Performance comparison of CPU and GPU on a discrete heterogeneous architecture

Winnie Thomas, Rohin D. Daruwala
Department of Electrical Engineering
Veermata Jijabai Technological Institute
Mumbai, India.
winniethomas@ieee.org, rddaruwala@vjti.org.in

***Abstract*—Today Graphics Processing Units (GPUs) in scientific computing have led the computing system to achieve tera-scale computing power to the laptops and peta-scale computing power to the clusters by combining multicore Central Processing Units (CPUs) and many core GPUs which can be called a heterogeneous computer architecture. This paper describes briefly an evolutionary journey of GPUs. For performance comparison, parameters considered are latency and throughput. So based on the execution time of a GPU and CPU for a given task, written with Compute Unified Device Architecture (CUDA) C language, the two parameters are measured with increasing size of workload. When the task size is increased GPU is found to be approximately 51% faster than the multithreaded CPU when GPU achieves 100% occupancy. Throughput of GPU is found to be 2.1 times higher than that of CPU for large task size. The GPU used is NVIDIA's GeForce GT630M with CPU of Intel's i-5 3210M 3rd generation processor.**

Keywords—CPU, GPU, CUDA, execution time, throughput, thread, speed up, mulitocore.

I. Introduction

Traditionally most of the programs are written in sequential manner. A sequential program will run on only one CPU and will not become faster than the most powerful CPU in use today. This is a huge obstruction for application developers because they will not be able to introduce different and new features to their software. Most of the software developers have relied on the improvement in hardware to increase the speed of their applications under the hood. This trend has slowed since 2003 due to energy-consumption and heat-dissipation issues that have limited the increase of the clock frequency and the level of productive activities that can be performed in each clock period within a single CPU [1].

Parallel programming is the only way that will give room for the performance improvement of applications. In a parallel programming model multiple threads of execution cooperate to complete the work faster. A thread here can be considered as a logical Von Neumann Machine with a processing core with its ALU, control unit and memory. We have in today's date such a massively parallel model in the form of Graphic Processing Units (GPUs). GPUs use hundreds of processor cores in parallel to execute thousands of parallel threads. These threads work on problem with inherent parallelism.

This paper describes the evolution of GPU; and program pieces to study and compare process times and thus speed up and throughput of CPU and GPU. The program was written in CUDA C language and has lead to more insight into some typical underlying architectural behavior of the GPU device for the given program.

II. Graphic Processing Unit (GPU)

A. Evolution of GPUs

First, The demand for very high quality real time graphics in computer applications has been the inspiration behind the advancement of graphics hardware. For e.g. rendering high definition complex 3D scenes at an ever increasing resolution is a challenge in electronic gaming application. When there were no GPUs in 1990s, Video Graphics Array (VGA) controllers were used in PCs to accelerate graphical user interface. VGAs generated 2D graphics displays.

A graphic programmer writes a single thread program that draws one pixel and GPU runs multiple copies of this program (thread) in parallel drawing multiple pixels in parallel. Now graphic programs written in C or C++ with the CUDA model scale transparently [10]. Scalability has enabled GPUs to rapidly increase their parallelism and performance with increasing transistor density as GPU transistor counts are increasing exponentially doubling every eight months [2].

The modern GPU started its journey from being a fixed function pipelines to micro-coded processors, configurable processors, programmable processors to today's scalable parallel processors. The first GPU introduced was GeForce 256 by NVIDIA in 1999 [5]. It was a single chip 3D real-time processor. It contained a configurable 32-bit floating point vertex transform, lighting processor and a configurable integer pixel-fragment pipeline programmed with OpenGL and Microsoft DirectX7 APIs. This GPU used floating point arithmetic to calculate 3D geometry and vertices first, to be applied it to pixel lighting and color values then to handle high dynamic range scenes [13].

In 2001, NVIDIA GeForce introduced General shader programmability there by allowing the application developer to work with instruction of the floating point vertex engine [2]. These programmability and floating point capability, extended to the pixel shader stage and made texture accessible from the vertex shader stage, e.g. ATI Radeon (2002), featured a programmable 24 bit floating point pixel shader processor programmed with DirectX9 and OpenGL. GeForce features 32 bit floating point pixel processors. GeForce 6800 [1,9] and 7800 [1] series introduced separate dedicated vertex and pixel processors. In 2005, Xbox 360 GPU achieved unified processing by allowing vertex and pixel shader to execute on the same processor.

GeForce 8800 GPU introduced in 2006, featured an array of unified processors. The unified processor supports dynamic partitioning of the array of processors to vertex shading stage, geometry processing (first introduced in GeForce 8800 GPU) and pixel processing stage.

B. General Purpose Computing on Graphic Processing Unit (GPGPU)

The GPUs were only capable to process graphic data. GPGPU allows the utilization of a graphics processing unit (GPU), to perform computation in applications traditionally handled by the central processing unit (CPU). To access the computational resources the programmer had to use OpenGL or DirectX API calls [4]. The NVIDIA Tesla GPU architecture designers replace shader processors with fully programmable processors with instruction memory, cache and instruction sequencing control [8, 14]. With NVIDIA developing CUDA C/C++ compiler libraries also, by now programmers can easily access the GPU. NVIDIA introduced Fermi GPU computing architecture in 2009 [5]. It increased double- precision performance, error correcting code (ECC) memory protection for large scale computing, 64 bit unified addressing, cached memory hierarchy and instruction for C, C++,Fortran ,OpenCL and DirectCompute .

In 2010 September, NVIDIA introduced Kepler architecture, which added new features of dynamic parallelism and Hyper Q [6]. Dynamic parallelism allows GPU to generate work for itself and to schedule that work through the best hardware path, without involving the CPUs. Hyper Q allows multiple CPU cores to call single GPU thereby dramatically increasing GPU utilization and significantly reducing CPU idle times [6].

III. Compute Unified Device Architecture (CUDA)

CUDA is software and hardware architecture for supporting heterogeneous parallel computing. It enabled the GPU to be programmed with a variety of high level languages. The programmer could now write C programs with CUDA extensions and target a general purpose, massively parallel processor. To a CUDA programmer, the computing system comprises a host which is a traditional CPU (such as an Intel architecture microprocessor in personal computers) and GPU(s) which are massively parallel processors with large number of arithmetic execution units .The program section often consist of some data parallelism, that allow many arithmetic operations to be safely performed on streaming data in a simultaneous manner. Streaming data can be considered as a stream of data elements that are required to be processed by same task or instruction based on Single Program Multiple Data (SPMD) [1] which is a data parallel model [3]. The CUDA devices accelerate the execution of these applications by harvesting a large amount of data parallelism. Images and video frames are photos or snapshots of the real world, in which different parts of picture, capture independent and simultaneous events [1]. Rigid body physics and fluid dynamics model natural forces and movements that can be independently evaluated .Such independent evaluation is the basis of data parallelism in these applications. CUDA device can significantly accelerate the execution of these data parallel part of application over a traditional CPU.

A. CUDA Program Structure

A CUDA program is a unified source code that comprises both host (CPU) and device (GPU). It consists of one or more portions that exhibit little or no data parallelism, and are implemented in host code and the portions of the program that exhibit rich amount of data parallelism are implemented in the device code. The NVIDIA C compiler separates the two during the compilation process. The host code is written in C or C++ language with keywords for labeling data parallel function called kernels and their corresponding data elements. The kernel functions typically generate a large number of threads to exploit data parallelism. These CUDA threads are of lighter weight than the CPU threads. These threads take few cycles to generate and schedule which is in contrast with CPU threads that typically take thousands of clock cycles to generate and schedule.

The program execution always begins with host (CPU) execution. The execution is moved from host to device (GPU) when a kernel function is called. All the threads that are generated due to the launch of kernel are called a grid [1]. When all the threads complete their execution, the grid formed by threads also ends for that kernel. The remaining non kernel part of the program is executed on host till the next kernel is called by host.

B. CUDA Thread Hierarchy

The threads on CUDA are organized into a hierarchy of threads, thread blocks and grid of blocks as shown in Fig. 1. Once a kernel is called or launched, the grid corresponding to the threads is generated. To assign the threads to execution resources, they are divided into blocks. In Fermi architecture the execution resources are in the form of streaming multiprocessors abbreviated as SMs. An SM consists of 8 or more streaming processors (SPs) also called as cores. For e.g., the NVIDIA GT630M GPU used for experimentation has two SMs shown in Fig. 2. Each SM consists of 48 cores and 1 core processes single block. Hence both SMs can service 96 blocks at a time in GT630M as long as sufficient resources are available for all the thread blocks. If the available resources do not suffice to the need of all 48 blocks per SM, CUDA runtime system automatically reduces the number of blocks per SM. The runtime system keeps the record of the blocks that are needed to be executed and as soon as the previous blocks are serviced, the new blocks are assigned or mapped to SMs.

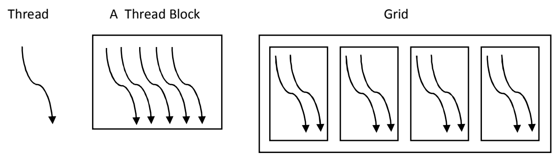


Fig. 1: CUDA Thread Hierarchy

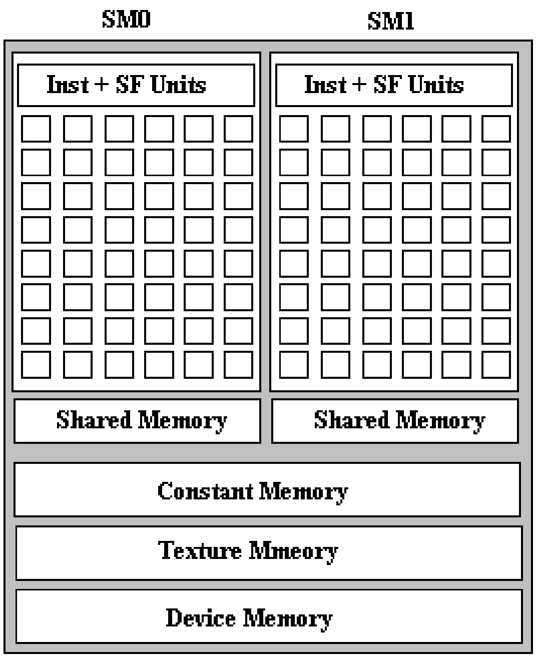


Fig. 2: Each SM has 48 cores or streaming processors in GT630M

In Fermi architecture, once a block is assigned to an SM, it is further divided into 32 threads units called warps so that the execution of a thread block is divided into warp execution. The core in the SM services a block in warp by warp basis.

The advantage of warp execution is that if a warp is waiting for the result of previous operation with long latency, this warp is not selected. Next or another warp which is not waiting for any previous operation result is brought into the core for execution. This mechanism is called latency hiding.

IV. EXPERIMENT WALKTHROUGH

Fig. 3 shows the implementation of a routine to increment each element of an array of certain length. Fig. 3a shows sequential and Fig. 3b shows parallel implementation of the routine to be executed on the CPU and CUDA device respectively. The host used for experimentation is 3rd generation i5-3210 processor, with 2 cores and with the memory of 4 GB RAM. The CUDA device used is Fermi Architecture Based NVIDIA GT630M and has dedicated memory of 2GB RAM. The overall architecture is a discrete heterogeneous architecture, where in CPU and GPU or any other processors are connected through PCI bus on different chips each with their respective global memories. All the cores of the CPU while running task were active so that the performance comparison of GPU and CPU is fair as much as possible .CPU supports multithreading with 4 threads and each core operates at 2.5 GHz. GPU operates at 0.95 GHz.

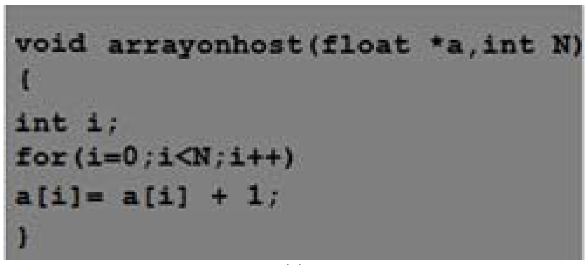
The objective of this routine is to measure the process time taken by the CPU and the GPU and to compare their execution time and throughput by recording the speed up. Throughput of a device is the number of tasks it performs in unit time. The elements of array are generated in host with a simple *for loop*. The length of the array is varied. The kernel for GPU is called by the host (CPU). Block size is also varied to analyze the impact of number of threads per block on performance and ability of the CUDA GPU.

For the device grid size is implemented with the equation

Gridsize =(N/BlockSize)+(N%BlockSize==0?0:1)

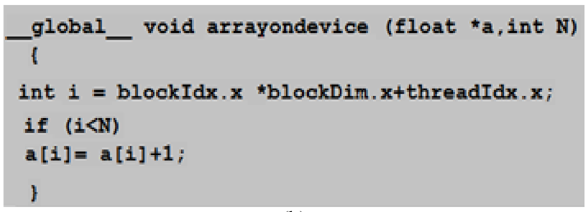
where N is the number of data elements, BlockSize is the number of threads per block. In cases when N is not divisible by BlockSize, the last term in the GridSize equation adds one extra block (which for some cases implies that some threads will not do useful work).

The host sequential routine and kernel function are called 100 times.and time of computation is recorded each time. The average of all 100 time periods are taken. The function *arrayonhost* in Fig. 3a is a *for loop* that increments one array element per iteration by adding 1 to each element..Whereas the kernel function *arrayondevice* in Fig. 3b, which is a parallel implementation, increments all elements simultaneously since each iteration is independent. The kernel assigns each iteration to a separate thread to compute the result. The keyword *__global__* in Fig. 3b indicates that the function *arrayondevice* is a kernel and it can be called by host through *arrayondevice<<<Gridsize, BlockSize>>>*to generate the grid of *GridSize* number of blocks and blocks of *BlockSize* number of threads. Since all the threads execute the same kernel, there needs a mechanism to distinguish themselves, so that they work with their set of designated data. All threads have their own unique global index values [1, 11].



```
void arrayonhost(float *a,int N)
{
int i;
for(i=0;i<N;i++)
a[i]= a[i] + 1;
}
```

(a)



```
__global__ void arrayondevice (float *a,int N)
 {
int i = blockIdx.x *blockDim.x+threadIdx.x;
 if (i<N)
 a[i]= a[i]+1;
 }
```

(b)

Fig. 3: a) Host Routine for CPU, b) Kernel for GPU

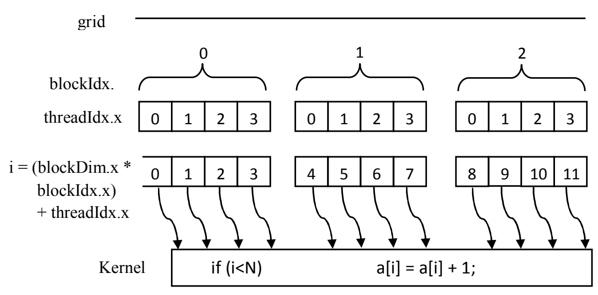


Fig. 4: CUDA Thread Organization

For that in CUDA, each thread has a *blockIdx* value, *threadIdx* value and *blockDim* value. *blockIdx* is the integer block index as shown in Fig. 4, *threadIdx* is the thread index within its block and *blockDim* is the number of threads per block[12]. If grid and its blocks are organized as 2D arrays, the kernel function can form a unique global thread index value in x-dimension as *blockIdx.x*blockDim.x+threadIdx.x*.

There is no inherent data partitioning when a kernel is launched so initially a random number of block size (*blockDim=4*) was chosen and time was measured starting with 100 elements to 10 crore elements. But it was found that both the kernel (GPU) and function (CPU) do not give correct output beyond a limit.

A. *Analysis on the limit of maximum data elements that give valid result on GPU*

Since block size chosen was 4, the kernel computed correct result for 262140 data elements. On further trial it is found that as the block size doubles the maximum number of input data elements for which GPU gives correct output (of addition) also doubles Fig. 5. In GT630M the maximum grid size in X dimension is 65535 (given in specification of device as 2^16-1). From the graph it can be observed that the for this task maximum number of inputs for which GPU gives correct result, let us call it M_{GPU}, is

M_{GPU}= *(Max. X grid dimension)* block size*

For e.g. if block size is 512 then

M_{GPU} = 65535*512=33553920

If N is number of data elements, For N> M_{GPU}, the GPU fails to give correct output for the block size.

B. *Analysis Analysis on the limit of maximum data elements that give valid result on CPU*

For the routine shown in Fig. 3a, where each data element is incremented by an index of "1", the specified CPU does not give correct result beyond 16777216 data elements. When varied the value of the index to be added to the data element, it was found that the maximum number of data elements depend on the value of index.

Let us call the maximum number of data elements for CPU as M_{CPU} and index value as *n*, then

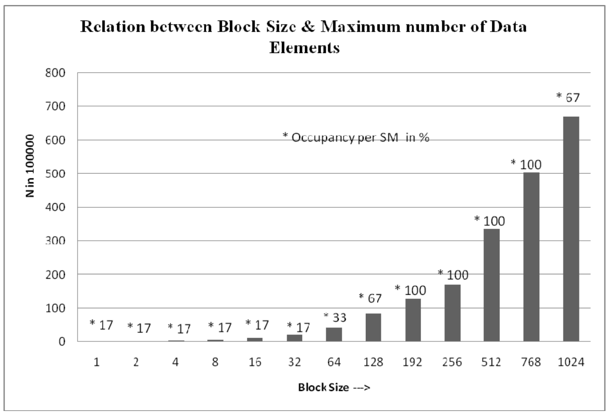


Fig. 5: Block Size and maximum number of input data relation

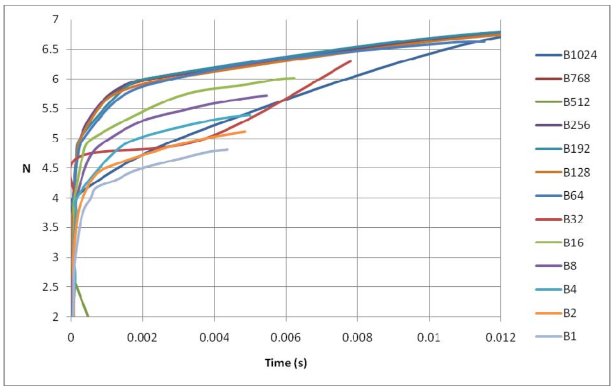


Fig. 6: GPU Process time for all blocks

$$M_{CPU} = (2^{\wedge}24)-(n-1) \quad (n\geq 0)$$
$$= (2^{\wedge}24+1) \quad (n\leq 0)$$

It was verified that for 0 and negative index values the M_{CPU} is constant which is (2^24).

Thus when GPU is configured to block size of 1024 we get $M_{GPU} =$ 1024*65535= 67107840, which is 4 times greater than M_{CPU} for the index value 0.

C. *Comparison of the execution time of GPU and CPU*

The time taken to add elements were recorded for the block sizes of 1 (M_{GPU} =65535), 2 (M_{GPU} =131070), 4 (M_{GPU} =262140), 8 (M_{GPU} =524280), 16 (M_{GPU} =1048560), 32 (M_{GPU} =2097120), 64 (M_{GPU} =4194240), 128 (M_{GPU} =8388480), 192 (M_{GPU} =12582720), 256 (M_{GPU} =16776960), 512 (M_{GPU} =33553920), 768 (M_{GPU} =50330880) and 1024 (M_{GPU} =67107840).

1024 is the maximum number of threads as per the specification of the device (GT630M). The time periods of all the block sizes are plotted in Fig. 6. Y axis is in logarithmic scale. These plots are compared with the time taken by CPU for set of 100 to almost (65535*1024) elements. From readings it is observed that for block sizes 1 to 128 the CPU outperforms the GPU and for 256 and 512 when N is increased above 10000 elements GPU performed better.

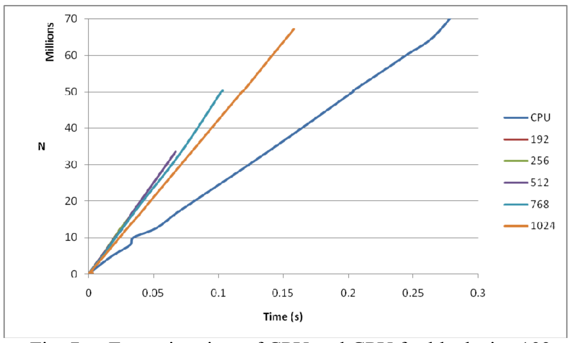


Fig. 7: Execution time of CPU and GPU for block size 192, 256, 512,768 and 1024

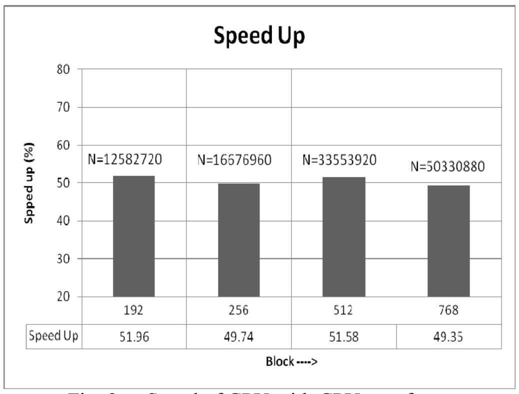


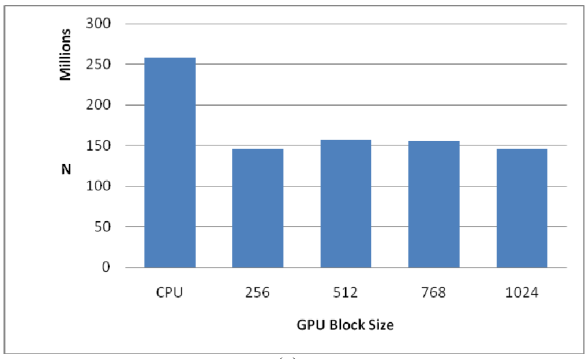
Fig. 8: Speed of GPU with CPU as reference

With the help of CUDA occupancy calculator [7] it is found that for this kernel in which each thread uses 4 registers and 24Bytes of shared memory, 100% occupancy can be obtained not only at block sizes 256 and 512 only but for 192 and 768 also. Occupancy implies how many cores (SPs) per SM is utilized. In this case 100 % occupancy implies that all 48 cores (SP) per SM are used when kernel is launched. For block size 1024 the occupancy was 67% and for rest of block sizes the occupancy is indicated in Fig. 5. So in Fig. 7 CPU speed up is compared with four block sizes of 192, 256, 512, 768 and also with 1024 block size since M is largest for 1024. Execution time of CPU was measured beyond N=M_{CPU} =16777216 as it was giving a linear increase in the amount of execution time as N increased. From the readings it is inferred that when the workload is small i.e. when N is small CPU performs much better than GPU. The time taken by the GPU starts decreasing with respect to CPU as the size of N is increased which is apparent from Fig. 7. The speed up of 51% was achieved when N=M_{CPU} for all four chosen block sizes which is plotted in Fig. 8. Speed up here is

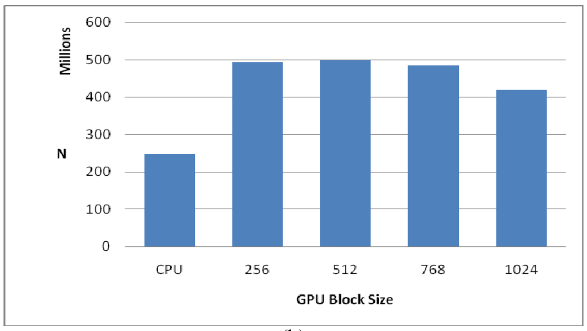
$$\text{speed up} = (CPUt - GPUt)/CPUt \times 100\%$$

Where GPUt and CPUt are time taken for computation by GPU and CPU respectively. Thus when the maximum occupancy is achieved in device, the device will outperform the host as the size of workload is increased.

D. *Comparison of throughput of GPU and CPU*



(a)



(b)

Fig. 9: a) Number of input elements N= 10,000 b) Number of input elements N= 16777216

From the recorded *readings* the maximum number of input elements that give valid output from the CPU is found to be 16777216. Whereas for GPUs if it is configured with block size of 1024 the number of data elements that can be processed in single dimension is 67107840, which is 4 times greater than that of CPU. Clearly in terms of throughput also this is an advantage with the GPU.

Two figures are shown here first for N=10000 in Fig. 9a and N= 16777216 in Fig. 9b. Only block numbers 256,512,768 and 1024 are used in both the figures to compare throughput. For block sizes 1 to 192 the M_{GPU} is less than 16777216 (M_{CPU}). It can be seen than in Fig. 9a for smaller workload CPU throughput is slightly more and when the size of workload is high throughput of GPU is approximately 2.1 times higher than that of CPU.

V. CONCLUSIONS AND FUTURE SCOPE

From the observation it can be concluded that the GPU used for the routine of simple addition presented larger range of number of valid computation than that of CPU. When GPU was configured to maximum block size it presented a 4 times larger range than the CPU. A relationship between block size (number of threads per block) and the capacity of GPU is established and verified. The verification and analysis of relation between index number for this routine and the maximum limit of the CPU to give valid result for is left for future work.

In terms of time taken to execute the given routine, for smaller workload CPU performs better than the GPU. As the size of workload increased in which number of data elements are increased the execution time of GPU starts reducing with respect to that of CPU. GPU performed the addition task 51% faster than CPU for 100% occupancy.

In terms of throughput i.e. the number of tasks performed per second, GPU has always fared better than CPU for many applications. CPU has few cores which are very powerful in terms of latency or speed compared to 100 weak cores of GPU. If both CPU and GPU are used then the high latency problem due to sequential program can be taken care by CPU and the problem of throughput can be handled by GPU by offloading the data parallel part of the program to GPU.

As GPU architecture will continue to evolve, GPU cores will not become CPUs, rather GPUs will be optimized for throughput rather than latency.

REFERENCES

[1] Kirk, David B., and Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Amsterdam: Elsevier/Morgan Kaufmann, 2012.

[2] Nickolls, John, and William J. Dally, "The GPU Computing Era.", In IEEE Micro 30.2, 2010, pp. 56-69.

[3] Bhujade, Moreshwar R, Parallel Computing, Tunbridge Wells, UK: New Age Science, 2009, pp. 33-34.

[4] McReynolds, Tom and Blythe, David and Grantham, Brad and Nelson, Scott., Advanced graphics programming techniques using OpenGL, *Siggraph 1998 Course Notes*. Citeseer, 1998.

[5] "NVIDIA Unveils next Generation CUDA GPU Architecture--codenamed 'Fermi'." In Advanced Imaging, Oct. 2009.

[6] NVIDIA "NVIDIA's Next Generation CUDATM Compute Architecture: Kepler TM K110", http:// www.nvidia.com/ content/ PDF/kepler/ NVIDIA-Kepler-GK110- Architecture-Whitepaper.pdf Web. 07 July. 2013.

[7] NVIDIA, CUDA, GPU Occupancy Calculator, *CUDA SDK*, 2010.

[8] E. Lindholm et al., ''NVIDIA Tesla: A unified graphics and computing architecture,'' IEEE Micro, vol. 28, no. 2, 2008, pp. 39-55.

[9] J. Montrym and H. Moreton, ''The GeForce 6800,'' IEEE Micro, vol. 25, no. 2, 2005, pp. 41-51.

[10] Khronos, The OpenCL Specification, 2009, http:// www.khronos.org/ OpenCL.

[11] M. Garland et al., ''Parallel Computing Experiences with CUDA,'' IEEE Micro, vol. 28, no. 4, 2008, pp. 13-27.

[12] Stratton, John A and Stone, Sam S and Wen-mei, W Hwu, MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs, *Languages and Compilers for Parallel Computing*. N.p.: Springer, 2008, pp. 16-30.

[13] Wu, Enhua and Liu, Youquan. "Emerging Technology about GPGPU." *Circuits and Systems*. Proc. of IEEE Asia Pacific Conference on Circuits and Systems. N.p.: IEEE, 2008, pp. 618-22.

[14] J.H. Huang, "2009: The GPU Computing Tipping Point,'' Proc. IEEE Hot Chips 21, 2009, http://www.hotchips.org/archives/hc21.