International Journal of Recent Technology and Engineering (IJRTE)
ISSN: 2277-3878 (Online), Volume-8, Issue-1, May 2019

A Performance Improvement Approach for CPU-GPU Heterogeneous Computing

Raju K, Niranjan N Chiplunkar

Abstract: ***The heterogenous computing system involving Central Processing Unit and Graphics Processing Unit (CPU-GPU) is widely used for accelerating compute intensive applications that exhibit data parallelism. In CPU-GPU execution model, when the GPU is performing the computation the CPU cores remain idle, wasting enormous computational power. The performance of an application on GPU can be further improved by efficiently utilizing the computational power of CPU cores along with that of GPU. In this paper we propose an approach to simultaneously utilize computational power of both CPU and GPU to perform a task. We execute different independent data parallel portions of an application concurrently on CPU and GPU. We use CUDA framework to execute the task on GPU side and POSIX threads (Pthreads) to execute the task on CPU side. Through several experiments we demonstrate that by judiciously allocating different kernels to suitable processors and executing them concurrently, our approach can improve the performance of a CUDA based application compared to the GPU-only execution of that application.***

Index Terms: Heterogeneous computing; GPU; Multicore CPU; Concurrent execution; CUDA kernels.

I. INTRODUCTION

The CPU-GPU heterogeneous system provides a powerful architecture for accelerating computation-intensive data parallel applications. The availability of thousands of cores in a GPU makes it a suitable architecture for executing applications that exhibit massive parallelism. With thousands of cores a GPU can improve the performance of parallel applications. Also, the modern CPUs possess multiple processor cores providing huge computational power. GPUs were originally developed for rendering images. However, with the emergence of Compute Unified Device Architecture (CUDA), a programming framework based on C language, the GPUs are widely used for general purpose computation. CUDA has reduced the programmer's efforts in parallelizing the applications on CPU-GPU heterogeneous systems.In CUDA, the CPU along with its memory is referred to as the host, and the GPU along with its memory is referred to as the device. The code that runs on the device is known as the kernel.

Revised Manuscript Received on 30 May 2019.
***** Correspondence Author
Raju K*, Department of Computer Science and Engineering, NMAM Institute of Technology, Nitte, Karkala, Karnataka, India.
Niranjan N Chiplunkar, Department of Computer Science and Engineering, NMAM Institute of Technology, Nitte, Karkala, Karnataka, India.

© The Authors. Published by Blue Eyes Intelligence Engineering and Sciences Publication (BEIESP). This is an open access article under the CC-BY-NC-ND license http://creativecommons.org/licenses/by-nc-nd/4.0/

Threads in CUDA are organized into thread blocks and grids. A thread block is a group of threads, and a grid is group of thread blocks. An instance of the kernel code is executed by each thread of the grid.

Each thread of a block has a thread id and each thread block within a grid has unique block id. Threads belonging to the same block share the data among themselves using shared memory associated with that block. Thread synchronization within a block is achieved through barrier synchronization mechanism. A global memory is available for all threads for reading and writing purposes.

The control flow during the execution of any CUDA program is as shown in Fig. 1. A CUDA program is the combination of host code and device code executed by CPU and GPU respectively. The address spaces of host memory and the device memory are different. Before the device code or the kernel begins its execution, the required input data resides in the host memory. Since the host memory is not directly accessible to the device, the input data is transferred from host to device memory through PCI-Express. After the host to device data transfer the kernel function is invoked. During the kernel execution each of thread in the grid execute an instance of the kernel code, but processing different portions of input data. On completion of the kernel execution the computed results are transferred from device to host memory.

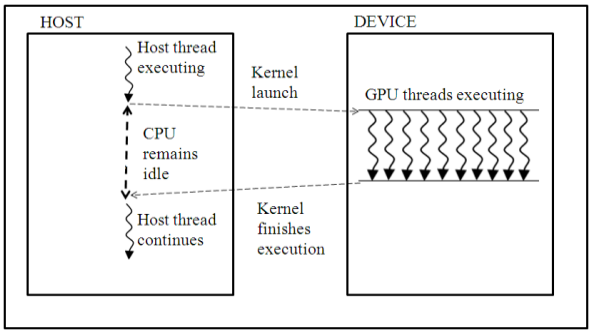

Fig. 1: Execution control flow of a CUDA program.

In the execution flow described above, the process of invoking a kernel is an asynchronous or non-blocking operation. That is, instead of waiting for the device to complete the kernel execution, the control is returned to the host immediately after the kernel is launched. Even though the host reacquires the control, it can only perform host to device data transfers and launch new kernels by using CUDA streams. The multiple cores of the CPU are neither used for kernel execution nor for executing any other independent computations of the current application.

Hence the enormous computational power of the CPU cores is wasted. The computational power of idle CPU cores can be effectively utilized by concurrently executing different independent computational tasks of an application on both CPU and GPU cores. That is, in an application with multiple independent tasks, when the GPU is running a kernel, other independent tasks can be assigned to the CPU cores. This method of execution improves the utilization of the computational resources in the CPU-GPU heterogeneous system, which in turn can improve the performance.

In this paper, we present our approach for the concurrent execution of independent kernels of a CUDA application on CPU and GPU. For a set of test applications we evaluate the effectiveness of our approach by orchestrating the CPU-GPU concurrent execution. Through experiments we analyze the suitability of a processing device (CPU or GPU) for the execution a given computational task and thereby determine an optimal assignment of tasks to processors. Finally, we compare the performance of CPU-GPU concurrent execution to the GPU-only execution of the application.

The rest of this paper is organized as follows. Section II provides the implementation details of concurrent execution of two independent kernels of a CUDA application. Section III presents the details of concurrent execution of multiple CUDA kernels. In Section IV, we analyze the results of our experiments. The related work is discussed in Section V, and we conclude our work in Section VI.

II. CONCURRENT EXECUTION OF TWO INDEPENDENT KERNELS

In this section we present the method to concurrently execute two independent kernels, one each on CPU and GPU. Fig. 2 depicts the GPU-only execution flow of two independent kernels of a CUDA application.

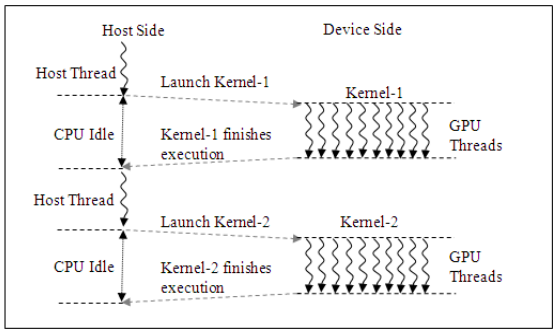

Fig. 2: GPU-only execution of two CUDA kernels.

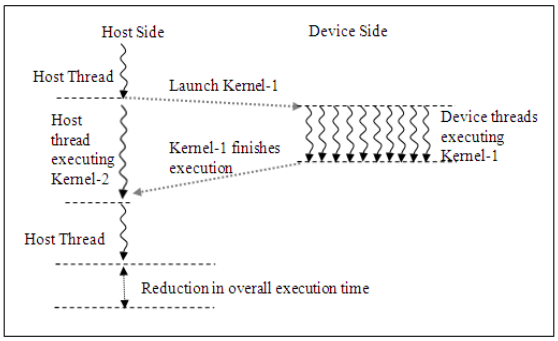

Fig. 3: CPU-GPU concurrent execution of two CUDA kernels.

It can be observed from Fig. 2 that the CPU cores remain idle when a kernel is executing on the GPU. As the kernel launch is non-blocking operation, the control returns to the host thread immediately after a kernel is invoked. The host thread could execute kernel-2 on the CPU while the kernel-1 is executing on the GPU as shown in the Fig. 3. However, the concurrent execution of the two kernels is possible only when the kernel-2 does not dependent on kernel-1, i.e. kernel-2 does not require the results produced by kernel-1for its execution. For an application having two independent CUDA kernels, say kernel-1 and kernl-2, we perform two separate runs to determine which kernel is to be executed on a particular processor (CPU or GPU) so that the overall concurrent execution time is minimal. Accordingly, the combination for first run is kernel-1 on CPU and kernel-2 on GPU, and the vice versa for the second run. For the first run, we implement the concurrent execution of the two kernels using the following steps:

1) Declare the host and device copies of the input and output data. Initialize the host copy of the input data values.
2) Send the input data from the host memory to the device memory that is required for the execution of the kernel-1on the GPU.
3) Launch the kernel-1 on GPU.
4) Soon after the kernel launch, the host thread spawns a child thread which in turn invokes the CPU function equivalent to the kernel-2. Among the various APIs and frameworks available for threading on the host side, we opted to use Pthreads (POSIX threads) due to the low overhead associated with thread creation and management.
5) After the execution of the kernel-1 on the GPU, the output data is transferred back to the CPU memory.
6) The CPU-GPU execution time for the two kernels is measured including the input and output data transfer time between CPU and GPU.

Similarly, a second run is performed by interchanging the assignment of kernels to processors. Among these two runs the one which yields low execution time is considered for computing execution speedup.

The pseudocode that translates the above six steps is as shown below. In this pseudocode we have taken 1D stencil operation as kernel-1 and binary search operation as kernel-2.

```
Step1:
 // For 1d stencil, allocate space for host and
 // device copies of input and output data (in, out,
 // d_din, d_out)  of size bytes, and setup values.
 int *in, *out;
 int *d_in, *d_out;
 in = (int *)malloc(size);
 out = (int *)malloc(size);
 cudaMalloc((void **)&d_in, size);
 cudaMalloc((void **)&d_out, size);
 // for binary search declare the input parameters
 // in a structure inpar and initialize values
 Step2:
 cudaEventRecord(start, 0);
 //Copy input data from CPU to GPU.
 cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(d_out,out,size,cudaMemcpyHostToDevice);

Step3:
 // Launch stencil kernel on GPU.
 stencil_1d<<<No_of_Blocks,Block_Size>>>();

 Step4:
 // Launch binary search thread on CPU.
 ret=
pthread_create(&thread,NULL,bin_srch,(void*)&inpar);

 Step5:
 // Copy results from GPU to CPU.
cudaMemcpy(out,d_out,size,cudaMemcpyDeviceToHost);

 //Wait for the CPU thread to complete.
 pthread_join(a_thread,NULL);

 Step6:
 //Measure execution time.
 cudaEventRecord(end, 0);
 cudaEventElapsedTime(&time, start, end);
```

Table 1: Test applications and the kernel pairs used in each application for pair-wise concurrent execution

Kernel Label	Kernel Name	CPU Time (ms)	GPU Time (ms)
K1	Vector Addition	1.215	2.259
K2	Vector Dot Product	3.231	1.571
K3	1-D Stencil Operation	3.21	2.205
K4	Parallel Reduction	0.951	1.581
K5	Binary Search	0.526	0.762
K6	Matrix Addition	1.445	2.265
K7	Matrix Transpose	1.544	1.461
K8	Matrix Multiplication	64.183	5.384
K9	Sparse Matrix Multiplication	113.016	9.21

To evaluate the performance of concurrent execution through this approach, we have conducted a series of experiments on an Ubuntu (12.04LTS) system with Intel Core i7-2600 quad-core processor at 3.4GHz, and an NVIDIA GeForce GTX 470 device with a clock speed of 1.215 GHz. The GPU has 14 streaming multiprocessors(SM), 32 CUDA cores per SM (total 448 CUDA cores). We use CUDA Development Toolkit 8.0 [17], NVIDIA Device Driver (version 295.41), and GCC 4.6.

Table 2: List of kernels used in the experiments.

Test application	Kernel Pair used in the test application
A1	Matrix multiplication and Dot product (K8,K2)
A2	Binary search and Stencil (K5, K3)
A3	Parallel reduction and Dot product (K4,K2)
A4	Vector addition and Dot product (K1,K2)
A5	Vector addition and Matrix multiplication (K1,K8)
A6	Matrix Transpose and Matrix multiplication (K7, K8)
A7	Sparse Matrix Multiplication and Matrix Addition(K9, K6)

We have executed seven different CUDA C test applications labeled A1 through A7, listed in Table 1. In each test application, we have devised the concurrent execution of a pair of kernels as explained above. The kernels used in our applications along with the GPU execution time and CPU execution time of each individual kernel is listed in Table 2. All input and output matrices as well as vectors used in different kernels are of uniform size and are of type *float*. The size of vectors is 65536 elements and the dimension of matrices is 256x256. Table 3 lists the concurrent execution times (in milliseconds) for kernel pairs of different test applications. Table 4 presents speedup obtained from CPU-GPU pair-wise concurrent execution compared to the GPU-only execution. In this table, the CPU-GPU concurrent execution time for a test application is the smaller of the execution times corresponding to the two runs (shown in Table 3) of that application.

Table 3: Time taken for the concurrent execution of kernel pairs.

Appli-cation	Run sequence	Kernel on the CPU	Kernel on the GPU	Execution time (ms)
A1	R1	K8	K2	64.725
	R2	K2	K8	**5.388**
A2	R1	K5	K3	**2.207**
	R2	K3	K5	3.937
A3	R1	K4	K2	**1.573**
	R2	K2	K4	3.503
A4	R1	K1	K2	**1.573**
	R2	K2	K1	4.404
A5	R1	K1	K8	**5.334**
	R2	K8	K1	65.458
A6	R1	K7	K8	**5.33**
	R2	K8	K7	64.348
A7	R1	K9	K6	113.542
	R2	K6	K9	**9.255**

Table 4: Execution speedup of pair-wise concurrent execution.

Appli-cation	Execution Time (ms)		Speedup
	GPU only	CPU+GPU (optimal)	
A1	6.853	5.388	1.272
A2	2.907	2.207	1.317
A3	3.125	1.573	1.987
A4	3.744	1.573	2.380
A5	7.567	5.334	1.419
A6	6.829	5.33	1.281
A7	11.42	9.255	1.234

III. CONCURRENT EXECUTION OF MULTIPLE INDEPENDENT KERNELS

In the method presented in Section 2, only a single CPU core is used to execute a routine that is functionally equivalent to a kernel. Applications having multiple kernels can be further benefited by utilizing all of the available CPU cores.

To evaluate the performance gain from concurrently executing multiple kernels on CPU and GPU, we extended our experiments by including five tasks in each test application, executing four of them on CPU and one on GPU.

We have executed eleven test applications, B1 through B11, each of which consists of different combination of kernels that are listed in Table 2. The execution time and the speedup for these test applications are listed in Table 5.

Table 5: Execution time and speedup for concurrent execution of multiple kernels.

Appli-cation	Kerne-ls on CPU	Kern-el(s) on GPU	Execution Time (ms)		Speed-up
			GPU only	CPU+ GPU	
B1	K3,K4, K6,K7	K1	9.563	2.386	4.008
B2	K1,K3, K4,K6	K2	9.642	3.181	3.031
B3	K1,K4, K6,K7	K3	9.563	2.218	4.312
B4	K1,K2, K3,K5	K4	8.191	3.642	2.249
B5	K1,K2, K3,K4	K5	8.191	3.431	2.387
B6	K3,K4, K6,K7	K8	12.734	5.41	2.354
B7	K5,K6, K7	K8, K9	18.898	14.582	1.296
B8	*K5,K7, K8, K9*	*K6*	*18.898*	*113.011*	*0.167*
B9	*K5,K6, K8,K9*	*K7*	*18.898*	*113.228*	*0.167*
B10	*K5,K6, K7,K9*	*K8*	*18.898*	*107.677*	*0.176*
B11	*K5,K6, K7,K8*	*K9*	*18.898*	*61.362*	*0.308*

The implementation method for each test application remains the same as the method explained in Section 2, except that the host spawns four child threads after launching a kernel on the GPU. Each CPU thread executes the functionally equivalent code of a kernel. The columns 2 and 3 of Table 5 show the kernel combination used in each application for execution on CPU and GPU respectively. The GPU-only execution time here means the time taken to execute the five different kernels of an application sequentially on the GPU. However, for GPU-only execution we have used individual CUDA streams to launch each kernel, so that device to host and host to device data transfers of two kernels are overlapped.

IV. OBSERVATIONS FROM THE EXPERIMENTS

The kernels listed in Table 2 are of compute-intensive and control-intensive types. Kernels in each application are chosen such that the kernels will be either of different types or of different degree of workloads. Kernels such as K8 and K9 are compute-intensive. The compute-intensive kernel is one in which the compute-to-memory access ratio is high. In this paper we refer to compute-intensive kernels as *long* kernels. We refer the kernels with less workload such as K1, K2, etc. as *short* kernels. Kernels such as K5 and K9 are control-intensive kernels, which involve more conditional branch instructions. From the results presented in Table 3, it can be observed that for any application the execution time is different in each of the two runs, R1 and R2. The optimal execution time is achieved by assigning a task to a suitable processor, CPU or GPU. Suitability of a processor for the execution of a computational task depends on several factors such as the type of the kernel, the amount of the computational load, etc. In our experiments, the GPU execution time for a kernel is measured including the input and output data transfer time. From Table 2, it can be seen that for kernels such as K8, and K9, the GPU execution time is much lesser than that of CPU. This is due to the fact that kernels K8 and K9 are compute-intensive and the availability of large number of cores on GPU enable higher throughput. From Table 2, it can also be observed that for kernels K1, K4, K5, K6, the GPU time is more than the CPU time. The workload in these kernels is so less that the computational power of hundreds of GPU cores is not utilized to their fullest capacity. Moreover, as the CPU operates at higher clock rate than that of GPU, execution time for a shorter kernel will be almost same on both CPU and GPU. When such a kernel is executed on GPU the data transfer time together with the execution time will become more the CPU execution time of that kernel. Thus, it can be observed from the run sequences A1-R2, A3-R1, A4-R1, A5-R1, and A6-R1 in Table 3 that for the concurrent execution of a pair of kernels the optimal execution time is always obtained when the shorter kernel is assigned to CPU and the longer one to the GPU. The control intensive kernels perform well on the CPUs compared to the GPUs. This is due to the availability of sophisticated branch prediction hardware support in the CPUs. Moreover, on GPUs, the control intensive kernels suffer performance degradation due to the thread divergence phenomenon. On a GPU a group of 32 threads, known as warp, is scheduled at a time. When a warp is selected for execution, all threads of the warp execute the same instruction, processing different data. But, a branching instruction forces the threads in the warp take different paths. In this case execution of the threads that evaluate to true is followed by the threads that evaluate to false. This phenomenon is known as thread divergence. Thus the, the simultaneous execution of all the threads in the warp is not possible and the overall execution time of the kernel increases. Hence it can be observed in Table 2 that the GPU execution time for kernel K9 is more than that of K8 even though the size of input data and the computations involved is same in both kernels. However, compared to CPU, the execution times for both K8 and K9 are lesser on GPU. Once again, in our experiment, the higher throughput of GPUs for executing long kernels with large size of input data subside the impact of thread divergence. However, a control intensive kernel having less computational workload performs better in CPU as in the case of kernel K5. Hence, among a pair of kernels, the longer kernel is suitable for GPU execution even when it involves conditional branch instructions (as in the run sequence A7-R2 in Table 3). Similarly, while executing two short kernels concurrently, the control intensive kernel is suitable for CPU execution and the one without conditional branch instructions is suitable for GPU execution (as in the run sequence A2-R1).

Fig. 4 shows the execution times for GPU-only execution and CPU-GPU concurrent execution of two kernels, in each of the applications A1 through A7. The graph in Fig. 5 plots the speedup achieved for each application, which ranges from 1.23x for A7 up to 2.38x for A4. The average speedup of the pair-wise concurrent execution is 1.55x.

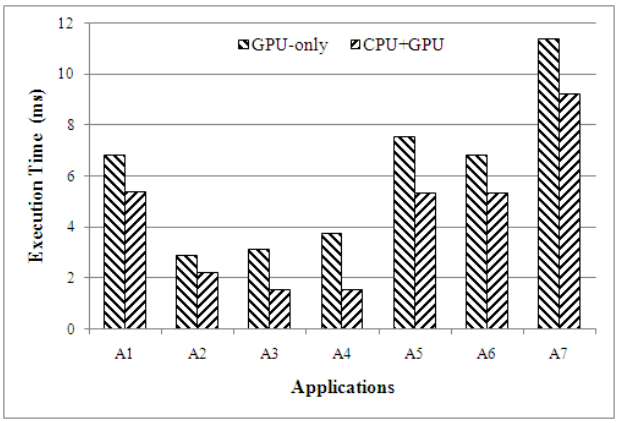

Fig. 4: Comparison of times for concurrent execution of two kernels in an application.

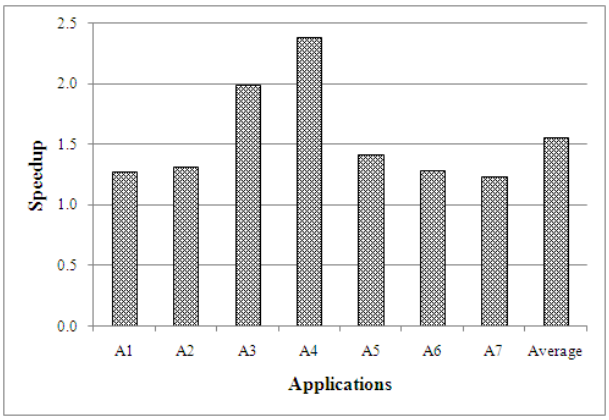

Fig. 5: Speedup obtained from concurrent execution of two kernels in an application.

Among eleven different test applications used for the concurrent execution of multiple kernels (discussed in Section 3), except in application B7 (in Table 5) we have executed one kernel on GPU and four kernels on CPU. Among these applications, for B1 through B6 we have observed speedup compared to the GPU-only execution of the kernels in the corresponding applications. From these results it is obvious that, for any application the overall execution time is equal to the time taken by the longest kernel of that application

For applications B8 through B11, the performance of concurrent execution is even poorer than that of GPU-only execution. As discussed earlier, the longer kernels K8 and K9 perform much better on GPU than CPU. Compared to GPU execution time, the CPU execution time of these two kernels is very high. In applications B8 through B11, one or both of the kernels K8 and K9 are executed on CPU. Hence for these applications, the CPU execution time of kernel(s) K8 and/or K9 determine the overall execution time of the individual applications. From these observations we can infer that in any application having one or more longer kernels, it is appropriate to execute the longer kernel(s) on GPU. Accordingly, it can be observed that the performance of B7 is comparatively better than that of applications B8 through B11. Though the speedup of the application B7 is marginal, the combination of kernel assignment used here appears to be optimal compared to the assignments used in applications B8 through B11.

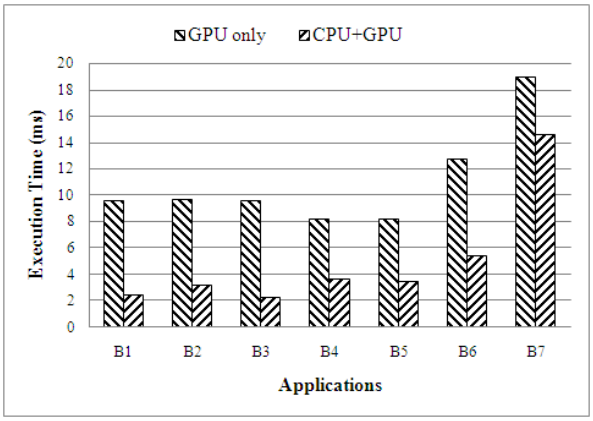

Fig. 6: Comparison of times for concurrent execution of multiple kernels in an application.

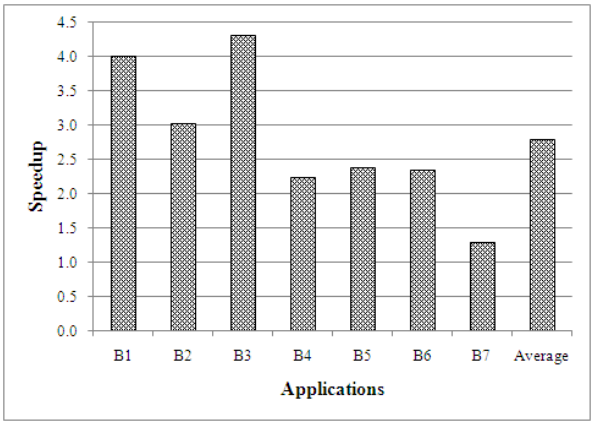

Fig. 7: Speedup obtained from concurrent execution of multiple kernels in an application.

The execution times for GPU-only execution and CPU-GPU concurrent execution of multiple kernels in each of the applications B1 through B7 is shown in Fig. 6. Since the applications B7 through B11 consist of same set of kernels, and the assignments of kernels to processors in applications B8 through B11 is not optimal, we have considered the performance of only application B7 for the comparison of execution times.

The graph in Fig. 7 plots the speedup achieved for applications B1 through B7. The speedup ranges from 1.29x for B7 up to 4.31x for B3. The average speedup of the pair-wise concurrent execution is 2.8x.

In general, the observations from the above experiments can be summarized as follows:

- Performance improvement from concurrent execution of kernels on CPU-GPU heterogeneous systems can be obtained by judiciously allocating kernels to processors.
- Assign longer kernels to GPUs as these kernels are benefited from the high throughput of the GPU.

- Even though shorter kernels may also be benefited from the high throughput of the GPU, the benefit is overridden by the CPU-GPU data transfer overhead. Since CPUs operate at higher clock rate than GPUs, the execution time for shorter kernels on a CPU will be almost same as that of a GPU. Hence shorter kernels are suitable for execution on CPUs.
- When two shorter kernels are to be executed, allocate a kernel with more conditional branch instructions to CPU and the one with more of computations to GPU.

V. RELATED WORK

There exist several mechanisms that allow the cooperative execution of CUDA or OpenCL kernels on CPU and GPU.

MCUDA [1] is a compiler-based framework to translate CUDA kernels into a code suitable for executing on CPU with multiple cores. It performs source to source translation of CUDA kernels, which requires an additional step of recompilation.

Ocelot [2] is complier framework that takes CUDA binary in the PTX (Parallel Thread Execution) form and generates target code for execution on multicore CPUs. Hence, Ocelot does not need a recompilation step as it is in MCUDA.

Execution of OpenCL kernels on CPUs is made possible by a framework called Twin Peaks [3]. It makes use of LLVM compiler framework [4] to generate device specific binaries.

The translators like MCUDA, Ocelot, and Twin Peaks make it possible to execute the GPU kernels on CPU only, but they do not support execution of kernels cooperatively on both GPU and CPU cores.

StarPU [5] is a runtime layer that provides API extension for OpenCL programs supporting the execution of a given kernel cooperatively on multicore processors and GPUs. The drawback of this system is that for heterogeneous execution, the existing OpenCL program has to be rewritten using StarPU APIs.

The FluidiCL[6] framework is capable of cooperatively executing a OpenCL kernel on both CPU and GPU. Without any prior training the workload is dynamically distributed among CPU and GPU based on the execution speed of the processors.

Single Kernel Multiple Device (SKMD) [7] divides a given OpenCL kernel into sub kernels for execution on CPU and GPU. A compiler dynamically transforms these sub kernels into CPU and GPU executable forms. The drawback of SKMD is that it requires profiling to distribute the load among different devices. Moreover, both FluidiCL and SKMD systems work only with OpenCL environment.

Execution of a single kernel on multicore CPU and GPU in JavaScript environment is enabled using JAWS (JavaScript framework for adaptive work sharing) [8]. JavaScript engines support a thread-like programming construct known as Web Workers [18], which is used in JAWS to run a kernel on CPU. Similarly, to run a kernel on GPU a parallel programming framework for data-parallel applications known as WebCL [19] is used.

There exist some application specific approaches for simultaneous execution of the application on CPU and GPU. The papers [9-14] present CPU-GPU hybrid implementation approaches for different applications.

Qilin [15] is training and profiling-based programming system that adaptively maps computations in CUDA programs to CPU and GPU cores. However, the existing CUDA program needs to be rewritten using Qilin APIs.

Cooperative Heterogeneous Computing (CHC) [16] is a framework that supports partitioning of a CUDA kernel for execution on host and the device. Workload division is done statically. The input to the framework is the CUDA kernel in PTX form. Ocelot is used to dynamically translate the input to Low Level Virtual Machine Intermediate Representation (LLVM IR) form. Finally, the code in the LLVM IR form is executed on the CPU using LLVM-JIT. But using this framework, at a time only one kernel can be executed concurrently on CPU and GPU. Whereas the approach discussed in this paper aims at executing multiple kernels concurrently.

VI. CONCLUSION

This paper presents an approach to efficiently utilize the computational power of multicore CPU in addition to the GPU device for the concurrent execution of multiple independent kernels of a CUDA application. This method improves the utilization of computational resources.

The concurrent execution results in an average speedup of 1.55x across a set of test applications, each having two independent kernels, and an average speedup of 2.8x across a set of test application, each having multiple independent kernels. Hence, we believe that the concurrent execution approach can be used to improve the performance of CUDA applications that involve multiple independent data parallel kernels.

In our approach, the allocation of kernels to processors is done heuristically which may not guarantee good performance. Hence, as a future work we plan to develop a scheme to determine the allocation of kernels to suitable processors.

REFERENCES

1. Stratton, J.A., Stone, S.S. and Wen-mei, W.H., MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In LCPC, Vol. 2008, (2008), pp. 16-30.
2. Diamos, G.F., Kerr, A.R., Yalamanchili, S. and Clark, N., 2010, September. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In Proceedings of the 19th international conference on Parallel architectures and compilation techniques (pp. 353-364). ACM.
3. Gummaraju, J., Morichetti, L., Houston, M., Sander, B., Gaster, B.R. and Zheng, B., Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In Proceedings of the 19th international conference on Parallel architectures and compilation techniques, (2010), pp. 205-216, ACM.
4. Lattner, C. and Adve, V., LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization , (2004), p. 75, IEEE Computer Society.
5. Augonnet, C., Thibault, S., Namyst, R. and Wacrenier, P.A., StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In Concurrency and Computation: Practice and Experience, 23(2), (2011), pp.187-198.
6. Pandit, P. and Govindarajan, R., Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. In Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, (2014), p. 273. ACM.

7. Lee, J., Samadi, M., Park, Y. and Mahlke, S., SKMD: Single Kernel on Multiple Devices for Transparent CPU-GPU Collaboration. In ACM Transactions on Computer Systems (TOCS), 33(3), (2015), p.9.
8. Piao, X., Kim, C., Oh, Y., Li, H., Kim, J., Kim, H. and Lee, J.W., JAWS: a JavaScript framework for adaptive CPU-GPU work sharing. In ACM SIGPLAN Notices,Vol. 50, No. 8, (2015), pp. 251-252). ACM.
9. Chen, R.B., Tsai, Y.M. and Wang, W., Adaptive block size for dense QR factorization in hybrid CPU-GPU systems via statistical modeling. In Parallel Computing, 40(5), (2014), pp.70-85.
10. Zhang, F., Hu, C., Wu, P.C., Zhang, H. and Wong, M.D., Accelerating aerial image simulation using improved CPU/GPU collaborative computing. In Computers & Electrical Engineering, 46, (2015), pp.176-189.
11. Wan, L., Li, K., Liu, J. and Li, K., Efficient CPU-GPU cooperative computing for solving the subset-sum problem. In Concurrency and Computation: Practice and Experience, 28(2), (2016), pp.492-516.
12. Yao, Y., Ge, B.H., Shen, X., Wang, Y.G. and Yu, R.C., STEM image simulation with hybrid CPU/GPU programming. In Ultramicroscopy, 166, (2016), pp.1-8.
13. Liu, J., Hegde, N. and Kulkarni, M., Hybrid CPU-GPU scheduling and execution of tree traversals. In Proceedings of the 2016 International Conference on Supercomputing, (2016), p. 2. ACM.
14. Antoniadis, N. and Sifaleras, A., A hybrid CPU-GPU parallelization scheme of variable neighborhood search for inventory optimization problems. In Electronic Notes in Discrete Mathematics, 58, (2017), pp.47-54.
15. Luk, C.K., Hong, S. and Kim, H., Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture , (2009), pp. 45-55, ACM.
16. Lee, C., Ro, W.W. and Gaudiot, J.L., Boosting CUDA Applications with CPU-GPU Hybrid Computing. In International Journal of Parallel Programming, 42(2), (2014), pp.384-404.
17. CUDA C Programming Guide, Version 8.0, Nvidia Corporation (2017). Available online: www.nvidia.com
18. Web Worker. Available online: http://www.w3.org/TR/workers
19. WebCL Standard. Available online: www.khronos.org/webcl/

AUTHORS PROFILE

Raju K. received the M.Tech. degree in Computer Engineering from Visvesvaraya Technological University, Belgaum, in 2007 and currently pursuing his Ph.D. degree in Computer Science and Engineering under VTU, Belgaum. He has 10 years of teaching experience at NMAMIT, Nitte. His research interests include performance improvement in GPU computation. Currently he is working as Associate professor at NMAMIT, Nitte.

Prof. Niranjan N. Chiplunkar received the B.E. degree in Electronics and Communication from NIE Mysore, in 1986 and M.Tech in Computer Science and Engineering from MIT, Manipal. He acquired his Ph.D. in Computer Science and Engineering from SJCE Mysore under Mysore University in 2002. He has more than 25 years of teaching experience. Currently he is working as the Principal of NMAMIT, Nitte. He has presented more than 50 technical papers in National and International conferences and journals. His research interests are in the areas of Parallel Computing and CAD for VLSI. He is a member of IEEE, CSI, and ISTE.