Optimizing Parallel Computing Systems

Author: Parjanya Vyas (16305R004)

Guided By: Prof. R. K. Shyamasundar

Abstract

Optimizing algorithms can only speed up the execution of a program up to an extent. To further decrease execution time, parallelizing the program is usually the best and only option. Parallel computing is achieved by dividing a task into multiple threads and executing each thread on multiple cores of a CPU (Central Processing Unit) or multiple cores of a GPU (Graphical Processing Unit). These techniques of parallel programming differ in several aspects as the architectures of CPU and GPU are quite different. Here in this project, an approach to take use both processing units for executing a program is discussed and implemented. Furthermore, result of this approach are compared with that of pure CPU or pure GPU parallel programs for execution time.

Introduction

Parallel computing is a very important technique to increase processing power in modern machines. Increasing clock rates is no longer a viable option to speed up the processing due to enormous power consumption and heat generation. Optimizing pipelining and increasing pipe stages can also improve the execution speed up to a threshold. After that, the simplest and most common approach to further decrease execution time is to use multiple processing units in parallel.

To parallelize an algorithm, one must be able to decompose the problem into smaller parts. This is a similar approach to “divide and conquer” but differs with it in a very critical fashion. To parallelize, all the smaller parts of the bigger problems, must be mutually independent. Only then, can they be computed parallelly. If the sub-problems depend on each other, they can be executed only into a sequential manner. This fact makes parallel computing, a slightly complicated optimizing approach then the other possible approaches and hence, cannot be applied to every algorithm in general.

There are two broad classes of architecture that are used to execute a parallel program. One is a traditional processing unit of a normal computer, known as CPU. Normally, all well-known languages provide inherent support to create, execute, join, kill, synchronize and otherwise manipulate CPU level threads. Pthreads, OpenMP, TBB (Thread building blocks), Click++, MPI are some of the well-known CPU multi-threading techniques [2]. Their comparative performance is discussed by E. Ajkunic et al [1].

Another class of processing architecture is a General Purpose Graphical Processing Unit (GPGPU). Most optimized techniques for executing parallel programs in such architectures are provided by the manufacturer itself as an extension of well-known languages. CUDA is such a language, which is essentially an extension of C++, made specifically for creating efficient GPU parallel programs for Nvidia GPUs. Other generic techniques, such as openGL [3] also exist for creating generic GPU programs. CUDA and OpenGL are also compared for their performance for various parallel algorithms [4][5].

The objective of this project is to utilize the combined power of Pthreads and CUDA to execute a program. There are several techniques to independently create efficient GPU or CPU programs but there is no generic framework available to utilize both processing powers. The research goal is to implement a framework which can efficiently divide the algorithm in two parts and execute threads of each part on CPU and GPU respectively. Pthreads library is used to execute the CPU threads and CUDA for executing GPU threads in this project. Matrix multiplication problem is taken as a sample problem to evaluate performance of the proposed combined approach and compare it with pure CUDA and pure Pthreads program for same matrix sizes. Results are tabulated at the end and possible explanations are proposed.

Literature Survey

Due to immense popularity of parallel computing systems, lot of research has been done to optimize available techniques and find new approaches. Both the models of parallel computing, i.e., CPU parallel computing and GPU parallel computing is described below.

The Pthreads library is POSIX standard for C and C++ language based multiprocessing. It provides efficient implementation of APIs to use CPU threads [6]. IT uses a two-level scheduler to schedule the user level threads. These threads are mapped to kernel level threads and then scheduled on available CPU cores by second level of the scheduler, which is OS's scheduler. The programming model and mapping between user threads and kernel threads using Pthreads library scheduler is shown in Figure 1.

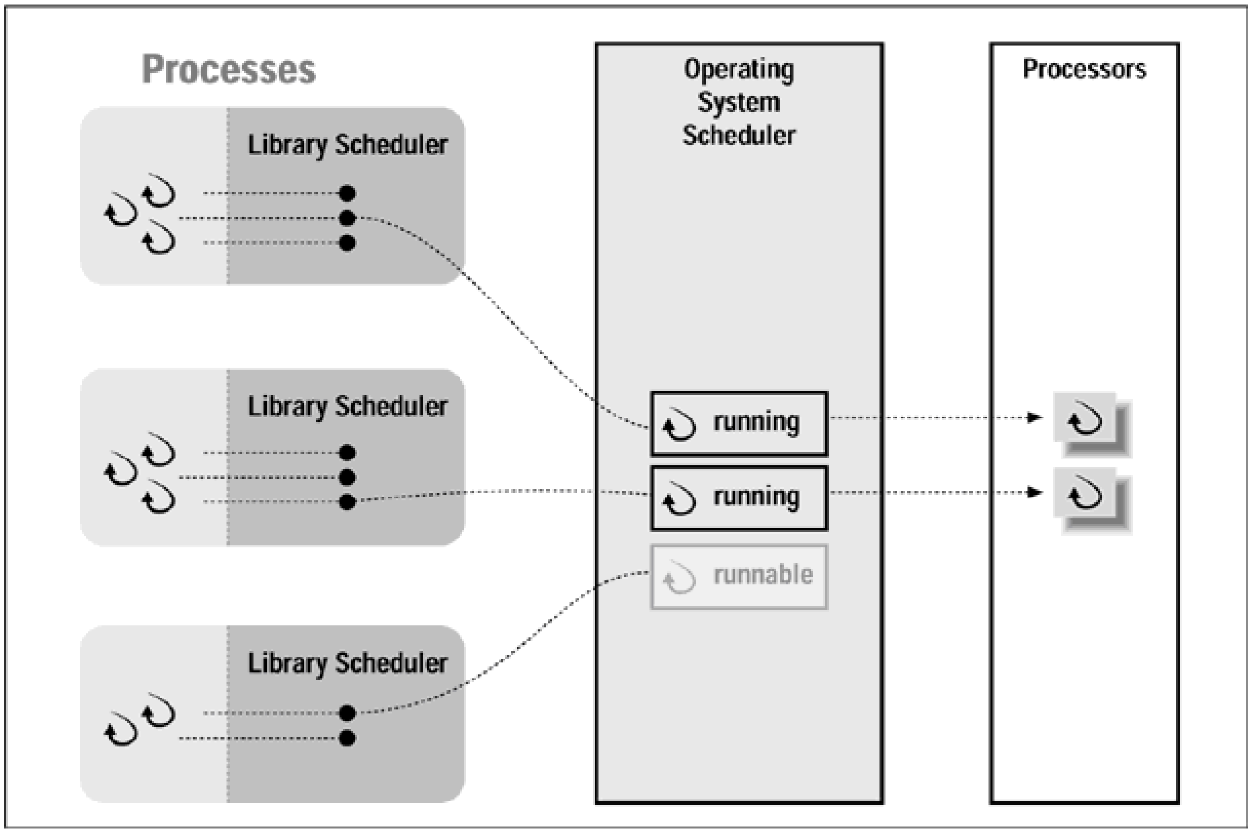

Figure 1 [6]

Typical CUDA programs follow a model consisting 4 steps as below [8]:

1. Memory initialization on GPU.
2. Transferring inputs from CPU to GPU.
3. Launching single or multiple kernels to create and start GPU threads.
4. Transferring results back from GPU to CPU.

Architecture of processors of a typical GPU is shown in Figure 2. The entire GPU consists of multiple Streaming Multiprocessors (SMs). Each SM in turn contains number of cores, that are actual processing unit. A CUDA program, written to be executed on such a GPU consists of logical components named blocks and threads. Each block consists of multiple threads. Threads inside a single block are scheduled on a single SM. Which block is scheduled on which SM cannot be controlled by programmer. The architecture of CUDA program and GPU are discussed in detail by Hong et al [9].

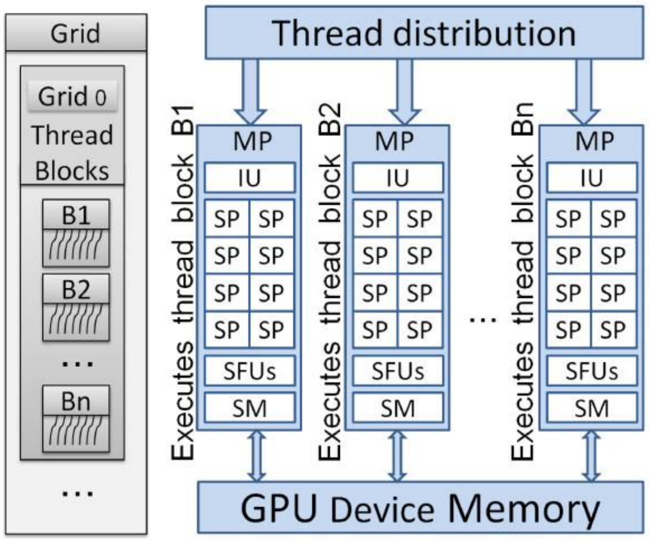

Figure 2 [11]

The memory model of a GPU is divided in three hierarchical categories. Global memory, black memory and local thread memory. Each of these is increasingly faster to access but smaller in size respectively. The memory architecture of GPU is shown in Figure 3.

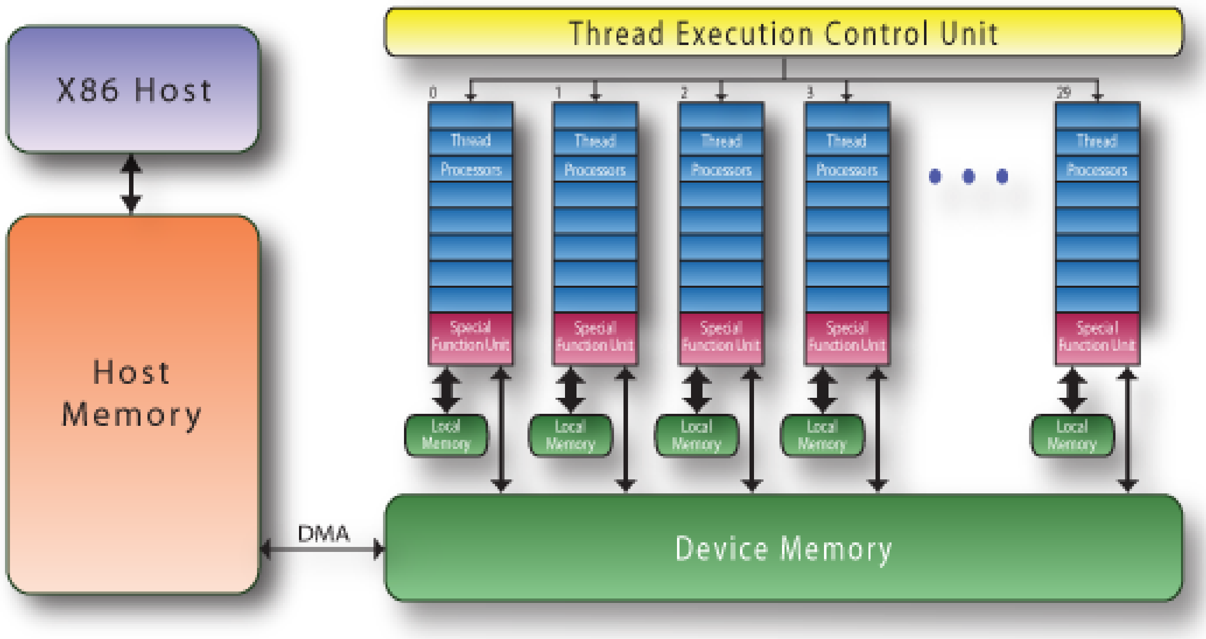

Figure 3 [10]

To utilize processing power of both, CPU and GPU, firstly, the problem at hand must be decomposed into independent sub-problems of smaller size. After death, optimum number of sub-problems must be selected to execute on CPU whereas the rest of them are to be executed on the GPU. Finally, the results from both are combined and returned to the user.

To optimally divide a problem into sub-problems one can use power lists [14]. Power lists provide a very good means to optimally calculate the recursion depth for a specific GPU given the number of SMs and threads per SM are available [13].

Narayan Joshi et al shows that there exists a threshold point at which the division of threads between CPU and GPU is optimum [12]. If this point can be found at run time, then one can easily divide the threads between CPU and GPU for optimum performance.

To distribute the threads between GPU and CPU, one can consider a CPU as Kth fraction of a GPUs. Now, distribution of threads between multiple GPUs is discussed by Anshu Anand et al [13]. The same technique can be used for distribution of threads between CPU and GPU if a CPU is considered as 'K' times a GPU.

Discussion

As stated previously, matrix multiplication is taken as a sample parallel program for performance evaluation here in this project. Three different generic programs are created, all of which takes size of matrix "N" as input and creates two random matrices of size NxN. Multiplication of these two matrices is calculated using parallel computing. Each element of resultant matrix is calculated as a separate thread of either Pthreads library or CUDA. The programs are as follows:

1. A pure Pthreads based program to calculate multiplication of two NxN matrices.
2. A pure CUDA based program to do matrix multiplication. Number of blocks and threads per block are computed using optimum recursion depth technique [13].
3. A mixed approach which takes "K" as input and does matrix multiplication by running K blocks on CPU using Pthreads whereas remaining blocks on GPU using CUDA.

For each program, the total number of threads are N^2, where N is the size of random square matrix. The results of time taken for different matrix sizes by different approaches are measured and tabulated in Table 1. Here the time taken is in milliseconds unit. Comparison of these approaches using a line chart is displayed in Figure 4.

Matrix Size	Pthreads	CUDA	Mixed_With_K1	Mixed_With_K2	Mixed_With_K3
10	51	355	56	62	80
20	283	343	366	480	558
50	2370	354	375	533	583
100	10348	350	359	446	531
150	-	348	414	474	585
200	-	345	394	490	648
250	-	353	387	439	604
300	-	370	384	492	540
350	-	381	394	481	549
400	-	414	408	490	587

450	-	430	446	479	653
500	-	455	494	541	642
550	-	483	533	572	793
600	-	554	618	703	796
650	-	590	674	759	896
700	-	651	699	745	894
750	-	731	776	831	995
800	-	809	830	872	980
850	-	892	980	1015	1136
900	-	994	1026	1117	1229
950	-	1097	1246	1292	1419
1000	-	1237	1315	1382	1501

Table 1

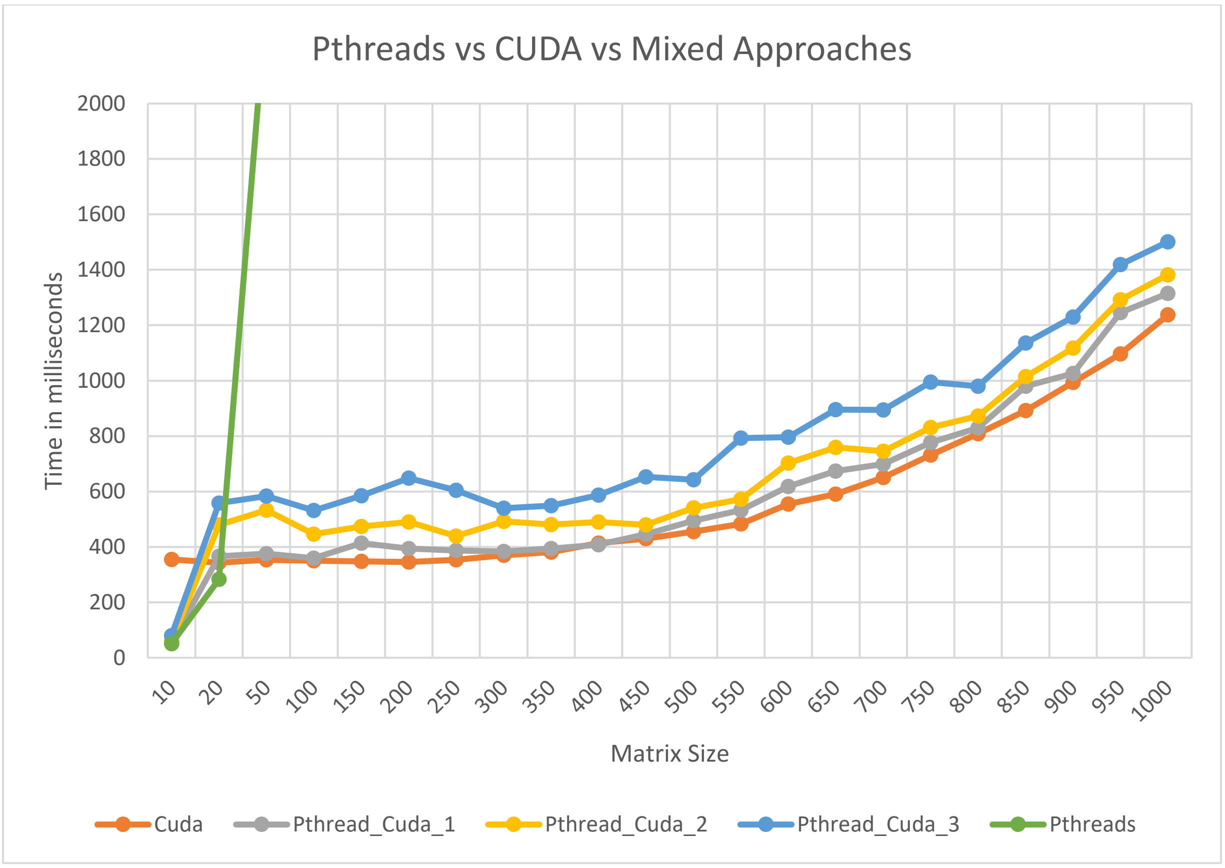

Figure 4

Conclusion

As seen from the results above, it is clear that pure Pthreads program is only good for very small sizes of matrices. The time complexity increases exponentially with matrix size for parallel Pthread program.

The performance of pure CUDA program and mixed approaches is quite comparable. Though pure CUDA program still attains best performance in most of the cases.

From the behavior of pure Pthreads program, we can say that increasing the size of "K" for mixed approach will only increase the time taken by the entire program after a threshold. But for smaller sizes of K, the mixed approach should've worked better than pure CUDA program whereas the results state otherwise. Possibly reasons for such a behavior might be inefficiency in memory management of the program.

Currently, the mixed approach of the program calculates two different results, one for Pthreads and one for CUDA part and combines the result in a new resultant matrix. If the results from both the parts would directly write into same resultant matrix, then the performance might be slightly better than the current mixed approach.

Future Work

It is clear from the result that the behavior of the current mixed approach is not sufficiently efficient to beat the pure optimized CUDA program. Theoretically, this should not be the case, as the mixed approach utilizes the processing power of CPU and GPU both in contrast to CUDA utilizing only the GPU. Hence, an open area of research would be to further optimize the proposed mixed approach in terms of memory management to reduce the number of memory transfers as much as possible and repeat the experiment.

Another open research question is to suggest a better approach to determine the threshold point for dividing the threads between CPU and GPU at run time. Though the current approach seems to be doing far better performance than the pure Pthreads program, it divides the number of threads only in multiples of blocks, determined by the GPU used. An optimal approach would use parameters of CPU and GPU both to determine the division better than the current mixed approach.

References

1. E.Ajkunic, H. Fatkic, E. Omerovic, K. Talic and N. Nosovic (2012), "A Comparison of Five Parallel Programming Models for C++", MIPRO 2012, Opatija, Croatia.
2. G. Narlikar, G. Blelloch, (1998), "Pthreads for dynamic and irregular parallelism. In Proceedings of the 1998 ACM/IEEE conference on Supercomputing (SC '98), IEEE Computer Society, Washington, DC, USA, 1-16.
3. Woo, Mason, et al. OpenGL programming guide: the official guide to learning OpenGL, version 1.2. Addison-Wesley Longman Publishing Co., Inc., 1999.
4. Sachetto Oliveira, Rafael, et al. "Comparing CUDA, OpenCL and OpenGL implementations of the cardiac monodomain equations." Parallel Processing and Applied Mathematics (2012): 111-120.
5. Weinlich, Andreas, et al. "Comparison of high-speed ray casting on GPU using CUDA and OpenGL." Proceedings of the First International Workshop on New Frontiers in High-performance and Hardware-aware Computing. Vol. 1. Proceedings of HipHaC'08, 2008.
6. Nichols, Bradford, Dick Buttlar, and Jacqueline Farrell. Pthreads programming: A POSIX standard for better multiprocessing. " O'Reilly Media, Inc.", 1996.

7. http://maxim.int.ru/bookshelf/PthreadsProgram/img/06FIG01_0.gif retrieved on April 28, 2017.
8. Sanders, Jason, and Edward Kandrot. CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents. Addison-Wesley Professional, 2010.
9. Hong, Sunpyo, and Hyesoon Kim. "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness." ACM SIGARCH Computer Architecture News. Vol. 37. No. 3. ACM, 2009.
10. http://rits.github-pages.ucl.ac.uk/research-computing-with-cpp/09Accelerators/sec02UsingAGPUAsAnAccelerator.html retrieved on April 28, 2017.
11. https://www.researchgate.net/profile/Satish_Chikkagoudar/publication/51168475/figure/fig1/AS:214151432544256@1428069088250/NVIDIA-GPU-Architecture-Simplified-GPU-Architecture-The-grey-rectangles-of-thread.png retrieved on April 28, 2017.
12. Dr. Narayan Joshi and Parjanya Vyas, “Performance Evaluation of Parallel Computing Systems”, International Journal of Advanced Research in Engineering & Technology (IJARET), Volume 5, Issue 5, 2014, pp. 82 - 90, ISSN Print: 0976-6480, ISSN Online: 0976-6499.
13. Anand, Anshu S., and R. K. Shyamasundar. "Scaling Computation on GPUs Using Powerlists." High Performance Computing Workshops (HiPCW), 2015 IEEE 22nd International Conference on. IEEE, 2015.
14. Misra, Jayadev. "Powerlist: A structure for parallel recursion." ACM Transactions on Programming Languages and Systems (TOPLAS) 16.6 (1994): 1737-1767.