PERFORMANCE EVALUATION OF PARALLEL COMPUTING SYSTEMS

Dr. Narayan Joshi
Associate Professor, CSE Department, Institute of technology
NIRMA University, Ahmedabad, Gujarat, India

Parjanya Vyas
CSE Department, Institute of technology,
NIRMA University, Ahmedabad, Gujarat, India

ABSTRACT

Optimization of any computational problem’s algorithm using a sole high performance processor can improve the execution of any algorithm only up to an extent. To further improve the execution of the algorithm and thus to improve the system performance technique of parallel processing is widely adopted nowadays. Implementing parallel processing assists in attaining better system performance and computational efficiency while maintaining the clock frequency at normal level. Pthreads and CUDA are well-known techniques for CPU and GPU respectively. However, both of them possess interesting performance behavior. The paper evaluates their performance behavior in varying conditions. The results and chart outline the CUDA as a better approach. Furthermore, we present significant suggestions for attaining better performance with Pthreads and CUDA.

Keywords: Parallel Computing, Multi-Core CPU, GPGPU, Pthreads, CUDA.

1. INTRODUCTION

Before the era of multi-core processing units began, the only way to make CPU faster was to increase the clock frequency. More and more number of transistors was attached on a chip, to increase the performance. However, addition of more number of transistors on the processor chip keep on demanding more electricity and thereby causing increase in heat emission; which imposed constraint of limited clock frequency. In order to overcome this severe limitation and to achieve higher performance with processors, advancements in micro electronics technology enabled an era of
parallel processing proposing an idea of using more than one CPUs for a common address space. The processors which support parallel programming are called multi-core processors.

Computational problems which include vast amount of data and operations that are entirely independent of the intermediate result of each other, then such problems can be solved much more efficiently with parallel programming then sequential programming. Ideally a parallel program will apply every operation simultaneously on all the data elements and will produce the final result in the time, in which sequential program can produce result of the operation on only one data element. Hence, when dealing with the problems where operations to be performed are highly independent or partially dependent on the intermediate results, parallel computing will provide immense improvement in the efficiency over sequential programs.

Section 2 describes the literature survey. Section 3 presents the architecture and working of GPGPU. The CPU parallel programming approach is mentioned in section 4. Section 5 consists of comparative study of these two parallel programming approaches. In section 6 behavior of these two approaches is discussed. The concluding remarks and further course of study are describes in section 7.

2. RELATED WORK

Because of its significant benefits, the technique of parallel computing has been widely adopted by research and development sectors to solve medium to large scale computational problems. Many researchers have studied the area of parallel programming including analysis of its various techniques in past. Jakimovska et al. have suggested optimal method for parallel programming with Pthreads and OpenMPI [3]. Moreover ShuaiChe et al. have presented a performance study of general purpose applications on graphics processor using CUDA [10]. They have compared GPU performance to both single core and multi core CPU performance. A unique approach towards gaining optimum performance is presented by Yi Yang et al; they have presented a novel approach to utilize the CPU resource to facilitate the execution of GPGPU programs as fused CPU–GPU architecture in their paper [11]. Abu Asaduzzaman et al. have presented a detailed power consumption analysis of OpenMPI and POSIX threads in [1]; they have summarized variations in the performance of MPI. Researchers have also worked on Pthreads optimization. One of such works is presented by Stevens and Chouliaras related to a parametrizable multiprocessor chip with hardware Pthreads support [4]. In another study published by Cerin et al. describes experimental analysis of various thread scheduling libraries [2].

3. GPGPU PARALLEL PROGRAMMING

A typical CUDA program follows below stated general steps [8]:

1. CPU allocates storage on GPU
2. CPU copies input data from CPU → GPU
3. CPU launches kernel(s) on GPU to process the input data
4. GPU does the processing on the data by multiple threads as defined by the CPU
5. CPU copies results from GPU to CPU

The GPGPU based parallel programs follow the master-salve processing terminology. CPU acts as master which controls the sequence of steps whereas GPU is merely a collection of high number of slave processors resulting in efficient execution of multiple threads in parallel [9][13].
As shown in the figure 1, a GPGPU consists of various Streaming Multiprocessor units (SMs); the SMs comprising of many co-operating thread processors, each of which can execute an instruction at a time. All of these thread processors do have their private local memory; as well each SM unit also has its private memory which acts as a global memory for the thread processors belonging to an individual SM unit [14]. Furthermore, the GPU also contain its device memory, which is global to all SMs. Memory model of a GPGPU can is shown in figure 2 [16][17][18].

Figure 1: GPGPU memory model

Figure 2: GPGPU architecture

4. CPU PARALLEL PROGRAMMING

Pthreads, OpenMP, TBB (Thread building blocks), Click++, MPI are some of the well known CPU parallel programming techniques available [7]. EnsarAjkunic et al have discussed their comparative study in [5]. As discussed in the introductory section above, the paper focuses on Pthread and CUDA, this section describes the Pthread library and its working [12].
The Pthread library considered in this paper for performance evaluation uses a two-level scheduler Pthreads implementation, as shown in figure 3. Pthread library facilitates a pthread_create() function for user-level thread creation. The Pthread library scheduler is responsible for thread scheduling inside a process. As and when a particular thread is scheduled by the Pthread library scheduler, it is associated with a kernel thread in the thread pool and now, scheduling of these kernel threads is done by the OS. Mapping of user level threads with kernel threads is NOT fixed or unique. Its (associated kernel thread's) ID can change over time as each user level thread can be associated with different kernel thread every time. As and when a new thread is scheduled, mapping to a kernel thread is done every time, which necessitates mode switching [15].

![Figure 3: Pthreads model](image)

Each individual thread has its own copy of stack whereas, all threads of a process share same global memory (heap and data section). Many other global resources such as process ID, parent process ID, open file descriptors, etc. are shared among the threads of a single process.

5. STUDY

The problem of matrix multiplication is addressed using Pthreads and Nvidia CUDA on CPU and GPU respectively. The test bed description of our experimental environment is shown below:

Hardware:

- CPU: Intel®CORE™ i7-2670QM CPU @ 2.20 GHz
- Mainmemory: 4 GB
- GPU: Nvidia GEFORCE® GT 520MX
- GPU memory: 1 GB

Software:

- OS: Ubuntu 13.04
- Kernel version 3.8.0-35-generic #50-Ubuntu SMP TUE DEC 3 01:24:59 x86_64 x86_64x64 GNU/LINUX
- Driver: NVIDIA-LINUX-x86_64-331.49
CUDA version: Nvidia-CUDA-6.0
Pthreads library: libpthread 2.17

**Pthreads Program:**

```c
int M,N,P; /* dimensions of input matrices and answer matrix are, 
MxN, NxP and MxP respectively */

unsigned long long int mat1[1000][1000];
unsigned long long int mat2[1000][1000];
unsigned long long int ans[1000][1000];

void* matmul(void *arg)
{
    int i,*arr;
    arr = (int *)arg;
    for(i=0;i<N;i++)
        ans[*arr][*(arr+1)] += mat1[*arr][i] * mat2[i][*(arr+1)];
}

int main(int argc, char *argv[])
{
    /*declare and initialize pthreads, pthread arguments and time variables*/
    /*Initialize the input matrices*/
    ...
    gettimeofday(&start,NULL);
    for(i=0;i<M;i++)
    {
        for(j=0;j<P;j++,k++)
            {
                arg[k][0]=i;arg[k][1]=j;/*Passing i and j as
                arguments in arg array*/
                pthread_create(&p[i][j],NULL,matmul,(void *)(arg [k]));
                ...
            }
    }
    for(i=0;i<M;i++)
    {
        for(j=0;j<P;j++)
            pthread_join(p[i][j],NULL);
        ...
    }
    gettimeofday(&end,NULL);
    /*Determine the execution time between end and start*/

    return 0;
}
```
CUDA program:

```c
__global__ void matmul(unsigned long longint *d_outp, unsigned long longint *d_inp1, unsigned long longint *d_inp2, int M, int N, int P)
{
    /*Get current working row and column number into integer variables r and c*/
    if(r*c<=M*P)//If thread is of use then
    {
        unsigned long longint temp=0;
        for(inti=0; i<N; i++)
            temp+=d_inp1[(r)*N+(i)]*d_inp2[(i)*P+(c)];
        d_outp[(r)*P+(c)]=temp;
    }
}

int main(intargc, char *argv[])
{
    /* Declare and initialize the constants M, N, P with dimensions*/
    /* Determine the total number of threads and blocks and initialize thrd and blkaccordingly */
    /* Initialize the input matrices */
    const int sz1 = M * N * sizeof(unsigned long longint);
    const int sz2 = N * P * sizeof(unsigned long longint);
    const int anssz = M * P * sizeof(unsigned long longint);

    unsigned long longint *d_inp1, *d_inp2, *d_outp;

    gettimeofday(&start,NULL);
    cudaMalloc((void**) &d_inp1,sz1);
    cudaMemcpy(d_inp1,inputarray1,sz1,cudaMemcpyHostToDevice);
    cudaMalloc((void**) &d_inp2,sz2);
    cudaMemcpy(d_inp2,inputarray2,sz2,cudaMemcpyHostToDevice);
    matmul<<<blk,dim3(thrd,thrd,1)>>>(d_outp,d_inp1,d_inp2,M,N,P);
    cudaMemcpy(outputarray,d_outp,anssz,cudaMemcpyDeviceToHost);
    gettimeofday(&end,NULL);

    /*Determine the execution time between end and start*/
    cudaFree(d_inp1);
    cudaFree(d_inp2);
    cudaFree(d_outp);
}```
Implementation and results

Here, in both of the cases CPU and GPGPU, the two input matrices are of MxN and NxP dimensions respectively; and therefore the dimension of resultant matrix is MxP. Considering the equal values of M, N, and P the experiments were carried out on CPU and GPGPU. The results in terms of time taken by each experiment on CPU and GPGPU are shown in Table 1.

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>Time taken in milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pthreads</td>
</tr>
<tr>
<td>50</td>
<td>52</td>
</tr>
<tr>
<td>55</td>
<td>63</td>
</tr>
<tr>
<td>60</td>
<td>71</td>
</tr>
<tr>
<td>65</td>
<td>80</td>
</tr>
<tr>
<td>70</td>
<td>94</td>
</tr>
<tr>
<td>75</td>
<td>107</td>
</tr>
<tr>
<td>80</td>
<td>130</td>
</tr>
<tr>
<td>85</td>
<td>156</td>
</tr>
<tr>
<td>90</td>
<td>168</td>
</tr>
<tr>
<td>95</td>
<td>195</td>
</tr>
</tbody>
</table>

6. DISCUSSION

Chart 1: Line chart of results in Table 1
Chart 1 also depicts an exceptional behavior of GPGPU – CUDA takes nearly the same time in solving the computational problem even for less number of dimensions and threads; apart from the thread management operations, CUDA program involves two way CPU-GPU I/O operations. As stated in section 3, each time a new thread is scheduled, a mode switch is mandatory in addition to a context switch, as each user level thread is associated with a kernel level thread. These mode switches and context switches increase with increase in number of threads. This is in support to the justification about the increase in execution time which is represented by the Pthreads line in chart 1. Based on the discussion made above, some performance-centric suggestions are made here: Problems comprising of threads less than the threshold point should be assigned for execution using Pthreads to the CPU; or the problem may be submitted to the GPGPU as a wholeelse the problem may be divided to run jointly on CPU and GPGPU simultaneously. Furthermore, it may become desirable to adopt the fused CPU-GPU processing units [ieee1], to deploy a computational problem upon them.

Another notable suggestion towards Pthreads library maintainers is given here. While solving the computational problem, the library may decide to divide the total threads between CPU and GPGPU with respect to the threshold point already discussed above. Moreover, one more suggestion to the CPU manufacturers and the operating system developers is to reserve and designate one specific core as a master core dedicated for scheduling, mode switching and context switching; it may result into freeing other cores i.e., slave cores from the scheduling and switching responsibilities thereby dedicating them only for solving of the computational problems and thus increase in overall performance.

7. CONCLUDING REMARKS

A noble approach comprising of the performance determination of the CUDA and the Pthreads parallel programming techniques is presented in this paper. The extra-ordinary performance behavior of CUDA along with the threshold point is also highlighted in the paper. Furthermore,
significant suggestions pertaining to performance improvement with the CUDA and the Pthreads parallel programming approaches are also presented in this paper. In future we intend to continue our work in the direction of betterment of the open source Pthreads technique.

REFERENCES