9] (shown by sBOA in the table) and parallel BOA [8] (shown by pBOA in the table) over Intel Core i7 920@2.67GHz processor with 4 GB of memory. The difference between pBOA and sBOA is that the pBOA uses an array of permutations to avoid cycles in BN [8]. This is the same mechanism that we employed in gBOA to avoid cycles. The results shown in this table are an average of 20 independent runs. The table clearly shows that we can achieve speedups of up to 13 times if compared to sBOA and 2.6 times if compared to pBOA.

Figure 7 shows the time taken by the BN construction (GPU time) and rest of the time (CPU time). Time taken by the GPU after parallelization over GPU still comprise more than 80% of the overall execution time (70% for smaller problem sizes). The remaining 20% of the time is the time taken by the serial part of the code and it runs over the host CPU. As mentioned earlier that this time can be reduced further by parallelizing the selection and BN sampling part. Although not included in the scope of the paper, this can be taken up as a future research to further improve the speedups over modern GPUs.

VI. Conclusions & Future work

BOA is an EDA that is widely used to solve complex global optimization problems. Construction of BN in BOA is an NP hard search problem. Therefore, BOA comprises of a search problem within a search problem. This makes BOA computationally very intensive. Over the time several parallel implementations for BOA were proposed, but all of them were designed with clusters or shared memory architectures in mind. Unfortunately, no work is done on designing a parallel BOA for SIMD hardware. In this decade other architectures like GPUs are becoming increasingly famous for general purpose parallel processing. Frameworks like CUDA are used to take the maximum advantage of the low lying hardware using an industry standard programming language. Even though different implementations of parallel BOA exists in literature, none of them are compatible with the architecture of modern GPUs. For an efficient implementation, the algorithm needs to be redesigned with a considerable care.

In this paper, we have proposed a GPU compatible approach to parallel BOA. We call it BOA on GPU or gBOA. gBOA takes maximum advantage of the SIMD architecture of the GPU. Moreover, the global memory access is coalesced and shared memory is used wherever possible in an attempt to reduce the overall execution time. In the results section, we show that significant speedups can be achieved by implementing gBOA over a modern commodity GPU.

In future we would like to parallelize other parts of BOA including selection and BN sampling over the GPU. This will further improve the overall speedup. Testing other variants of BOA over GPU will be another interesting area of research. In this paper, we have used a GPU only implementation for gBOA. Although it is not in the scope of this paper but a better approach would be to use the CPU and GPU resources in parallel to achieve an even better speedup.

Acknowledgment

We would like to thank Jiri Ocenasek for his help regarding the parallel Bayesian Optimization Algorithms. We would also like to thank Martin Pelikan for offering the simple BOA code for free [9]. We are also grateful to the anonymous users on CUDA forums who replied to our queries regarding CUDA and GPGPU.

References

[1] Erick Cantu-Paz, Martin Pelikan, and David E. Goldberg. Linkage problem, distribution estimation, and bayesian networks. Technical report, Evolutionary Computation, 1998.

[2] David Heckerman. A tutorial on learning with bayesian networks. Technical report, Learning in Graphical Models, 1996.

[3] David Heckerman, Dan Geiger, and David Chickering. Learning bayesian networks: The combination of knowledge and statistical data. In *Machine Learning*, pages 197–243, 1994.

[4] http://www.nvidia.com/cuda/.

[5] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.

[6] Heinz Mhlenbein, Thilo Mahnig, and Alberto Ochoa Rodriguez. Schemata, distributions and graphical models in evolutionary optimization. *Journal of Heuristics*, 5:215–247, 1999.

[7] Asim Munawar, Mohamed Wahib, Masaharu Munetomo, and Kiyoshi Akama. gboa: Parallel bayesian optimization algorithm over nvidia gpu using cuda. In *Super Computing 2009 (SC'09) Poster Session (to appear)*, Portland, OR, November 2009.

[8] Jiri Ocenasek and Josef Schwarz. The parallel bayesian optimization algorithm. In *Proceedings of the European Symposium on Computational Inteligence*, pages 61–67. Springer Verlag, 2000.

[9] Martin Pelikan. A simple implementation of the bayesian optimization algorithm (boa) in c++ (version 1.0), 1999. http://www.cs.umsl.edu/pelikan/boa.html.

[10] Martin Pelikan, David E. Goldberg, and Erick Cantu-Paz. Boa: The bayesian optimization algorithm. pages 525–532. Morgan Kaufmann, 1999.