Optimizing Parallel Computing Systems

Author: Parjanya Vyas

Guided by: Prof. R. K. Shyamasundar

Agenda

- Motivation – Why optimize?
- Objectives – Utilizing all available resources!
- Literature Survey – How much is already optimized?
- Approach and Observations – What is proposed? How good is it?
- Concluding Remarks – Explaining the behavior!
- Future Work – Some open questions!
- Questions?

Motivation

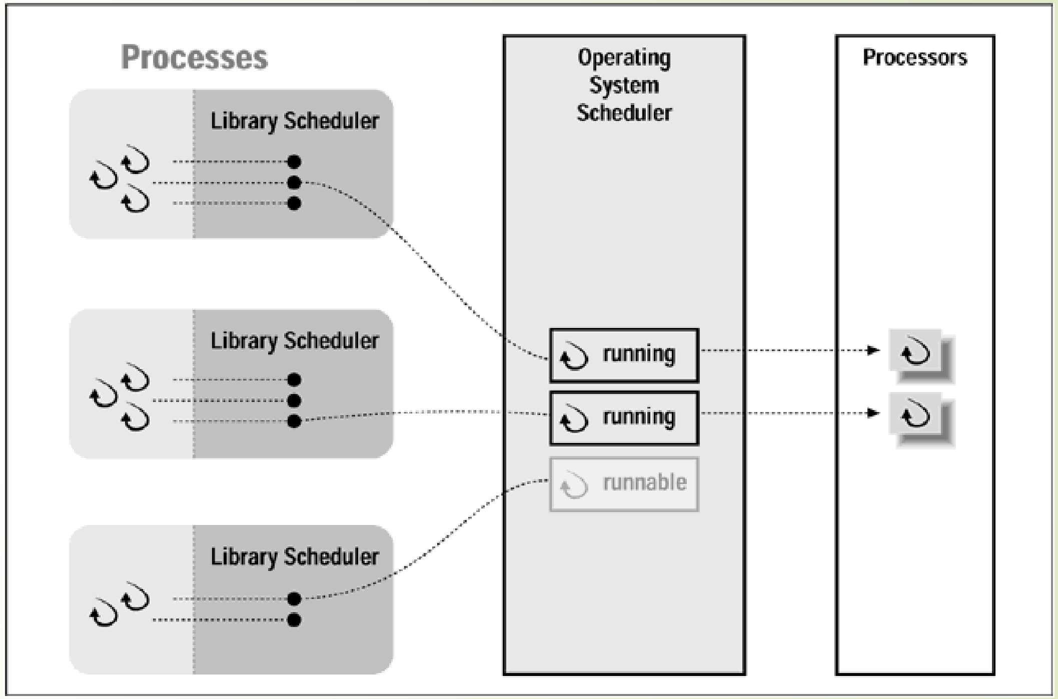
- Improving algorithm – Cannot go beyond a point
- Increasing clock frequency – Not an option any more
- Pipelining efficiency – Can work only up to a point
- Simplest option – Parallelize!

Objectives

- Two classes of architectures available for multi-threading
  - CPU multithreading
  - GPU multithreading
- Technologies available for utilizing any one of them
- Why not use both?
- Devise approach to efficiently divide threads between CPU and GPU
- Utilize both to execute the parallel program efficiently
- Implement the approach and observe the behavior
- Compare with available approaches

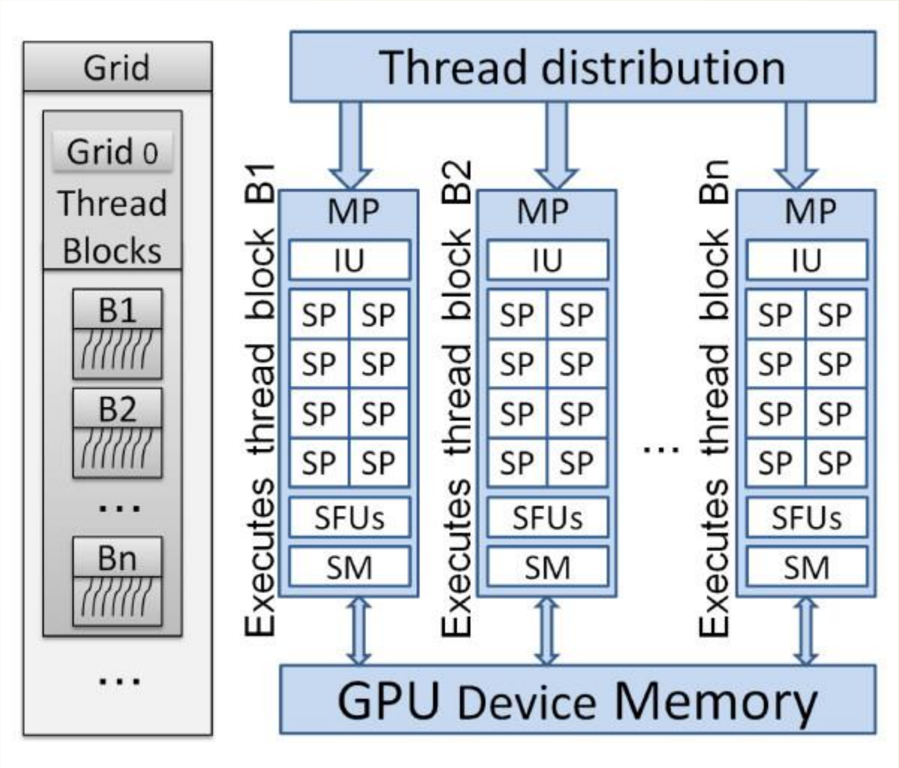
Literature Survey

- Implementing CPU threads – Pthreads, OpenMP, TBB (Thread building blocks), Click++, MPI [1]
- POSIX Threads library - selected to implement CPU threads [4]
- Uses 2-level scheduler for user thread scheduling
- Each user thread mapped to a kernel thread



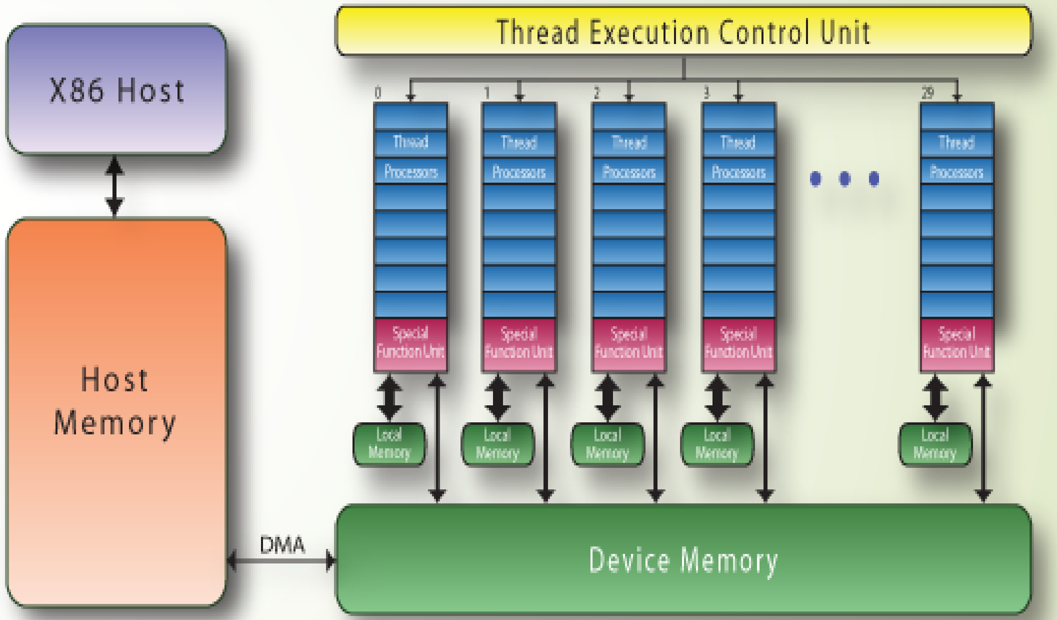
Literature Survey

- Implementing GPU threads – CUDA, OpenGL, OpenCL [2][3]
- CUDA – chosen for the project
- A typical CUDA program:
  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.
- Logical components – Grids, Blocks and Threads
- Physical components – Streaming Multiprocessors, Processor cores



Literature Survey

- Memory model of a GPU
- Types of memory – Global, Block, Thread local
- Speed
  - Thread local > Block > Global
- Size
  - Global > Block > Thread local
- Scope
  - Global > Block > Thread local



Literature Survey

- Parallelizing a program – Decompose in smaller sub-programs
- Sub-programs must be independent – Can be tricky (e.g., array-sum)
- Using Power List data structure [7] – Recursive parallel programs
- Power Lists
  - List of size in power of 2
  - Tie operation - <1,2> | <3,4> = <1,2,3,4>
  - Zip operation - <1,2> | <3,4> = <1,3,2,4>
- Matrix multiplication – Using Power Lists [6]
- Can also find optimum recursion depth using hardware parameters [6]
  - How many blocks?
  - How many threads per block?

Literature Survey

- There exist an optimum division – threads for CPU and threads for GPU [5].
- Intersection of execution time of CPU threads and execution time of GPU threads – the optimum division [5].
- How to find this division at run-time?
- Can optimally divide threads between multiple GPUs [6].
- What if we just assume a CPU to be a GPU?

Discussion of proposed approach

- Assume CPU to be able to execute “k” number of GPU blocks efficiently.
- Calculate optimum recursion depth using GPU hardware parameters.
- CPU threads = K x number of threads per block.
- GPU threads = all remaining threads (if any).
- Get results by CPU threads in pthreads_results.
- Get results by GPU threads in cuda_results.
- Combine the results to formulate final resultant matrix.

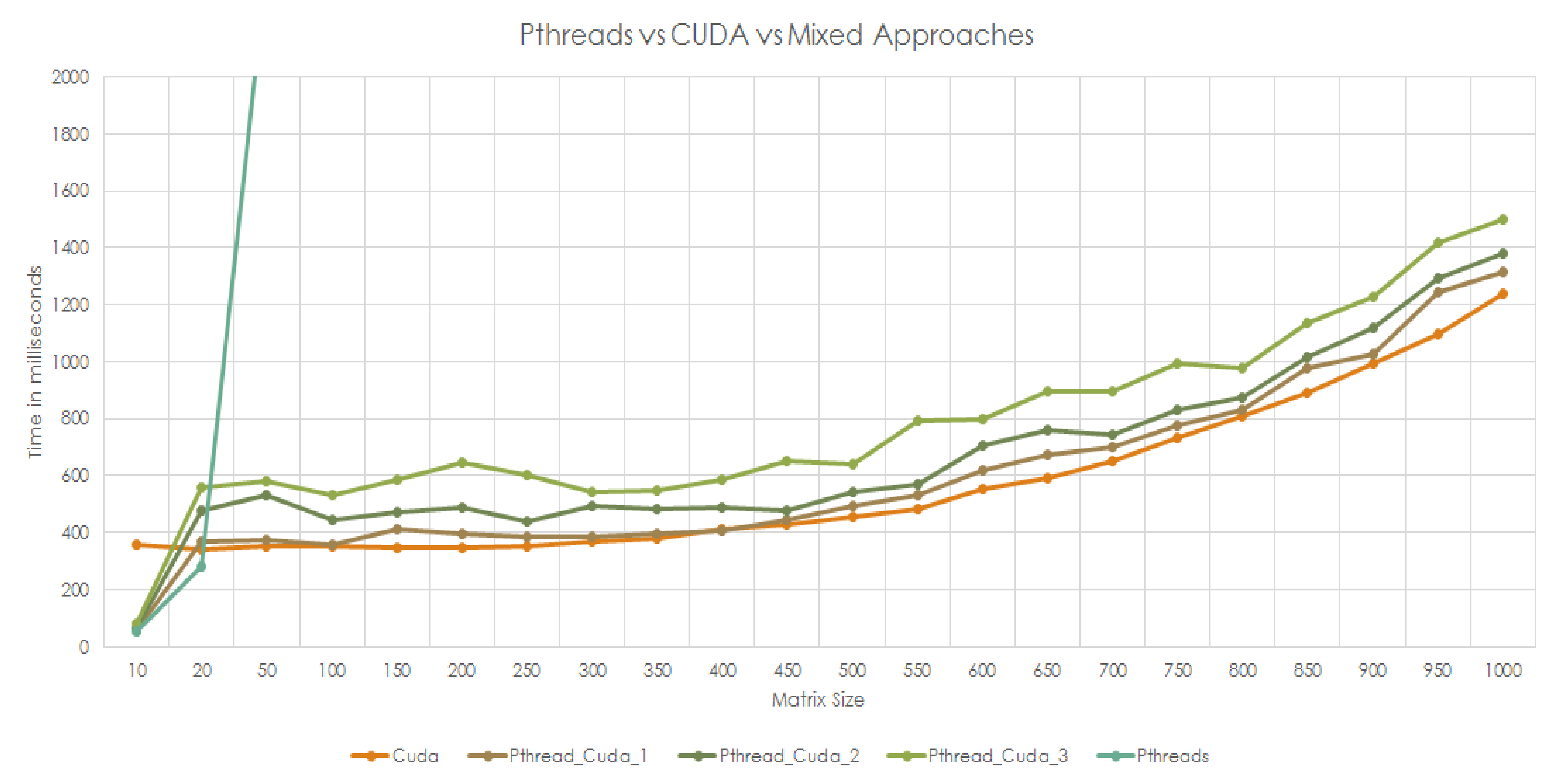
Observations

- Three different generic programs are created.
  - Using pure Pthread.
  - Using pure CUDA.
  - Using mixed approach – divide threads between Pthreads and CUDA.
- Each program takes size of matrix “N” as an input.
- Generates two NxN random matrices.
- Starts a timer.
- Calculates resultant matrix by multiplying the two matrices.
- Stops timer and outputs execution time in milliseconds.

Observations

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

Behavior



Pthreads vs CUDA vs Mixed Approaches
Time in milliseconds
0
200
400
600
800
1000
1200
1400
1600
1800
2000
10
20
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750
800
850
900
950
1000
Matrix Size
Cuda
Pthread_Cuda_1
Pthread_Cuda_2
Pthread_Cuda_3
Pthreads

Concluding Remarks

- Pthreads – Worst of all, as expected!
- CUDA vs Mixed Approaches – quite comparable but CUDA still dominates in most cases, not expected!
- Problem with extra memory management – CUDA manages memory efficiently using DMA.
- Mixed approach memory management time – regular memory management + combining results of Pthreads and CUDA.
- If results by CUDA and Pthreads are written in same matrix instead of different, then might be able to dominate CUDA

Future Scope

- Further optimizing the mixed approach to waste lesser time in memory access.
- Devising better to divide number of threads between CPU and GPU – take CPU hardware parameters also into consideration.

References

1. 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.
2. Woo, Mason, et al. OpenGL programming guide: the official guide to learning OpenGL, version 1.2. Addison-Wesley Longman Publishing Co., Inc., 1999.
3. Sanders, Jason, and Edward Kandrot. CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents. Addison-Wesley Professional, 2010.
4. Nichols, Bradford, Dick Buttlar, and Jacqueline Farrell. Pthreads programming: A POSIX standard for better multiprocessing. " O'Reilly Media, Inc.", 1996.
5. 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.
6. 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.
7. Misra, Jayadev. "Powerlist: A structure for parallel recursion." ACM Transactions on Programming Languages and Systems (TOPLAS) 16.6 (1994): 1737-1767.