Accelerated Parallel Generation of Binomial Coefficients using GPU

Mohsin Altaf Wani
Research Scholar computer science
Mewar University
Chittorgarh
Rajasthan

S.M.K Quadri
Research Supervisor computer sciences
Mewar University
Chittorgarh
Rajasthan

ABSTRACT

GPUs (Graphics processing units) can be used for general purpose parallel computation. Developers can develop parallel programs running on GPUs using different computing architectures like CUDA or OpenCL. The Binomial Coefficient Generation is used to generate a table of binomial coefficients each entry in row n and column k of this table contains number of combinations of n objects taken k at a time. It is known that this problem can be solved by dynamic programming technique using O(nk)-time complexity algorithm where the table to be generated has n rows and k columns. The main contribution of this paper is to present a parallel implementation of this O(nk)-time algorithm on a GPU and to analyze the speed up possible when compared to a CPU based implementation.

Keywords

Dynamic programming; Parallel algorithm; GPU; OpenCL.

1. INTRODUCTION

GPUs are specialized processors designed to accelerate computations for drawing and manipulating images[5]-[7]. Modern GPUs have teraflops of processing power and manufacturers are designing them for general purpose computing in addition to their traditional domain of graphics processing. Nowadays GPUs are attracting attention of a large number of developers. AMD GPUs can be programmed using OpenCL(Open Computing Language) framework which is specified by the khronos group[8]. Similarly NVIDIA provides a parallel computing architecture known as CUDA(Compute Unified Device Architecture)[3], the computing engine in the NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in the GPUs.

Dynamic programming is an algorithmic technique used to find a solution of a problem over an exponential number of candidate solutions [2]. In this approach we solve small instances first, store the results, and later, whenever we need a result, look it up instead of recomputing it. The term "dynamic programming" comes from control theory, and in this sense "programming" means the use of an array (table) in which a solution is constructed. The steps in the development of a dynamic programming algorithm are as follows:

i. Establish a recursive property that gives the solution to an instance of the problem.

ii. Compute the value of an optimal solution in a bottom-up manner.

Several important problems like optimal binary search tree, edit distance problem can be solved using the dynamic programming [1].

The main contribution of this paper is to implement a dynamic programming solution to generate binomial coefficients [2] on the GPU and to analyze the performance when compared to a CPU based implementation.

2. BINOMIAL COEFFICIENTS AND DYNAMIC PROGRAMMING

The main purpose of this section is to define how the Binomial Coefficients are generated and to review the dynamic programming algorithm to solve it.

The goal here is to develop an algorithm that generates Binomial coefficients. Binomial coefficients can be generated using the formula given below as equation 1.

Equation 1.

$$^{n}C_{k} = \frac{n!}{n!\,(n-k)!} \qquad \text{for } 0 \le k \le n$$

For values of n and k that are not very small, we cannot calculate the binomial coefficients directly from this definition because n! is very large for even modest values of n. However it has been established that binomial coefficients can also be computed using an alternative method given below equation 2.

Equation 2.

$$^{n}C_{k} = \begin{cases} ^{n-1}C_{k-1} + {}^{n-1}C_{k} & 0 < k < n \\ \text{or} & \\ 1 & k=0 \text{ or } k=1 \end{cases}$$

The equation 2 eliminates the need to compute n! or k! by using recursive property of binomial coefficients.

An efficient dynamic programming algorithm to compute binomial coefficients based on equation 2 exists. We will use equation 2 to construct our solution in an array C where C[i][j] will contain ${}^{i}C_{j}$. The steps required to construct a dynamic programming algorithm are as follows:

i. Establish a recursive property. Which in our case is equation 2. Now state that equation in terms of array C, which is

Recurrence 1

$$C[i][j] = \begin{cases} C[i-1] + C[i-1[j] & 0 < j < i \\ \text{or} & \\ 1 & j=0 \text{ or } j=i \end{cases}$$

ii. Solve a given instance of the problem in bottom-up manner by computing rows in C in sequence starting with first row.

This method gives us the ability to use previously computed results to compute new ones. Also saving us the burden of generating huge numbers like n! or k! in our algorithm. The steps required to generate Binomial coefficients is shown in figure 1 below. As can be seen in figure 1 this array is basically Pascal's triangle. In general we need to compute the values in each row up to the kth column only.

Figure 1.

	0	1	2	3	4	j	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
						C[i-1][j-1]C[i-1][j]	
i						→ C[i][j]	
n							

The computation proceeds as follows:

I. Compute row 0: C[0][0] = 1

II. Compute row 1: C[1][0] = 1 , C[1][1] = 1

III. Compute row 2: C[2][0]=0, C[2][1] = C[1][0] + C[1][1] = 1 + 1 = 2, C[2][2]=1

.

.

We can go on computing larger values of the Binomial coefficient in sequence. After each iteration the values needed in the next iteration are already available. This procedure is fundamental to dynamic programming approach.

3. CUDA

CUDA (compute unified device architecture) is a parallel computing architecture with a parallel programming model and instruction set architecture that is used to harness the parallel compute engine in Nvidia GPUs to solve many complex computational problems in a more efficient way than on a CPU[3].

CUDA parallel programming model has a hierarchy of thread groups called grid, block and threads[3]. A single grid is composed of multiple blocks, each of which has equal number of threads. Blocks are allocated to streaming multi processors such that all threads in a block are executed by the same streaming multi processor in parallel. All threads can access the global memory. However, threads in a block can only access shared memory of streaming processor to which block is allocated; shared memory is on chip unlike global memory which is DRAM on board having higher latency. Threads in different blocks cannot share data in shared memory.

The multiprocessor executes threads in groups of 32 parallel threads called warps. Threads composing a warp start together at the same program address, however they are free to branch and execute independently. But a divergent branch may lead to poor efficiency[4].

Threads can access data from multiple memory spaces. Each thread has its own register and private local memory. Each block has a shared memory with high bandwidth only visible to all threads of the block.

CUDA C extends C language and allows programmers to define C functions known as kernels in CUDA. By invoking a kernel, all blocks in a grid are allocated to streaming multiprocessors. The kernel call terminates when all threads in a block finish the computation. All threads in a block are executed by a single streaming multiprocessor, they are barrier synchronized by calling CUDA C __syncthreads() function[3]. However there is no direct way to synchronize threads executing in different blocks.

In this paper a parallel implementation of a dynamic programming algorithm on a GTX 570 GPU having 480 CUDA cores in 15 Streaming multiprocessors is developed.

4. PARALLEL IMPLEMENTATION

Purpose of this section is to show an implementation of Binomial coefficient generation using dynamic programming on GPU. A parallel implementation of this algorithm on a GPU is developed and its performance is analyzed.

4.1 PARALLEL ALGORITHM

This parallel algorithm for Generation of binomial coefficients is designed as follows. Each row in the array is computed by the parallel algorithm separately. The element in first row and column is set to 1, then the kernel is invoked with parameters which adjust the number of threads executing at a time. The approach here is to use a single thread to compute individual element in the array.

Algorithm 1.

```
__global__ void binomial(double *C, int row)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int col = tid + 1;
    B[row*N+col] = B[(row-1)*N + (col-1)] + B[(row-
_                   1)*N +col];
}
```

This kernel is invoked from host code where b is the array in which binomial coefficients are calculated and row is used to select the row to be computed. Each row in the array is computed using a separate kernel call. First kernel call is used to compute first row, after first kernel call completes and returns to host code second kernel call is issued to compute second row. The computation is repeated in this manner. This method of issuing separate kernel calls eliminates the problems of synchronizing computation of different rows on GPU because CUDA doesn't guarantee to maintain the order of execution of different blocks of threads.

5. EXPERIMENTAL RESULTS

The dynamic programming algorithm for generating Binomial coefficients has been implemented using CUDA C . Nvidia GeForce GTX 570 with 480 processing cores(15 Streaming multiprocessors with 32 cores each) and 1.25GB GDDR5 DRAM is used for computing results. For the purpose of estimating speedup of a GPU based implementation, a CPU based implementation has also been developed. Intel core i5 760 running at ~3.0 GHz and 8GB RAM is used for sequential implementation of dynamic programming algorithm.

Table 1 shows computing time in seconds for binomial coefficients in tables of size 5000, 7000, 9000, 12000.

Table 1: Time in milliseconds for generating Binomial coefficients GPU and CPU

Size	4500	5500	6500	7500	8500	9500
Time on GPU	102.3	148.5	202.0	269.9	338.4	422
Time on CPU	983	1591	2278	3073	4009	5039

Table 1 shows us, that for generating small instances of Binomial Coefficients GPU based implementation is 9.6 times faster than CPU based implementation. Once larger instances are considered the GPU based implementation is up to 12 times faster than CPU based implementation. So a best possible speed up factor of 12 is possible for large instances of chained matrix multiplication using the proposed implementation.

6. CONCLUSION

In this paper a parallel implementation of dynamic programming algorithm for generating Binomial coefficients on GPU has been proposed. Though the proposed algorithm requires minimum additional effort still it manages to be 12 times faster than a CPU based implementation. This proves that even a small amount of work on our part can achieve significant amount of speed up on current generation of GPUs. In our case up to 12 times as compared to CPU based implementation.

References

[1] Cormen, T. H., Lieserson, C.E., Rivest, R.L. 1990 Introduction to Algorithms. MIT Press

[2] Neapolitan, R and Naimipour, K. 2003 Foundations of Algorithms using C++ pseudocode.

[3] Nvidia Corp. 2011 Nvidia CUDA programming guide version 4.1.

[4] Nvidia Corp. 2011 CUDA C Best Practices Guide version 4.1.

[5] W. W. Hwu. 2011 GPU Computing Gems Emerald Edition. Morgan Kaufmann,.

[6] D. Man, K. Uda, Y. Ito, and K. Nakano. Dec. 2011 "A GPU implementation of computing euclidean distance map with efficient memory access," in Proc. of International Conference on Networking and Computing, , pp. 68–76.

[7] A. Uchida, Y. Ito, and K. Nakano. Dec. 2011 "Fast and accurate template matching using pixel rearrangement on the GPU," in Proc. of International Conference on Networking and Computing , pp. 153–159.

[8] AMD. 2011 Introduction to OpenCL programming.