International Journal of Computer Applications (0975 – 8887)
Volume 182 – No. 15, September 2018

Optimization Method to Reduce Matrices Multiplications in the Context of CUDA

Arezoo Khatibi
University of Kashan, Faculty of Computer Science
Kashan, Iran (Corresponding Author)
BLVD Ghotb Ravandi 6 Kilometers

Omid Khatibi
University of Vienna, Faculty of Mathematics
Vienna, Austria
Oskar morgenstern platz no 1, 1090

ABSTRACT
Parallel programming is an effective way to increase the speed of processing applications. It is carried out simultaneously by multiple processors rather than by a single processor. We compare the number of necessary calculations for multiplying the chain matrix in normal mode with the parallel mode. Since we used the famous parallel language named CUDA in our program, we will first present a brief description of the language and secondly, we explain essential mathematical notions and compare the performance of both programs.

Keywords
CUDA, GPU, Parallel programming

1. INTRODUCTION
In early computers, a processor called CPU had to do all calculations and administrative processes. After a while, experts designed a processor called GPU to delegate tasks to support the CPU. Thus, the graphics card (GPU or graphics processor that is installed on the card) with additional memory performs graphics operations and is connected to the computer's graphics power. The question is which system is more effective and on which component (the CPU or GPU) should money be spent in order to develop a more robust system?

Since the current processor speed is high, and frequency (something that usually can be seen in newer Intel processors) does not significantly contribute to overall system efficiency, the costs to purchase a better processor seem unnecessary. On the other hand, for users of graphics, games, animation and three-dimensional software, a good graphics card is much more important than a CPU. Furthermore, a modern GPU designed to help the CPU with graphical calculations has the ability to work on all kinds of arithmetic operations. Another interesting point is that a GPU does not need a large cache (memory that can be embedded in the processor) and therefore, can process much faster than a CPU. This is true both in terms of graphical computing and for all routine and professional applications [1].

The differences between a CPU and a GPU are that a CPU has a delay oriented core, large warehouses, advanced control, branch prediction, transport information, powerful ALUS and reduced latency. A GPU has an output oriented core, a small cache to increase memory ability, simple controls does not anticipate branches, and has no transit information. The common goal is to improve pipeline exploitation. The purpose of a processor (CPU) is to improve the performance of single-threading. A multi-threading processor is used to hide latency. A GPU uses shared memory to reduce memory latency. In a CPU, workloads do not require a lot of memory access and data is brought where as in the GPU there is a lot of memory access and the bandwidth is developed very well [2]. A GPU is a pile of parallel co-processors. The Kernel is composed of a grid from the thread blocks.

CPU
(host)
DRAM
GPU
(DEVICE)
DRAM

Figure 1: Allocating DRAM and ALU to CPU and GPU

2. RELATED WORKS
Chittampally Vasanth Raja, Srinivas Balasubramanian and Prakash S aghavendra made a full study heterogeneous highly parallel implementation of matrix exponentiation using GPU [1]. In [2], the authors provided an interface and a general matrix routine Implementation for multiplying small matrices that were simultaneously processed on GPU. Their matrix size was less than 16. They introduced a distributed Cuda method and compared their work with Cuda. The authors of [3] compared the speed of the CPU and GPU on systems working with daylight, and provided a better algorithm than the CPU in GPU environments in OpenCL. [4] is about the hardware architecture of matrix multiplication on real multi-core systems. The authors considered the system data as data matrices and tried to find a better destination for temporary system data in multi-layer cashes. They compared their work with a method that considered the data as a block and improved the speed of the multi-core system with their proposed algorithm. They tested their work on matrix multiplication. In [5], the authors showed memory access patterns for execution and hit cache rates in the GPU kernels. They implemented their method details for matrices whose data sizes were larger than the GPU memory and showed the results for square matrices. [6] compares two methods of balancing the load for matrix multiplication in multi-cores + GPU systems. The authors optimized the utilization of the combination of CPU and GPU libraries and attempted to reduce the runtime of these systems. [7] focuses on OpenCL kernels and uses matrix multiplication as case study. The authors obtained a new method to automatically generate performance portable GPU code.

3. MINIMIZING THE NUMBER OF MATRIX MULTIPLICATIONS
For multiplying a chain of matrices we use the following dynamic algorithm: Calculate the minimum number of multiplications for A1 * A2, A2 * A3, ..., A(n-1)*An. Initially we calculate for two matrices and then for three matrices and so on, so that the optimum of multiplications for n matrices is

calculated and obtained through the recurrence relation in each previouse step [3].

M [I] [J] = minimum (M[i] [k] + M [k+1] [j] + d [i-1]*d[k]*d[j], i<=k<=j-1).

The above process is stored as the diagonal (with column indices greater than row indices) in a matrix named M and d is an array of length n + 1 that maintains the dimensions of the matrix. The matrix structure, which consist of height, width and matrix dimensions is used in the program code and * element is a pointer to the matrix. Members are stored as rows in the matrix:

M (row, col) =*(M.element +row*M.width+col)

So the elements of the matrix diagonal will be counted by the following code:

```
for (int dia=1; dia < m.width-1; dia++)
for (int i=1; i<=m.width-dia; i++) {
        int j=i+dia;
        *(m.element+i*m.width+j) =*(m.element+i*m.width+i) + *(m.element
        + (i+1)*m.width+j) + *(d.element+i-1)* *(d.element+i)* *(d.element+j);

        for (int k=i+1; k<=j; k++)
*(m.element+i*m.width+j)=(*(m.element+i*m.width+j))<=(*(m.element+i*m.width+k)
        + *(m.element+(k+1)*m.width+j)+ *(d.element+i-1)* *(d.element+k)* *(d.element+j))?
        *(m.element+i*m.width+j) :*(m.element+i*m.width+k) + *(m.element+(k+1)*m.width+j)
        + *(d.element+i-1)* *(d.element+k)* *(d.element+j);

}
```

The result of the last diagonal is the solution to the problem. Algorithm execution time is $o(n^3)$. To use the above algorithm in parallel algorithm using CUDA first, we mapped conventional memory (dedicated matrices) in the host to the memory of graphics processors (device), which is stored in the form of a grid and blocks as follows:

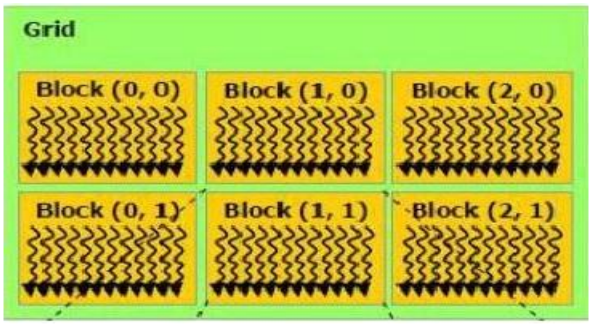

Figure 2: Grid

The number of blocks in the grid will be determined and introduced by Dim3 grid (3, 2). The threads of each block are processed by a graphics processor that can specify thread numbers in each block. Dim3 block (3, 2), we want diagonal elements to be calculated in parallel mode in a single step, for this purpose we write a global function as follows:.

```
__global__ void test(matrix D,matrix M,int DIA ){
        Int col = blockIdx.x *blockDim.x+threadIdx.x;
        Int row=blockIdx.y*blockDim.y+threadIdx.y ;
If (row>0 && col>0 && col-row == DIA) {
    Int temp =M.element [row*M.width+row] +
        M.element [(row+1)*M.width+col] +D.element [row-1]
*D.element [row]*D.element [col];
M.element [row*M.width+col] =temp;
For (int k=row+1; k<col; k++)
M.element[row*M.width+col]=(M.element[row*M.width+col])<=(M.element[row*M.width+k]
+M.element[(k+1)*M.width+col]+D.element[row-1]*D.element[k]*D.element[col])?
(M.element[row*M.width+col):(M.element[row*M.width+k]
+M.element[(k+1)*M.width+col]
+D.element[row-1]*D.element[k]*D.element[col]);
}
}
```

As shown in the above function condition,

(if (row>0 && col>0 && col-row==DIA))

Each time you start the function above, elements located on the DIA diagonal are run by the graphics processors simultaneously. After mapping the host memory to the device in the body of the program the above function will be called to calculate the elements of any diagonal (with sending size of the block and grid).

```
For   (int   dia=1;   dia<h d.width-2;
dia++)

        Test<<<grid,block>>>(d_d,d_m,dia);
```

The size of the block and grid are considered as follows:

```
Dim3 block (10, 40);

Dim3 block (10, 40);

Dim3 grid (h_m.width/10) +1, (h_m.width/40)
+1);
```

After the values of the diagonal were calculated by this function, the result is copied from the device to the host.

```
cudaMemcpy(h m.element,d m.element,si
zem,cudaMemcpyDeviceToHost);

cudaMemcpy(h_d.element,d_d.element,size,c
udaMemcpyDeviceToHost);
```

4. ANALYSIS

Table 1: Execution Time of series and parallel programs with different matrices

A total of matrices	Time of the series program with CPU intel(R) core(TM) i7-4710HQ-2.5GHZ	Time of the parallel application with NVIDIA,GEFORCE ,GTX850M
100	3.086	4.047
200	27.791	5.269
500	10.367	8.287
800	8.791	31.512
1200	12.339	7.862
1500	9.864	6.288

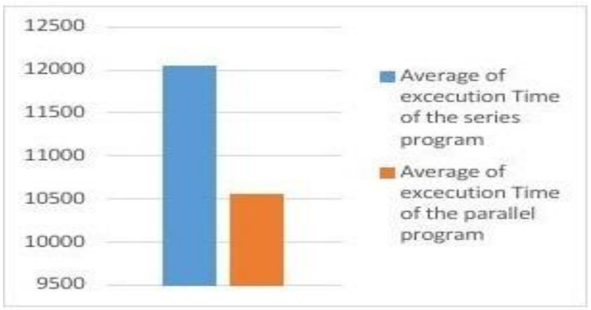

Figure 3: Comparison of the average execution time between series and parallel programs

The time unit in table 1 is millisecond. The research was conducted using a computer with CPU 2500 and 8 core. To date, we have not seen an article that optimized multiplication matrices, so we were unable to compare our work with that of other researchers. In this paper, we tried to reduce the number of matrices multiplications using parallel programming with CUDA language. In the experiment, the number of matrices is 200, the number of matrices multiplications in parallel state is very optimized then we have a huge speed up for both the series program and parallel application. When the number of matrices is 800, parallel programming is not able to optimize the number of matrices multiplications and the opposite result occurred-a significant slow down in both the time of the series program and of the parallel application. This shows that the optimization of the multipliers is not entirely useful, and in the six cases, one case has failed.

5. CONCLUSION

Speed up is a calculated by the time of the sequential program divided by the time of the parallel program.

In the case of n=100, both programs find the answer very fast and in less than a second. The largest n that our serial program is able to manage is 420, whereas the parallel program finds the answer until n = 649 in less than 5 seconds. Perhaps by finding a more suitable block and grid size a better result can be achieved. Matrices are one of the mathematical objects used in computer graphic programs, computer networks, and electrical modelling. We saved time and complexity by optimizing the multiplication of matrices and in fact, we increased the speed of programs using multiplication matrices.

6. REFERENCES

[1] Chittampally Vasanth Raja, Srinivas Balasubramanian, Prakash S aghavendra. 2012. Heterogeneous Highly Parallel Implementation of Matrix Exponentiation Using GPU. International Journal of Distributed and Parallel Systems (IJDPS), 3(2).

[2] Chetan Jhurani, Paul Mullowney. 2015. A GEMM Interface and Implementation on NVIDIA GPUs for Multiple Small Matrices. Parallel Distrib. Comput. 75, 133-140.

[3] Wangda Zuo, Andrew McNeil, Michael Wetter and Eleanor S. ee. 2014. Acceleration of The Matrix Multiplication of Radiance Three Phase Daylighting Simulations with Parallel Computing on Heterogeneous Hardware of Personal Computer. Journal Of Building Performance Simulation, 7(2), 152-160.

[4] Minwoo Kim, Won Woo Ro. 2014. Architectural Investigation of Matrix Data Layout on Multicore Processors. Future Generation Computer Systems, 37, 64-75.

[5] Kazuya Matsumotoa, Naohito Nakasato, Tomoya Sakai, Hideki Yahagi, Stanislav G. Sedukhin. 2011. Multi-level Optimization of Matrix Multiplication for GPU-equipped Systems GPU-equipped SystemsInterfaces. Procedia Computer Science, 4, 342-351.

[6] Luis-Pedro Garc´ıa, Javier Cuenca and Domingo Gim´enez. 2015. On Optimization Techniques for the Matrix Multiplication on Hybrid CPU+GPU Platforms Annals of Multicore and GPU Programming, 2(1).

[7] Toomas Remmelg, Thibaut Lutz, Michel Steuwer, Christophe Dubach. 2011. Proceedings of the 9th Annual Workshop on general purpose processing using graphics processing unit, 12 March 2016, pp.22-31.