Performance Evaluation of CPU-GPU with CUDA Architecture

Conference Paper · July 2015

CITATIONS
0

READS
1,400

2 authors, including:

Chandrashekhar Naikodi
Nitte Meenakshi Institute of Technology
9 PUBLICATIONS **12** CITATIONS

SEE PROFILE

Performance Evaluation of CPU-GPU with CUDA Architecture

Haneesha H K [1], Chandrashekhar B N [2], Lakshmi H [3], Sunil [4]

Hybrid Computing, R&D, Nitte Meenakshi Institute of Technology, Bangalore-64.

Abstract:

Traditional CPU technology, however, is no longer capable of scaling in performance sufficiently to address the required computation power demand. The parallel processing capability of Graphics Processing Unit (GPU) can be exploited by dividing computing tasks into 100s of smaller tasks that canbe run concurrently. This made scientist to address some of the challenging computational problems. In this paper, we had evaluated the CPUs and GPUs performance by considering the few popular Benchmark application like GpuTest and Furmark. GpuTest shows the number of frames that are buffered in CPU and GPU in a given time. And Furmark shows the number of frames that differ with the time to check the performance of CPU with that of GPU. We also experimented with Matrix Multiplication separately on CPU with MPI by varying the number of processors. And on GPU with CUDA C by varying the number of cores. We therefore compare both the CPU and GPU performance.

Keywords: CPU, CUDA, Furmarkbenchmark, GPU, GpuTestBenchmark, Performance evolution

1. Introduction

The world of high performance computing is a rapidly evolving field of study. Many options are open to business when designing a product. GPUs can provide astonishing performance using the hundreds of cores available. The field of high performance scientific computing lies at the crosswords of a number of disciplines and skill sets, and correspondingly, for someone to be successful at using high performance computing in science requires at least elementary knowledge of and skills in all these areas. Computations stem from an application context, so some acquaintance with physics and engineering sciences is desirable. A computational scientist needs knowledge of several aspects of numerical analysis, linear algebra, and discrete mathematics which is an efficient implementation of the practical formulations of the application of parallel computing. Finally, in addition to mastering all these sciences, a computational scientist needs some specific skills of software management.

1.1 CPU

The brain of the computer, processor, central processor or microprocessor, is responsible for handling all instructions it receives from hardware and software running on the computer.

1.2 GPU

A Graphics Processing Unit (GPU) is a single-chip processor primarily used to manage and boost the performance of video and graphics used primarily for 3-D applications. These are mathematically intensive tasks, which would put quite a strain on the CPU. Lifting the burden from the CPU frees up cycles that can be used for other jobs.

1.3 CUDA

CUDA is NVIDIA's parallel computing architecture which enables dramatic increases in computing performance by harnessing the power of the GPU (graphics processing unit).The programmer can choose to express the parallelism in high-level languages such as C, C++, FORTRAN or open standards as OpenACC Directives. The CUDA parallel computing platform is now widely deployed with 1000s of GPU-accelerated Applications shown in Fig.1.

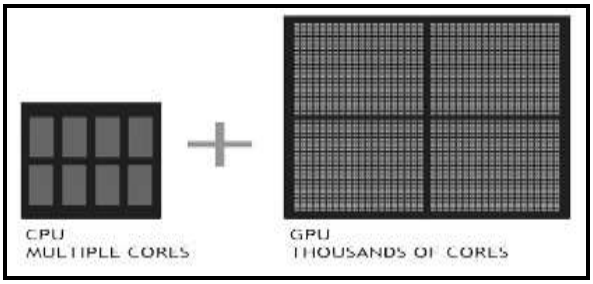

Fig 1: Multi-core Arrangement of CPU-GPU

1.4 Parallel Acceleration

Multi-core programming with 64-bit CPUs is difficult and often results in marginal performance gains when going from single core to 4 cores to 16 cores. Beyond 4 cores, memory bandwidth becomes the bottleneck to further performance increases [1].

To harness the parallel computing power of GPUs, programmers can simply modify the performance critical portions of an application to take advantage of the hundreds of parallel cores in the GPU. The rest of the application remains the same, making the most efficient use of all cores in the system. Running a function on the GPU involves rewriting that function to expose its parallelism, then adding a few new function-calls to indicate which functions will run on the GPU or the CPU. With these modifications, the performance-critical portions of the application can now run significantly faster on the GPU.

2. GPU architecture

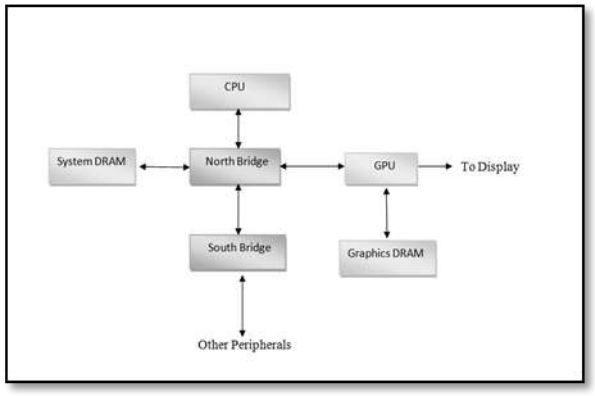

Fig 2: GPU Architecture

The CPU in a modern computer system communicates with the GPU through a Graphic Connector such as a PCI Express or AGP slot on the motherboard, [2] as the graphic connector is responsible for transferring all command, texture, and vertex data from the CPU to GPU. AGP 2x, 4x, and 8x followed, each doubling the available bandwidth, until finally the PCI Express standard was introduced in 2004, with a maximum theoretical bandwidth of 4 GB/sec simultaneously available to and from the GPU. Above Fig.2. Shows the Available Memory Bandwidth in Different Parts of the Computer System.

Component	**Bandwidth**
GPU Memory Interface	35 GB/sec
PCI Express Bus (x16)	8 GB/sec
CPU Memory Interface (800 MHz Front-Side Bus)	6.4 GB/sec

It shows that there is a vast amount of bandwidth available internally on the GPU. Algorithms that run on the GPU can therefore take advantage of this bandwidth to achieve dramatic performance improvements.

3. Programming model of GPU: CUDA

According to the proposed architecture, i.e., the "Performance evaluation of cpu-gpu with cuda *architecture"*, the implementation of the proposed architecture should show the high performance of the GPU when compared to that of CPU. Hence, using the standard performance formula, i.e. the performance is the reciprocal of the execution time.

$$Performance \; \alpha \; (1/Execution\ Time) + communication\_time \quad (1)$$

Where,

$$Execution\ time = \frac{(Process\_start\_time) - (process\_end\_time)}{(Clock\_per\_second)} \quad (2)$$

Applying the same formula to evaluate the performance of the CPU-GPU, we tried to satisfy the above mentioned formula for both CPU and also GPU which are having different configurations and take the readings and representing it through a graph to show the comparison. One of the Theano's design goals is to specify computations at an abstract level, such that the internal function compiler has a lot of flexibility about how to carry out those computations. One of the ways in which the advantage of this flexibility is in carrying out calculations on a graphics card. There are two ways currently to use a GPU, first which only supports NVIDIA cards (CUDA backend) and the second, in development that should any OpenCL device as well as NVIDIA cards (GpuArray Backend). Hence, we used the one which only supports NVIDIA cards.

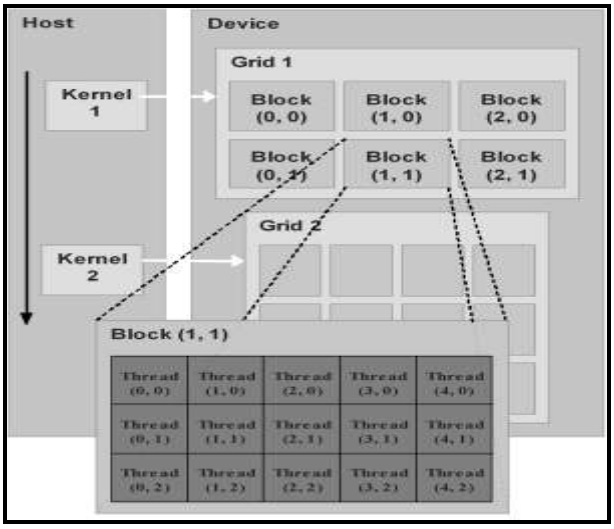

Fig 3: GPU Kernel structure and invocation

The above Fig.3. Shows the representation of the GPU Kernel and the invocation of the kernels. The Fig.3 explains that a single kernel invocation runs the kernel multiple times in parallel on the GPU [3]. Each independent, concurrent execution is called a thread. The number of threads is specified by the developer, by grouping threads in equally-sized blocks, and distributing blocks on a grid. In the experiments we use NVIDIA Tesla M2075 dual-slot graphic card where it contains 448 threads that are again divided into number of threads to run in parallel. Threads in the same block run on the same multiprocessor and can use the same shared memory. Each multiprocessor has a limited number of resources (registers, shared memory). This limits the maximum size of a block.

Kernel invocation:

kernelName<<<gridSize, blockSize>>>(...)

Where:

- gridSize: number of blocks in the grid
- blockSize: number of threads in each block

Grids and block can have one, two or three dimensions, so they are a dim3 type, a structure with 3 unsigned integer components (.x, .y, .z) that default to the value 1 [4]. GPU kernels are defined like C functions with a void return type and a __global__ attribute, and they can use the built-in variables gridDim, blockIdx, blockDim and threadIdx to locally determine the current thread. There are two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces [5]. The global, constant, and texture memory spaces are optimized for different memory usages. Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats. The global, constant, and texture memory spaces are persistent across kernel launches by the same application. Serial code executes on the host while the parallel code executes on the device. The compute capacity of a device is defined by a major revision number and a minor revision number. Devices with the same major revision number are of the same core architecture. The major revision number is 3 for devices based on the Kepler architecture, 2 for devices based on Fermi architecture and 1 for devices based on Tesla architecture. The minor revision number corresponds to an incremental improvement to the core architecture, possibly including new features.

4. Performance analysis

4.1 Platform.

In our experiments the computing is on the CUDA Architecture which is a parallel computing platform and programming model created and developed by NVIDIA Corporation. The CUDA platform is accessible to software developers through CUDA-accelerated libraries, compiler-directives (like OpenACC) and extensions to industry-standard programming languages, including C, C++ and FORTRAN.

4.2 Benchmark Applications with results

Here the performance of MPI-based CPU running on threads is compared with that of CUDA-based GPU running on the NVIDIA Tesla M2075 GPU. The runtime of each core is evaluated on the test dataset using the configuration as described above. The benchmark that is use to compare the performance is “GpuTest” in Linux and “Furmark” in Windows. The Benchmark results show that GPU does provide performance improvement, but not too much, given the fact that it has 448 cores with 24 streamed processors.

4.3 Impact of GPU workload:

The most important factor of performance in heterogeneous CPU/GPU computing is the workload assignment. If we assign too little work to CPU, it is not enough to keep the CPU busy during GPU kernel launch and memory transfer, and thus the latency cannot be well hidden. On the other hand, if we assign too much work to the CPU, when the GPU kernel finishes, it has to wait for the CPU to finish the searching job before generating the resultant vector, which will also result in inferior performance [6]. Therefore the remaining question is: what is the optimal GPU workload for each core searching function under different parallel configurations? To answer this question, we vary the GPU workload from 60% to 100% for each core searching function under two configurations, GPU+1thraed CPU and GPU+2 threads CPU, to check the performance impact. The speedup is measured as:

$$\frac{\textbf{[runtime of X\% GPU workload]}}{\textbf{[runtime of 100\% GPU workload].}}$$

The speedup increases before the optimal workload, and decreases afterwards. For either parallel configuration, core has 3 smaller optimal GPU workload than the other cores**.** This is because it requires wildcard matching, which is fine for CPU parallelization but not good for GPU parallelization; and with more CPU threads, the optimal GPU workload becomes smaller, which is due to the fact the more searching work can be assigned to CPU. Similarly, we can predict that with faster GPU or multi-GPU parallelization, the optimal GPU workload will become larger, since more searching work can now be assigned to GPU.

4.4 GpuTest Benchmark with Results

Geeks3D GpuTest Benchmark is the GPU stress test and OpenGL benchmark. GpuTest is the first public version of a new cross-platform GPU stress test and benchmarking utility. We performed experiments of GpuTest based on OpenGL and is available for Windows (XP, Vista, 7 and 8), Linux and OS X. With graphics card Tesla GPU cluster NVIDIA – Tesla M2075 with 448 cores. The following table shows the test case results for the GpuTest Benchmark which shows the number of frames that are buffered in CPU and GPU in a given time in milliseconds:

Table1: Test case results for GpuTest Benchmark

Time (in ms)	CPU	GPU
10	6	25
20	9	29
30	17	35
40	26	47
50	29	53
60	34	61

The following graph Fig.4. Shows the comparison of CPU and GPU for the Gputest Benchmark:

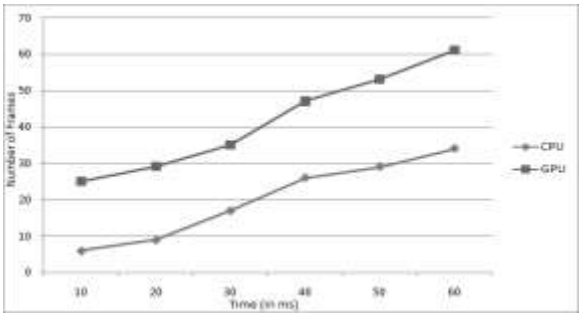

Fig 4: Comparison of CPU-GPU with GpuTest

On Windows and OS X, GpuTest comes with a graphical user interface (GUI) called "Furmark" to launch the different tests whereas on Linux, a set of scripts is provided to launch the tests.

4.5 Furmark Benchmark with Results

FurMark is a very intensive OpenGL benchmark that uses fur rendering algorithms to measure the performance of the graphics card. Fur rendering is especially adapted to overheat the GPU and that's why FurMark is also a perfect stability and stress test tool (also called GPU burner) for the graphics card. The benchmark offers several options allowing the user to tweak the rendering: full screen / windowed mode, MSAA selection, window size, duration. This benchmark is experimented on OpenGL 2.0 withWindows (XP, Vista, 7 and 8), complaint graphics card: NVIDIA GeForce 5/6/7/8 (and higher), AMD/ATI Radeon 9600 (and higher) or a S3 Graphics Chrome 400 series with the latest graphics drivers. The start-up interface allows you to tweak the benchmark features such as:

- Benchmark mode or stability test mode (for overclockers)
- GPU temperature monitoring and recording in a file
- Window size selection (standard or custom)
- MSAA samples selection
- Benchmarking parameters: time based or frame based

The results are analysed by running the same benchmark on CPU and also the GPU, plot the appropriate graphs for the number of frames that differ with the time to check the performance of CPU with that of GPU. The experiment is repeated for many times and then the graph is plotted by taking the average of all the readings for the particular time steps. The graph which is plotted with these time steps clearly shows the difference between the CPU performance and the GPU performance. Upon running the benchmark in which the time is preset for around 60000 ms, it shows the number of frames per second along with the time in millisecond, also the minimum number of frames per second and maximum number of frames per second, and at the end of the pre-set time, it calculates the average number of frames per seconds in time. The following table lists the number of frames per second for CPU and GPU respectively.

Table 2: Test case results for Furmark benchmark

Time (in ms)	CPU (Frames per Second)	GPU (Frames per Second)
10	2254	3082
20	4501	6156
30	6748	9233
40	8993	12314
50	11239	15391
60	16582	18466

The following graph Fig.5. Shows the Furmark comparisons of CPU and GPU:

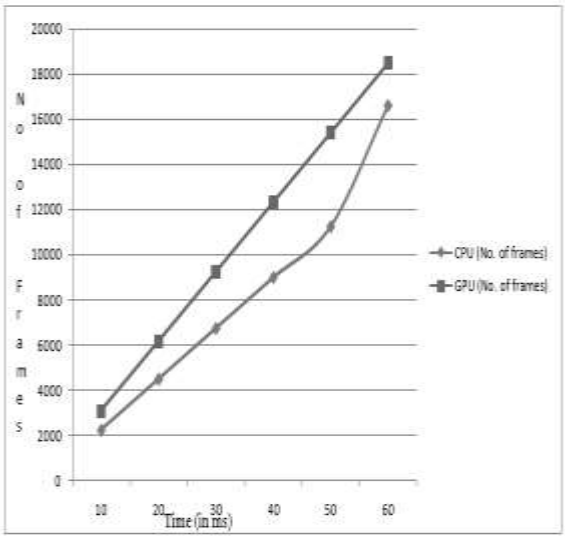

Fig 5: Comparisons of CPU and GPU with Furmark

4.6 Matrix Multiplication with Results

The major application that we considered to evaluate the performance is the Matrix Multiplication program, [7] which is implemented in CPU with MPI programming in which the number of processors are varied to calculate the time taken for various set of matrix data with a large number of rows and columns, whereas it is implemented in GPU with CUDA C/C++ programming which runs on NVIDIA Tesla graphic card in which the number of cores are varied to calculate the time taken for various set of matrix data with a very number of rows and columns. The following table shows the test case results of Matrix Multiplication on CPU with various matrix size by varying the number of processors:

Table 3: Test case results for Matrix multiplication on CPU

Matrix Size / Processors	200*200	400*400	600*600	800*800	1000*1000
2	0.46	3.74	12.56	48.62	97.63
4	0.12	1.23	9.63	30.30	59.48
8	0.10	1.02	6.12	19.14	47.54
16	0.07	0.16	3.34	9.86	13.73

The following graph Fig.6.Shows matrix size versus the time taken on CPU with 4 processors:

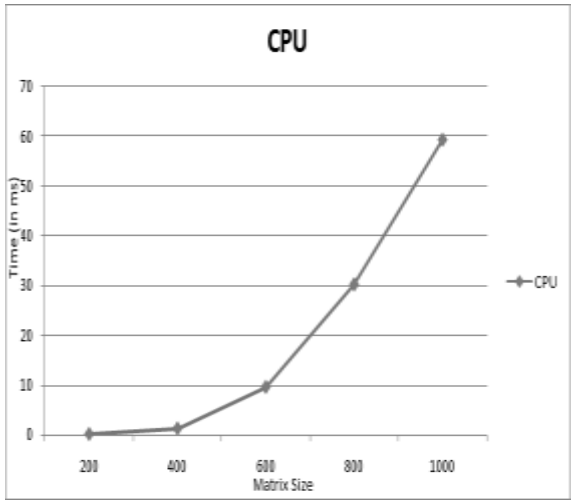

Fig 6: Matrix Multiplication on CPU

The following table shows the test case results of Matrix Multiplication on GPU with various matrix sizes by varying the number of cores in Graphic card:

Table 4: Testcase results for Matrix multiplication on GPU

Matrix size / Cores	1000*1000	2000*2000	3000*3000	4000*4000
100	9.178	81.419	331.361	817.255
200	8.946	81.152	354.745	777.089
300	9.124	81.173	333.788	760.036
400	8.998	81.280	330.569	807.365
448	9.114	81.617	336.744	784.841

The following graph Fig.7. Shows matrix size versus the time taken on GPU with 300 GPU cores:

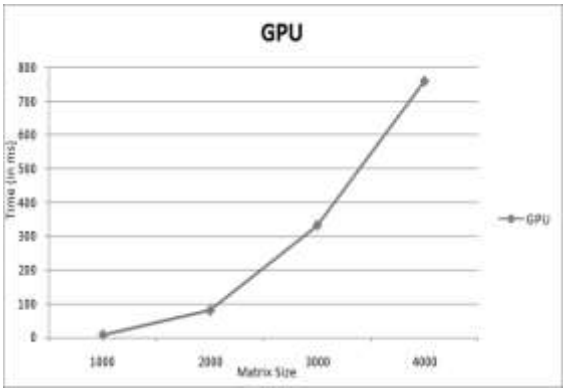

Fig 7 : Matrix multiplication on GPU

Conclusion

High Performance Computing is a rapidly growing field that will require more research to understand. The technology surrounding CPUs and GPUs is rapidly evolving and will continue in future years. Benchmarking will be a constant process to satisfy different systems as well as different devices. With the information in this report, some light is shed on the processing power for different applications between CPUs and GPUs [8]. These results are useful to compare the two systems discussed as well as in comparison with other devices during future studies. The world of High Performance Computing is constantly evolving field that will play a significant role in the years to come in many diverse fields. Hence, the effort showed that the performance of the GPU when compared to that of CPU is always step ahead in all the applications.

Future work

Based on our investigations, the following future work is suggested:

- Use of multiple streams on a single GPU so that the GPU can perform the kernel computation and memory transfer in an asynchronous way in order to further hides the latency.
- Trying multi-GPU parallelization and verifying the impact of no GPUs on the optimal GPU workload.
- Trying GPU parallelization in the other levels of the code thus lowering the communication-to-computation ratio (current GPU parallelization is on the lowest level – core searching functions).

References:

1. Theodre, B., Tabe. : "The use of the MPI Communication Library in the NAS Parallel Benchmarks". IEEE Computer Society, IEEE.

2. AmanMadaan, Dr. Sunil, K., Singh and Ankur Aggarwal. :"Junk Computing: Performance Evaluation and Comparison of an MPI based Heterogeneous Cluster". International Journal of Advanced Research in Computer Science and Software Engineering, 2013.

3. Reiji, Suda., and DaQiRen.: "Accurate Measurements and Precise Modeling of Power Dissipation of CUDA Kernels toward Power Optimized High Performance CPU-GPU Computing". University of Tokyo, 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies.

4. Hari, K., Raghavan, Satish, S. Vadhiyar.: "Efficient asynchronous executions of AMR computations and visualization on a GPU system". IISc, Bangalore ,Published in the Journal of Parallel and Distributed Computing. J.ParallelDistrib. Comput. 73(2013) 866-875.

5. Sang-Min, Song., Young-Min, "GPU-driven Parallel Processing for Real-time Creation of Tree Animation". Department of Game Engineering Tongmyong University. International journal of Software Engineering and Its Applications, Vol.8, No.6 (2014), pp.183-194.

6. Pennycook, S. J. et al,: "Performance Analysis of a Hybrid MPI/CUDA Implementation of the NASLU Benchmark". Technical Report, Oxford eResearch Centre University of Oxford, UK, 2010.

7. DaQiRen, Reiji, Suda., : "Power Efficient Large Matrices Multiplication by Load Scheduling on Multicore and GPU platform with CUDA". Department of Computer Science, The University of Tokyo, Hongo, Bunkyo-ku, Tokyo, 113-003, JAPAN,

8. SivagamaSundari, M., Sathish, S. Vadhiyar., Ravi, S. Nanjundiah., "Large improvements in application throughput of long-running multi-component applications using batch grids". Supercomputer Education and Research Centre and Centre for Atmospheric and Oceanic Sciences, Indian Institute of Science, Bangalore, India. Published online 20 December 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1878