Accelerating Bayesian Computation with Parallel Reduction using CUDA

Thanakij Pechprasarn and Noppadon Khiripet
Knowledge Elicitation and Archiving Laboratory
National Electronics and Computer Technology Center
Pathumthani, THAILAND
thanakij.pechprasarn@nectec.or.th, noppadon.khiripet@nectec.or.th

***Abstract*—Machine learning community is often interested in determining the best hypothesis given observed training data. Bayes' theorem provides a way to calculate the probability of such the hypothesis. In addition, Monte Carlo integration is often needed to help approximate the posterior distributions required for the Bayesian analysis. However, Monte Carlo integration takes time to compute due to a huge number of random samples required for acceptable accuracy, especially in high dimensional problem space. We propose a CUDA approach using graphic processing units to accelerate the computation by orders of magnitude with special care taken on the floating-point errors encountered in the parallel reduction step.**

Keywords—Bayesian probability, Monte Carlo integration, parallel reduction, CUDA

I. INTRODUCTION

In the area of machine learning, Bayesian method has been used by experts in the field to construct intelligent learning systems [1]. The posterior distributions are the essential parts of determining the probability of the hypothesis of the system. However, the posteriors may not be easily calculated because often they are in a form of integrals. It is extremely difficult, if not impossible, to calculate the value of the posteriors due to lacking of a closed form [2,3]. Typically, approximation methods are used to find the values of the integrals [2]. Monte Carlo integration is one of the approximation methods. It involves in a random process which will generate random samples according to the target population.

After sampling, the approximate value can be obtained via the calculation of a sample mean multiplied by an integration volume. It is trivial to implement a sequential program to calculate such the sample mean. However, the naïve implementation leads to two major drawbacks: 1) the result may not be always correct 2) the performance of the program is relatively slow. Firstly, a naïve implementation may not always yield an acceptable result [4] because of a subtle problem called absorption. The problem is a kind of round-off errors in floating-point numbers. Secondly, although the algorithm of the sequential code has a linear scale, it is considered ineffective, especially when the sample space is quite large [5].

To reduce errors from the absorption problem, the Kahan summation algorithm can be used [5]. Next, there is a known pattern to accelerate the computation of finding a sample mean called parallel reduction [6]. Fortunately, in addition to the performance improvement, the absorption problem is also less likely to occur in parallel reduction depending on the sample numbers. Although parallel reduction can be combined with the Kahan summation, we did not include the Kahan summation in our implementation. The program with the Kahan summation would have some performance decrease due to extra instructions inserted. Instead, we present a sophisticated technique that can be combined with parallel reduction to prevent the absorption problem.

A data parallel style like parallel reduction indicates that a general purpose graphical processing unit (GPGPU) can be used to speed up the computation. In order to do GPGPU, compute unified device architecture (CUDA) is used because it is the major leading programming framework at the present time [7]. In this paper, we propose a solution to accelerate the computation of finding a sample mean with parallel reduction using CUDA.

This paper is organized as follows. In Section II, we review the required backgrounds. Our proposed method including design and implementation is described in Section III. We evaluate our solution through experiments in Section IV. Finally, conclusion and future work are discussed in Section V.

II. BACKGROUND

A. *Bayesian Probability*

Bayes rule is defined as:

$$P(H|D) = \frac{P(D|H)P(H)}{P(D)} \quad (1)$$

where,

D = Observed data
H = Hypothesis
P(D|H) = Likelihood of *H*
P(H) = Prior probability of *H*
P(H|D) = Posterior of *H* given *D*
P(D) = Probability of *D*

Since it is difficult to determine *P(D)*, in practice, one often shows the relationship of (1) as:

$$P(H|D) \propto P(D|H)P(H) \quad (2)$$

Because the posterior is also a probability distribution, (2) can be divided by some normalizing factor to ensure that the whole space of the posterior is 1. Hence, the equation becomes:

$$P(H|D) = \frac{P(D|H)P(H)}{\int P(D|H)P(H)\,dH} \quad (3)$$

Alternatively, in (4) the posterior is marginalized and called the marginal posterior.

$$P(H|D) = \int P(H|D,\theta)\,d\theta \quad (4)$$

B. Monte Carlo Integration

Monte Carlo integration is a method to approximate the value of an integral and is defined as:

$$\int f\,dV \approx \mathrm{V} < f > = \frac{V}{N}\sum_{i=1}^{N} f(x_i) \quad (5)$$

where,

V = Integration volume
<*f*> = A sample mean

Monte Carlo integration utilizes a sampling method to generate *N* samples corresponding to the target population *f*. Markov Chain Monte Carlo (MCMC) is one of the sampling techniques that can be used to obtain samples with acceptable quality. A sample mean is an arithmetic mean of the samples. The approximate value is then calculated by the sample mean multiplied by the integration volume.

Monte Carlo integration can help approximate the value of the posterior through the normalizing factor in (3) or the marginalized posterior in (4).

C. Absorption Problem and The Kahan Summation

An absorption problem is a kind of round-off errors in floating-point numbers. The error will occur when we try to add two positive floating-point numbers that differ greatly. For example, instead of getting 1.23456709876543 from adding 1.234567 and 0.00000009876543, we get 1.234567. When finding a sample mean, this error can accumulate and lead to surprisingly incorrect final result in the end. The Kahan summation is an algorithm to significantly reduce the error. The idea is that the algorithm tries to keep another accumulate variable to compensate the error.

D. Parallel Reduction

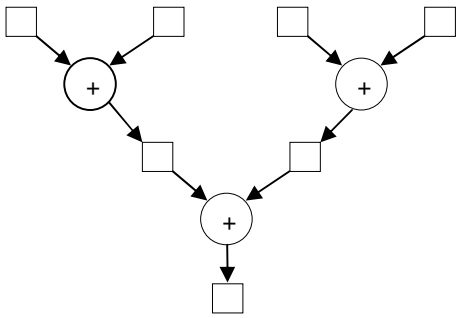

FIG 1. TREE-BASED STRUCTURE OF PARALLEL REDUCTION

Parallel reduction is a pattern for reducing a set of numbers using tree-based approach as shown in Fig 1. Let *N* be the number of samples. With parallel reduction, a sample mean is calculated as follows.

$$mean = \frac{parallel\_reduction(samples)}{N} \quad (6)$$

Assuming that *N* is a power of two, there are $log_2 N$ tree levels. Initially the first level starts with all *N* numbers. There are *N/2* add operations being performed at this level. After that, we move to the next level. Now we have *N/2* numbers with *N/4* add operations. This process repeats itself until there is only one number left. With parallel programming, the actual computation of the add operations at each level can be done in parallel.

E. GPGPU and CUDA

There is an idea to bring computing power of GPU, which is used to solving computer graphic problems, into a more general-purpose computing device [6]. This idea coined the term GPGPU. CUDA is one of the major leading software toolkits for programming GPGPU.

In CUDA, a problem will be divided into sub-problems that can be solved independently. Each sub-problem is then handled by a group of threads called a block. All threads in the same block will share some information and cooperatively work together. Because blocks can be executed in parallel, any CUDA program would automatically scale up by simply running more blocks.

A kernel is a function that utilizes the computing power from GPU. A kernel resides in the GPU side *(device)* waiting to be invoked by the CPU side *(host)*. To define a kernel, *__global__* qualifier is applied at the definition of the function. A kernel will be called by the host via a kernel launch configuration *<<<gridDim, blockDim>>>*. *gridDim* and *blockDim* are required parameters for specifying numbers of blocks and threads consecutively. *__device__* qualifier is used to define a function that resides in the device and also be invoked within the device.

III. OUR METHOD

A. Primitive Concept

Basically, the absorption problem is caused by round-off errors after adding a big and small floating-point number together. An add operation will result in an incorrect answer. This problem can be found very often in a sequential program used for finding a sum of floating-point numbers because there is only one accumulator throughout the program. After adding for several times, the accumulator would become a big number compared to the other numbers. On the other hand, with parallel reduction, the problem is less likely to occur because there are many accumulators. At each tree level, the number of accumulators is equal to the number of additions.

We assume that all the numbers are positive. In addition, we also assume that adding the maximum and minimum does not cause around-off error otherwise it may not be an interesting case. Hence, the error can occur only when an addend at some point in the parallel reduction step becomes greater than the maximum enough so that it can cause the error later when adding with a small number. Therefore, under certain sets of numbers, the error can be found even parallel reduction is used. For instance, we try to apply parallel reduction on the following numbers: 0.005, 0.005, 0.00000001 and 0.00000001. After the first level of reduction, the addends are 0.01 and 0.00000002. Next, this time the error occurs after adding 0.01 and 0.00000002. The result is 0.01 instead of 0.01000002.

Notice that from the example above if the maximum and minimum are added together, there is no error produced. That is, if we can ensure that at any stage any addend is between the values of the maximum and minimum, the error cannot occur. Next, we develop a technique that every time after adding any two numbers, we will divide the intermediate result by two. This technique will ensure that the addend is always between the two operands. To be precise, in our case, the addend is always in the middle of the two operands. As a consequence, at any time any addend computed will never become greater than the maximum. Therefore, using our technique, the absorption problem will be prevented.

When combining with tree-based structure of parallel reduction, the technique gains some benefit if N is a power of two. The sum from the reduction is not necessary to be divided by N as it is already covered in the reduction step. At each tree level, a division by two is applied to all add operations in that level. The total value of divisions by two in parallel reduction is equal to 2^{height} which is equal to N. Therefore, if N is a power of two, a sample mean can be calculated as follows:

$$mean = parallel\_reduction\_with\_div2(samples) \quad (7)$$

To handle the case that N is a non-power of two, a simple zero-padding approach is used. The sample size is extended to be the next higher power of two. Then we can repeat the parallel reduction with the division step mentioned above. However, because now the denominator N does not corresponding to the number of the divisions applied in the parallel reduction step. Therefore, a correcting term must be calculated to compensate the effect of the divisions. Eventually, we present a general form of our solution as:

$$mean = C * parallel\_reduction\_with\_div2(samples) \quad (8)$$

where,

$$C = \begin{cases} 1 & (if\ N\ is\ a\ power\ of\ two) \\ \frac{higher\ number\ that\ is\ power\ of\ two}{N} & (else) \end{cases}$$

C is equal to 1 if N is a power of two otherwise C is equal to the next higher power-of-two number from N. Note that when finding the next higher number that is a power of two, there can be many numbers, but the nearest number to N is used.

Assuming that N is not a power of two, the maximum value of C can be close to the limit of 2. However, the value of C cannot be 2 otherwise N would be a power of two which is a contradiction to the assumption. Therefore, the range of C is $[1,2)$.

B. Design and Implementation

There are two kernels in our CUDA program. Our first kernel, *solve*, is for the parallel reduction step. Let N be the number of samples and B be the block size. We have N/B blocks with each block of size B. Each sub-problem is handled by a block. Each block will be processed independently running reduction on different chunks of data. Hence, the kernel launch configuration for *solve* becomes <<<*N/B*, *B*>>>. To simplify the implementation, B is chosen to a power of two. Therefore, all sub-problems are ensured to be full blocks each of size B. We will cover the case that N is not divisible by B later. For now, we just assume that N is divisible by B.

After solving all sub-problems, we now have N/B partial results. Next, another kernel, *compact*, is used to gather all N/B dispersed results and form new data of size N/B. The new data will be divided into N/B^2 sub-problems and re-submitted to the *solve* kernel. This process repeats itself until there is only one partial result left returned from *solve*. Notice that assuming that N is divisible by B is already not enough. In this case, N has to be a power of B. Again, the general case that N is an arbitrary value will be covered later.

Eventually, *parallel_reduction_div2* is defined in Fig 2.

Algorithm parallel_reduction_div2(*samples*,*N*)
Input: *samples*—data from sampling
N—the number of samples
Output: *result*—a result from parallel reduction
L = log(*N*)/log(*B*);
LAST = *L*-1;
n = *N*;
for *i* = 0 .. *LAST*
solve<<<*n*/*B*, *B*>>>(*samples*,*n*);

```
    if i == LAST then break;
    n = n/B;
    compact<<<n, 1>>>(samples,n,B);
    cudaThreadSynchronize();
  end for
  cudaMemcpy(&result,samples);
  return result
```

FIG 2. PARALLEL REDUCTION WITH DIVISIONS BY TWO

Note that *cudaThreadSynchronize()* is needed to ensure that the *solve* kernel will never be executed until the *compact* kernel finishes its execution.

In the *solve* function, all *B* threads in each block will calculate an addend divided by two in parallel. At each level of the tree, half of the memory locations from the previous level are going to be fetched again. Therefore, using shared memory would result in a significant performance improvement. The structure of the *solve* function is shown in Fig 3.

```
Algorithm __global__ solve(samples,N)
  Input: samples—data from sampling
         N—the number of samples
  __shared__ s_data[B];
  load_into_shared_mem(s_data,samples);
  reduce(s_data,B);
  save_to_global_mem(samples,s_data);
```

FIG 3. THE SOLVE KERNEL

The shared memory is associated with *s_data* variable. Firstly, in *load_into_shared_mem*, each worker thread simultaneously copies an element of array from global memory to shared memory. Next, the *reduce* function will perform the parallel reduction step on *s_data*. After that, *save_to_global_mem* is used to copy back an answer from shared memory to global memory.

According to Harris [8], there are two major memory-addressing schemes: interleaved addressing and sequential addressing. Because the sequential addressing version results in conflict-free style of memory access, we expect that in general the sequential addressing implementation would perform better. Next, we have *reduce_i* for interleaved addressing version and *reduce_s* for sequential addressing.

Fig 4 and 5 shows *reduce_i* and *reduce_s* respectively.

```
Algorithm __device__ reduce_i(s_data,N)
  Input: s_data—data in shared memory
         N—the size of shared memory
  id = 2*threadIdx.x;
  for(s = 1; s < N; s *= 2)
    __syncthreads();
    idx = s*id;
    if idx < N then
      s_data[id] = (s_data[id]+s_data[id+s])/2;
  end for
```

FIG 4. REDUCTION FUNCTION WITH INTERLEAVED ADDRESSING

```
Algorithm __device__ reduce_s(s_data,N)
  Input: s_data—data in shared memory
         N—the size of shared memory
  id = threadIdx.x;
  for (s = N/2; s > 0; s /= 2)
    __syncthreads();
    If id < s then
      s_data[idx] = (s_data[idx]+s_data[idx+s])/2;
  end for
```

FIG 5. REDUCTION FUNCTION WITH SEQUENTIAL ADDRESSING

In current implementation so far, even in the first level of the tree half of the threads are wasteful. To utilize the worker threads, a number of threads are halved. This way in the first iteration, all threads will be performing addition. Consequently, the kernel launch configuration for *solve* becomes *<<<N/B, B/2>>>*. For clarity, the current implementation of *load_into_shared_mem* is displayed in Fig 6.

```
Algorithm __device__
load_into_shared_mem(s_data,samples)
  Input: samples—data from sampling
  Output: s_data—data in shared memory
  id = 2*threadIdx.x;
  gid = blockIdx.x*B+id;
  s_data[id] = samples[gid];
  s_data[id+1] = samples[gid+1];
```

FIG 6. LOAD INTO SHARED MEMORY WITH HALF THREADS

Next, we employ a technique from [8] called "First Add During Load." This technique will try to further half a number of threads needed by performing an initial addition during load into shared memory. Now, the kernel launch configuration is *<<<N/B, B/4>>>* and Fig 7 displays the new *load_into_shared_mem* function.

```
Algorithm __device__
load_into_shared_mem(s_data,samples)
  Input: samples—data from sampling
  Output: s_data—data in shared memory
  id = 2*threadIdx.x;
  gid = blockIdx.x*B + id;
  s_data[id] = (samples[gid]+samples[gid+B/2])/2;
  s_data[id+1] =
(samples[gid+1]+samples[gid+1+B/2])/2;
```

FIG 7. LOAD INTO SHARED MEMORY WITH FIRST ADD DURING LOAD

C. *Non-Full Block Handling*

In case of the value of *N* is not a power of *B*. This case would lead to a remainder from *N/B*. Basic zero-padding approach is used to extend a non-full block to virtually become a full block. Instead of extending to a full block of size *B*, we can save some memory space and computation time by using a smaller block size. For example, *next_higher_power_of_two(remainder)* can also be used as our block size. However, we may not freely choose any a power of two to be our block size for a non-full block. A non-full block can be caused from either because it is the last non-full block (with many other full blocks) or because

this is the only single block left (no any other block). The two situations have to be handled differently. In case of the last non-full block, we have to use *B* as our block size regardless of the value of the remainder. If we choose a number that is less than *B* to be the block size, the number of the divisions would also be less than the number of divisions from the other full blocks. In case of only single block left, *next_higher_power_of_two(remainder)* must be used. *B* may not be used as the block size because using too large block size than necessary implies that we have extra divisions and again would lead to an incorrect result.

We can freely choose any a power of two to be the block size for non-full block if we are not using divisions by two. The summation alone would not affect the computation result.

IV. EXPERIMENT AND RESULTS

To set up the experiment, a machine with NVidia GeForce GTX 280 graphic card is used. The CUDA Toolkit version 2.3 is installed. Hence, the compute capability is 1.3.

Typically, the block size *B* is equal to *blockDim*, and the maximum value of *blockDim* from the compute capability is 512. The minimum value of *B* is selected to be 32 because 32 is also the warp size. It would not be advantageous to have the number of threads less than the warp size. In addition, because *B* is also enforced to be a power of two, we can have all possible values of *B* calculated. The value of *B* can be 32, 64, 128, 256 and 512.

Next, because *blockDim* is referring to the number of threads, it may or may not be equal to the block size depending on the number of threads per block. However, in the *solve* kernel, only *B/4* threads are used per block. That is, we can increase the block size up to 4 times so all eligible block sizes become 128, 256, 512, 1024 and 2048.

With the compute capability of 1.3, *gridDim* is 65535. Because we know that *gridDim* is calculated from *N/B*, we can anticipate the maximum problem size as follows.

TABLE 1
MAXIMUM PROBLEM SIZE CORRESPONDING TO THE BLOCK SIZE

Block Size	Maximum Problem Size
128	8388480
256	16776960
512	33553920
1024	67107840
2048	134215680

A. Correctness

We generate samples for the experiment using two random number generators. The first generator is a uniform random number generator which will generate the data set within a range of 0 and 1. Next, a normal random number generator is used as the second generator with the center of the bell-shaped distribution at 6. For comparison, there are three implementations included: the naïve implementation; the Kahan summation; and our method, parallel reduction with divisions by two. The programs are executed and the results are displayed in Table 2.

TABLE 2
RESULTS WHEN RUNNING WITH DIFFERENT DATA SETS

Data Set	Problem Size	Result		
		Naïve	*Kahan Summation*	*Our Method*
U(0,1)	8388480	0.49998	0.499981	0.499981
	16776960	0.499953	0.499981	0.499981
	33553920	0.499944	0.499981	0.499981
	67107840	0.250004	0.499981	0.499981
	134215680	0.125002	0.499981	0.499981
N(6,1)	8388480	6.00014	5.99992	5.99992
	16776960	6.60712	5.99992	5.99992
	33553920	4.14664	5.99992	5.99992
	67107840	2.25161	5.99992	5.99992
	134215680	1.30409	5.99992	5.99992

The above table shows that our method computes more reliable results than the naïve implementation. We show that our method can compute a valid result as good as a result from the Kahan summation.

B. Running Time

To show the performance improvement, we capture the running time of our CUDA programs with both the interleaved and sequential addressing schemes and compare it with the running time of the program with the Kahan summation. The Kahan summation is chosen to be a representative of the sequential side because it computes a valid answer.

The logarithmic chart, Fig 8, shows that our parallel programs perform significantly better than the sequential program. It also indicates that the sequential addressing implementation is slightly faster than the interleaved addressing implementation.

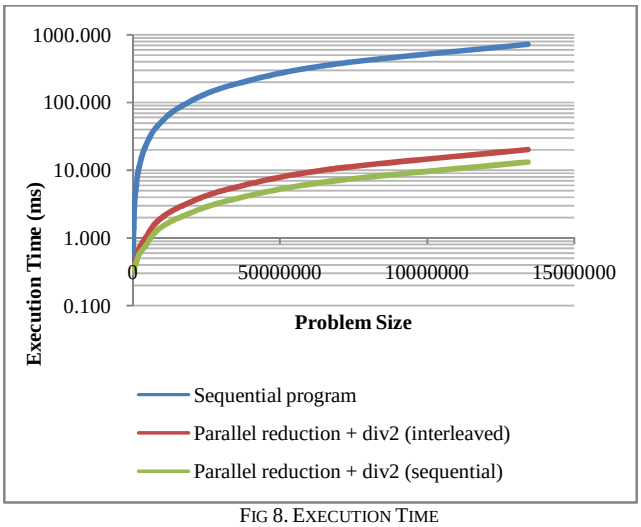

FIG 8. EXECUTION TIME

C. Speed-Up

We calculate the speed-up of the parallel program with different block sizes. To simplify the chart, we include only the sequential addressing implementation since it performs better than the interleaved addressing version.

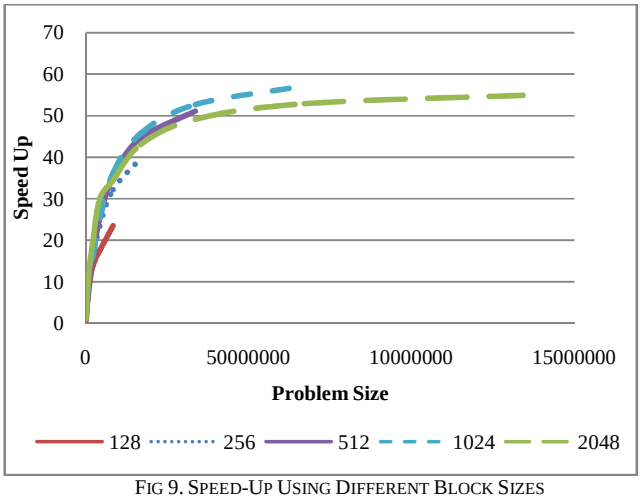

FIG 9. SPEED-UP USING DIFFERENT BLOCK SIZES

According to Fig 9, for each block size, the speed-up is increasing with a higher accelerate at first before the rate of increase becomes much lower later. This diminishing return indicates that there must be a bottleneck in CUDA core resources. With larger problem sizes, many blocks may have to be idle waiting to be executed due to limited resources of available CUDA cores.

In addition, the chart also reveals that in general using a larger block size is better because one can scale as well as gain higher speed-up. However, when the largest block size, 2048, is used, the speed-up is actually dropped. This may be caused from some congestion of threads due to limitation of some shared resources of threads within the same block.

Note that each line in the above figure is for a certain block size. So, referring to Table 1, the lines in the chart will have difference in length. For example, the line with the block size of 128 is short because it can only scale up to the problem size of 8388480.

TABLE 3
SPEED-UP USING DIFFERENT PROBLEM SIZES AND BLOCK SIZES

Problem Size	**Block Size**				
	128	***256***	***512***	***1024***	***2048***
65535	0.75	1.115385	1.084112	1.080745	1.067485
131070	1.465363	1.487179	2.115502	2.115502	2.102719
262140	2.823887	2.894191	4.201807	4.078947	4.020173
524280	5.229907	5.507874	5.518738	7.603261	7.521505
1048560	9.116694	9.833916	10.04464	13.75306	13.26651
2097120	13.89176	15.42896	16.06543	16.11127	19.60764
4194240	17.43948	23.3909	25.21628	25.35762	29.84037
8388480	23.4529	32.20398	35.45462	36.2488	34.45703
16776960		39.67046	44.54398	45.68901	43.00142
33553920			51.08197	52.70009	49.11728
67107840				57.19499	52.88059
134215680					54.9557

To obtain the optimal performance, one may have to tune the block size for a particular problem size of interests. Table 3 shows details of the speed-up obtained with a variation of the problem size and the block size.

According to Table 3, the peak value of the speed-up obtained is 57.19499 when using 1024 as the block size and 67107840 as the problem size.

V. CONCLUSION AND FUTURE WORK

We propose a method to be combined with parallel reduction to ensure the correctness of the result and also improve the performance. We develop CUDA programs with some optimization techniques to help accelerate the computation. From the experiment, the maximum speed-up obtained is 57.19499 times the sequential code.

In addition, some of our primitive ideas presented are general so that they can be applied into much broader contexts such as to scale the problem size by letting each thread does more work or to reduce the numerical error in floating-point summation by using the division technique.

Nevertheless, the work can be further improved in many ways. For example, more optimization techniques such as loop unrolling and special instructions like *__mul24* can be employed into the CUDA code.

One may try to modify the program to utilize the multidimensional feature of CUDA. That is, the problem may be formulated and divided into 2D or 3D blocks.

In the future, it would be interesting to see how the programs scale and perform on newer hardware like Fermi that has the compute capability of 2.0. Therefore, with a more powerful graphic card like NVidia GeForce GTX 480 which is also empowered by the Fermi architecture, it would greatly enhance the capability of the CUDA programs.

REFERENCES

[1] D. Heckerman, A tutorial on learning with Bayesian networks. in Learning in Graphical Models (ed. M.I.Jordan) 301-354. 1998.

[2] L. Tierney, and J. Kadane, "Accurate Approximations for Posterior Moments and Marginal Densities," Journal of the American Statistical Association, 81, 82-86. 1986.

[3] M. Spiegel, Probability and Statistics. New York: McGraw-Hill, 1975.

[4] S. Chapra, and R. Canale, Numerical methods for engineers (2d ed.): New York, McGraw-Hill, 812 p. 1988.

[5] E. De Doncker, Y. Shimizu, J. Fujimoto, F. Yuasa, K. Kaugars, L. Cucos, J. Van Voorst, Loop integration results using numerical extrapolation for a non-scalar integral, in Nuclear Instruments and Methods in Physics Research, Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 534 (1-2), pp. 269-273. 2004.

[6] M. Harris, Mapping computational concepts to GPUs, in: M. Pharr (ed.), GPUGems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation, chap. 31, Addison-Wesley, pp. 493–508. 2005.

[7] NVIDIA CUDA: Compute Unified Device Architecture, NVIDIA Corp., 2007.

[8] M. Harris, Optimizing Parallel Reduction in CUDA. NVIDIA Developer Technology. 2008.