ISSN:0975-9646

Solving Monte-Carlo Method by Using GPU and CPU.

Jameer Kotwal[1] , Dr. Sachin D Babar[2]
Sinhgad Institute Of Technology, Lonavala, Pune, India.

Abstract. An arrival of high performance computing for a big data and graphics processing unit(GPU) used for fast rendering, 3D scenes etc present an huge computational resources for big data that requires more computational power for processing the data parallel for robust and giving the correct data analysis. CUDA (Compute Unified Device Architecture) is a platform built by NVIDIA for implementing the task parallel. World is moving very fast from serial execution to parallel execution. One of the experimental result is shown in this paper by solving the monte carlo method for calculation of pi(). The calculation of pi code is executed on CPU and GPU. Here in this paper the measure factor is time required for executing the pi code on CPU well as on GPU.

Keywords: GPU, CUDA programming, C programming, Monte Carlo method, parallel programming.

1 INTRODUCTION

The use of GPU is used to accelerate the level of executing the code in parallel. Here in this paper it show how the Monte Carlo method is used to solve the calculation of pi() in parallel on GPU as well as serial on CPU. Monte Carlo is used to solve the computational problems that depend on repeated random number of sampling results. Monte Carlo methods is classify into three main classes:- numerical integration, probability distribution and optimization. Monte Carlo methods are very important for getting the simulation results with many coupled degrees of freedom such as fluids, disordered materials, strong coupling etc.

2 OVERVIEW OF PARALLEL AND SERIAL EXECUTION

GPU computational provides more powerful solving techniques than the CPU computational. Few year ago, the code is executed on CPU In the form of serial execution. CPU architecture it works completely like SISD(single instruction single data) programming style. While executing the code on cpu it requires more time to execute the code. But here today the drastic change is made in technology to go for execution of code in parallel.

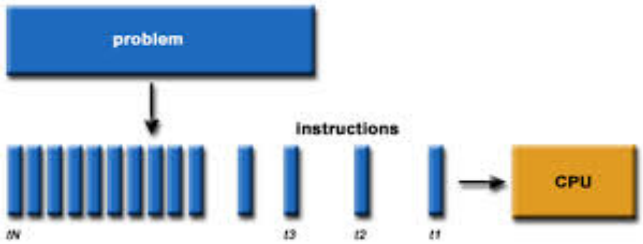

Fig 1. Serial Execution.

There are so many programming languages that work's on parallel execution like OpenACC, OpenMP, CUDA, OpenGL etc. But NVIDIA had implemented a good platform for executing the code parallel by CUDA programming on GPU(Graphical Processing Unit). GPU works on SIMD (Single Instruction Multiple Data) programming style for execution. In parallel execution, more than one task we can execute parallel by using multiple threads in the form of cores (processor).

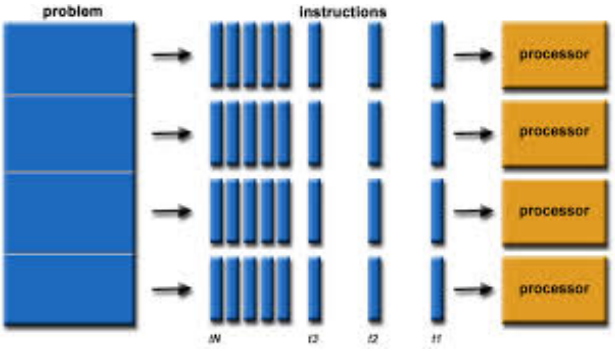

Fig 2. Parallel Execution.

3 GPU ARCHITECTURE

The GPU architecture has number of processor (cores) that work together to execute the data/code given in the application. While compare to GPU, the CPU having less cores and ALU(Arthimetic and Logical Unit). But in GPU, it have more number of cores and ALU to process the data fast. The green color show's the ALU. In GPU, cache memory is provided less compare to CPU.

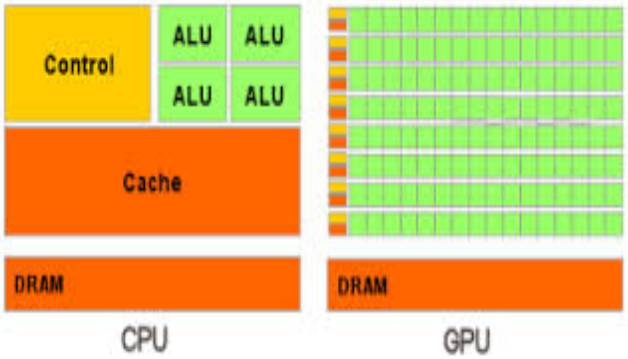

Fig 3. CPU vs GPU.

3.1 CUDA Process Flow

CUDA was developed in 2006 by NVIDIA company. In this section, we have a short look on how the programming features provided by CUDA. There are number of features are provided by CUDA to the developer. Some of them like thread, streaming processor, streaming multiprocessor etc. The diagrammatic structure is shown below :-

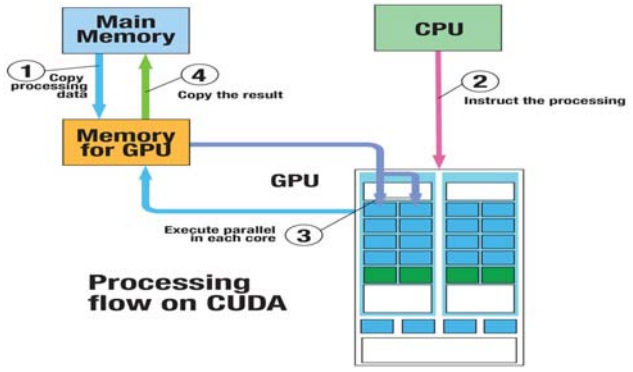

Fig 4. CUDA Processing Flow.

1. Process Flow

The CUDA program execution is done in two parts:

- **Host**.
- **Device**.

Host is referred to GPU and Device part is referred to CPU. Program is basically interacting with both the host and device during execution. When we are executing the program/code with the help of CPU and GPU the following step is as:

- The program is first copied into the cpu memory.
- Then the compiler bifurcate the serial code and parallel code. Mainly the code is composed of serial code and parallel code
- The serial code is executed on CPU and the parallel code is executed on GPU. The parallel code is copied from CPU memory to GPU memory by using function HostToDevice().
- The GPU compute the code or process the code and then again copy back from GPU memory to CPU memory by using function DeviceToHost().
- And at last the result is display with the help of CPU.

4 CUDA Architecture

In this architecture, the main role is played by threads. Threads is used to execute the code parallel on streaming multiprocessor. In Streaming multiprocessor, the n number of streaming processor is there. SP it acts as a threads. A group of 32 threads is 1 wrap. For creating the thread, the kernel function is launch, this function is used to create the grid,block and threads. The thread is run in the block, block is in the grid and grid is launch on SM. The kernel function code is <<< >>>.

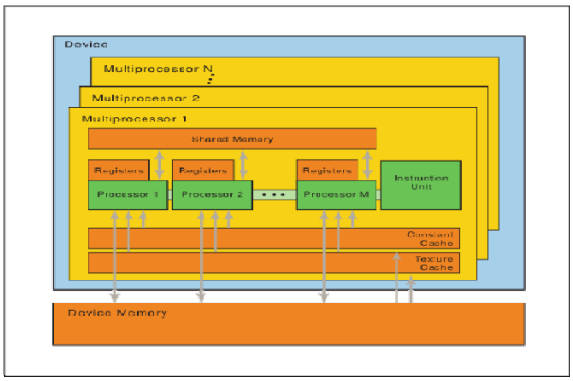

Fig 4. CUDA Architecture.

5 GPU vs CPU Experiment Result.

For illustration purpose, we implemented a calculation of pi by using monte carlo method.

- **Serial code execution on CPU**

  To set up the exeution od pi on CPU, randomly located points is generated within a given square which has a circle surface within it. Here it generates a more number of random number and checks if the x,y co-ordinates points is inside the circle. The p inside the circle to the total number of points tried is calculated. To adjust the number the pi is multiplied by 4 to get the approximate result of pi. Using this ratio, the approximation of π can be computed by assuming that P approaches the value of ratio of the area of the circle to the area of the square when the number of points, niter, is large

```
Begin Program For Monte carlo
double niter = 10000000;
double x,y;
int i;
int count=0;
double z;
double pi;
srand(time(NULL));

for (i=0; i<niter; ++i)
{
    x = (double)random();
    y = (double)random();
    z = sqrt((x*x)+(y*y));
    if (z<=1)
    {
        ++count;
    }
}

pi                                    =
((double)count/(double)niter)*4.0;
    WriteLn (pi);
END.
```

- **Parallel code execution on GPU**

  CUDA is used for execution of code parallel and it is developed by NVIDIA that extends C. In CUDA, the programmer defines the kernel function to launch millions of threads to execute the code. The programmer very keen to launch the kernel function by keeping in mind how many threads and blocks of threads they want to launch on GPU.

```
__global__      void     kernel(int*
count_d, float* randomnums)
{
    int i;
    double x,y,z;
    int    tid    =    blockDim.x    *
blockIdx.x + threadIdx.x;
    i = tid;
    int xidx = 0, yidx = 0;
```

```
    xidx = (i+i);
    yidx = (xidx+1);

    x = randomnums[xidx];
    y = randomnums[yidx];
    z = ((x*x)+(y*y));

    if (z<=1)
            count_d[tid] = 1;
    else
            count_d[tid] = 0;
}
```

The Kernel function for this above code is loaded: Kernel <<<blocks, threads>>>. In this, 100 blocks is created and in each blocks 1000 threads is created for executing the code.

The NVCC(Nvidia Compiler For C) compiler bifurcate the serial code and parallel code. By using __global__ decorator the compiler comes to know that the code is parallel and execute on GPU. GPU does not have input and output devices. CUDA launch the thread in x and y co-ordinates so that the threadIDx.x ,threadIdx.y variable is defined. We can define the x and y co-ordinates for block also.

For allocating the memory in the CPU the function CUDAMalloc() is used and for releasing the memory we use CUDAfree().For copying the code from CPu to GPU memory the function HostToDevice() is used. And for copying back the result from GPU to CPU memory the function DevicetoHost() is used.

For allocating the memory the following code is :

```
cudaMalloc((void**)&count_d,
(blocks*threads)*sizeof(int));
```

For copying the data in between CPU an GPU the code is:

```
cudaMemcpy(count,count_d,blocks*threads*
sizeof(int), cudaMemcpyDeviceToHost);
```

For releasing the memory from the GPU the code is:

```
cudaFree(randomnums);
cudaFree(count_d);
```

6. CONCLUSION:

After the comparison of result between the CPU and GPU. It is clear that the future of parallel processing is very much in the hands of the NVIDIA. Because CUDA programming gives a very more powerful result then compare to CPU for calculation of pi. CUDA programming gives a more prominent result for pi. We can solve the large number of problems with huge data by using CUDA. The IT industry is also diverting from serial computing to parallel computing. Everyone taking the challenges in the upcoming technology, both software and hardware are taking into the direction of parallel processing. The GPUs are gaining more popularity in the scientific computing community due to their high processing capability and easy availability as we have demonstrated throughout the paper, and are becoming the preferred choice of the programmer due to the support offered for programming by models such as CUDA.

REFERENCES

1. Anthony Lippert – "NVidia GPU Architecture for General Purpose Computing".
2. Wikipedia, "Pthreads," Pthreads, 2013. [Online]. Available: http://en.wikipedia.org/wiki/POSIX_Threads. [Accessed: 11-Dec-2013].
3. A. Nin, J. Diaz, and C. Mun, "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era," IEEE Trans. Parallel Distrib. Syst., vol. 23, no. 8, pp. 1369–1386., 2012.
4. J. Cook, "Pthreads, Tutorials and Tools," Tutorials and Tools, Computer Science Department, New Mexico University. [Online]. Available: http://www.cs.nmsu.edu/~jcook/Tools/pthreads/pthread_cond.html. [Accessed: 11-Dec-2013].
5. T. Liu and E. D. Berger, "D THREADS : Efficient Deterministic Multithreading," in *In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, 2011, pp. 327–336.
6. K. Lu, X. Zhou, X. Wang, W. Zhang, and G. Li, "RaceFree : An Efficient Multi-Threading Model for Determinism," in *In Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '13).*, 2013, pp. 297–298.
7. R. Crawfis, "'Modern GPU Architecture', Lecture Notes for CSE 694G Game Design and Project."Ohio State University, Spring 2007.
8. E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture.," *IEEE Microarchitecture*, vol. 28, no. 2, pp. 39–55, 2008.
9. B. J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE 96*, no. 5, pp. 879–899, 2008.
10. NVIDIA, "CUDA C Programming Guide," no. July. NVIDIA Corporation, 2013.
11. Wikipedia, "General-purpose computing on graphics processing units," 2013. [Online]. Available: http://tiny.cc/g4pw6w. [Accessed: 21-Nov-2013].
12. Wikipedia, "CUDA," *Wikipedia, The Free Encyclopedia.*, 2013. [Online]. Available: http://en.wikipedia.org/wiki/CUDA. [Accessed: 22-Nov-2013].
13. J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.
14. N. Gupta, "CUDA Programming, Complete Reference on CUDA," *CUDA programming Blog*, 2013. [Online]. Available: http://tiny.cc/2niy6w. [Accessed: 22-Nov-2013].
15. OLCF, "Oak Ridge National Labs Accelerated Computing Guide: CUDA Vector Addition," *Accelerator Programming*, 2013. [Online]. Available: https://www.olcf.ornl.gov/tutorials/cuda-vector-addition/. [Accessed: 24-Nov-2013].
16. GWU, "Supercomputers - HPCL – The George Washington University High Performance Computing Lab.," 2013. [Online]. Available: http://hpcl2.hpcl.gwu.edu/index.php/supercomputers . [Accessed: 25-Nov-2013].