Applied Computer Science, vol. 73, no. 3, pp. 5–18
doi: 10.23743/acs-2021-17

Submitted: 2021-06-13 | Revised: 2021-09-07 | Accepted: 2021-09-21

CPU-GPU, High-performance computing,
Kernel, Data transfer, CUDA streams

K. RAJU [0000-0001-6731-5427]*, *Niranjan N CHIPLUNKAR* [0000-0003-4223-2355]**

PERFORMANCE ENHANCEMENT OF CUDA APPLICATIONS BY OVERLAPPING DATA TRANSFER AND KERNEL EXECUTION

Abstract

The CPU-GPU combination is a widely used heterogeneous computing system in which the CPU and GPU have different address spaces. Since the GPU cannot directly access the CPU memory, prior to invoking the GPU function the input data must be available on the GPU memory. On completion of GPU function, the results of computation are transferred to CPU memory. The CPU-GPU data transfer happens through PCI-Express bus. The PCI-E bandwidth is much lesser than that of GPU memory. The speed at which the data is transferred is limited by the PCI-E bandwidth. Hence, the PCI-E acts as a performance bottleneck. In this paper two approaches are discussed to minimize the overhead of data transfer, namely, performing the data transfer while the GPU function is being executed and reducing the amount of data to be transferred to GPU. The effectiveness of these approaches on the execution time of a set of CUDA applications is realized using CUDA streams. The results of our experiments show that the execution time of applications can be minimized with the proposed approaches.

1. INTRODUCTION

The Graphics Processing Unit (GPU) was originally developed for rendering the images. With hundreds of processing cores modern GPUs are found to be ideal for speeding up data parallel applications. CUDA C/C++ API is used for programming the GPU for general-purpose computation. The execution time of a program can be easily reduced by offloading the CPU computations to the GPU. As a result, nowadays the CPU-GPU heterogenous computing systems are extensively used in many high-performance computing applications.

The GPU is a co-processor operated under the control of central processing unit (CPU). According to CUDA terminology (NVIDIA, 2015), the CPU with its memory is called as host and the GPU with its memory is called as device. The function to be executed on the device is known as kernel. Thousands of threads are created when the kernel is invoked from the host. These threads are organized as blocks, and the blocks are organized as grid. An instance of the kernel code is executed by each thread of the grid.

* Department of CSE, NMAM Institute of Technology, Nitte, India, rajuk@nitte.edu.in, nchiplunkar@nitte.edu.in

Figure 1 depicts the control flow during the execution of any CUDA program. A CUDA program is the combination of host executable and device executable code. Since the host memory is not directly accessible to the device, input data is copied through PCI-e from the host to device. The kernel function is invoked after transferring the input data. The kernel is parallelly executed by several threads. However, each thread computes on separate portion of input data. On completion of kernel execution, the computed results are transferred from the memory of device to that of the host.

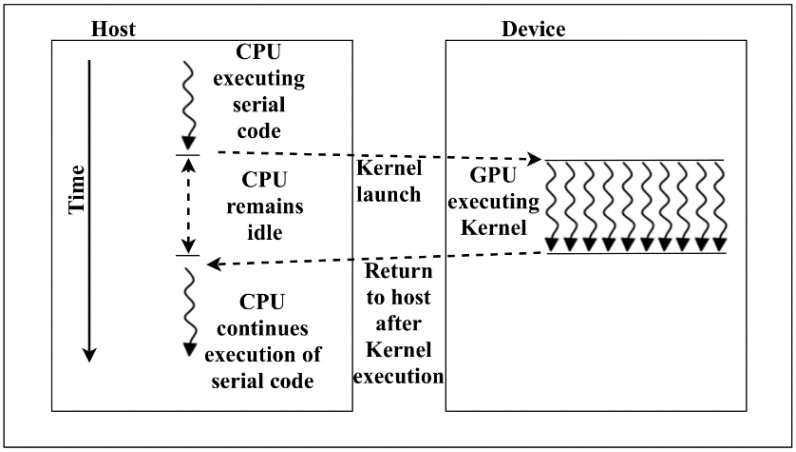

Fig. 1. Flow of control during the execution of CUDA program

The communication bandwidth of GPU memory is much higher than that of PCI-e. The data transfer speed is limited by the bandwidth of PCIe bus. The latest version of PCIe standard (PCI-e 5.0) supports a bandwidth of 128 GBps. NVIDIA's recent, most powerful supercomputing GPU architecture for PC, TITAN V, has a memory bandwidth of 652.8 GBps (NVIDIA TITAN V, n.d.), which is much higher than the bandwidth of PCIe 5.0. The data transfer overhead reduces the performance gain obtained from the use of GPU (Gregg & Hazelwood, 2011). This overhead can be reduced if the data transfer can happen in parallel with the kernel execution.

Another method to mitigate the data transfer bottleneck is to offload a part of GPU workload to the CPU cores. From the host's perspective, calling a kernel is an asynchronous process. A call to GPU function returns immediately and does not wait until the completion of kernel execution. While the GPU is executing the kernel, the CPU can continue the execution of host code, where it can perform computations, launch any other kernels, or transfer data between host and device memory. Modern CPUs comprise multiple cores, each with immense computing capacity. Huge amount of CPU computational power is wasted unless these cores are involved in some useful computation when the GPU is executing the kernel function. It is hard to find computation that can be performed on CPU cores in parallel with kernel execution, as these computations which are performed after the kernel launch normally depend on the results produced by the current kernel. However, in the case of data parallel applications a part of GPU workload can be offloaded to CPU cores. This approach reduces the GPU workload and also the volume of data traffic between CPU and GPU (Huang et al., 2012). Moreover, with this technique the idle CPU cores can be efficiently utilized.

CUDA provides streams, a mechanism that facilitates execution of the kernel in parallel with data transfer. Thus, CUDA applications can be positively sped up by offloading GPU workload to the CPU together with the overlapped kernel execution and data transfer.

In a CUDA program the operations that involve the device, such as CPU-GPU data transfer, kernel launch, etc., are issued by the host. A queue of GPU operations or commands is known as stream. The commands in a stream are executed in their arrival order. Every GPU command is associated with a stream, whose identifier is specified as a parameter to the command. If no stream identifier is specified by the programmer then that command is issued to the default stream or null stream.

A host thread can define multiple non-default streams. The commands in the different non-default streams are considered to be independent and can be executed concurrently. The commands coming from the same stream are executed in the sequential order. Figure 2-a depicts the execution timeline of the GPU operations with a default stream. The captions HtoD and DtoH followed by an index stand for the host to device and device to host data transfers respectively. A kernel is referred to by the caption K followed by an index. The index of a data transfer operation is same as the index of the corresponding kernel.

The commands are performed in the same order that they appear in the stream. The commands in the non-default streams will be started only after the completion of operations in the default stream. However, a new operation in the default stream cannot be started before the completion of already initiated operations in the non-default streams. Figure 2-b gives the execution timeline of the GPU operations on two non-default streams. As shown in this figure, the operations from two different streams can be overlapped. For example, execution of kernel K1 and the HtoD2 data transfer can happen simultaneously.

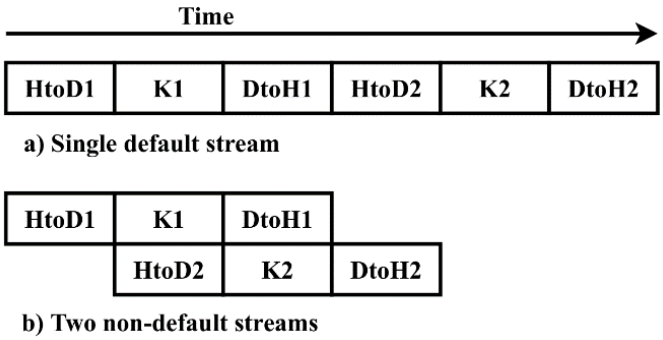

Fig. 2. Execution time line of CUDA commands with (a) single default stream, and (b) two non-default streams

Modern GPUs possess one execution engine and two copy engines. The host to device and device to host data transfer operations use separate copy engines. Data transfer commands from different streams are issued to the relevant copy engines. The copy engines can utilize the full duplex nature of PCI-e bus. That is, two data transfer operations of opposite directions which are from two different streams can be overlapped. Multiple cores of a GPU are considered as a single execution engine for kernel scheduling purpose. The following are the requisites to use CUDA streams to enable concurrent execution of operations through copy and the execution engines:

1. The device must support concurrent copy and kernel execution.
2. Two data transfer operations can be overlapped only if they are in different directions.
3. Commands (for copying the data and invoking the kernel) which are to be overlapped need to be added to separate non-default streams.
4. Pinned or non-pageable memory must be used at the host side.

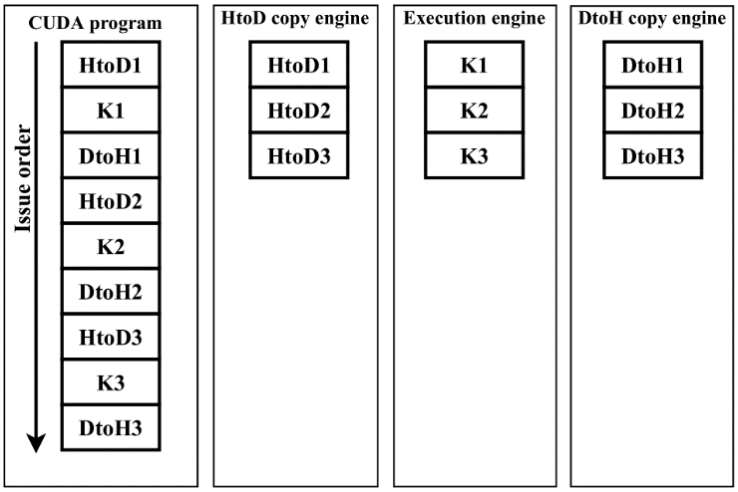

Fig. 3. Issue order of commands to CUDA engines

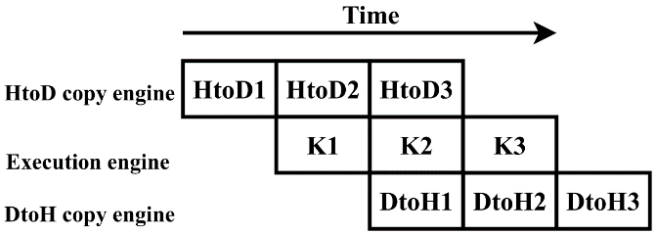

Fig. 4. Execution order of commands in copy and execution engines

Figure 3 shows the order in which the data transfer and kernel launch operations of a program are issued to appropriate CUDA engines. The execution order of commands from different engines are as shown in the figure 4. The operations from different engines can be overlapped if there exists no dependency among the them. Execution of kernel K1 cannot be overlapped with HtoD1 data transfer, as K1 and HtoD1 are from the same stream as shown in figure 2-b and K1 depends on the completion of HtoD1. Whereas HtoD2 data transfer and execution of kernel K1 can be overlapped since these two operations are from different streams and K1 does not depend on the HtoD2 data transfer. Data transfer operations HtoD3 and DtoH1 can be overlapped as these operations are from separate streams and are in opposite directions.

In CUDA, the data transfer operations using `cudaMemcpy()` are synchronous or blocking in nature. The function returns only after the data is copied. The non-blocking version this operation is `cudaMemcpyAsync()`, which requires the data to be allocated in the page-locked memory. The memory allocated using `malloc()` is pageable memory.

Host can swap this memory to disk when more space is required. Page-locked memory or pinned is a portion of memory that is not available for swapping. The pageable memory of the host does not support DMA transfers to or from GPU. Data has to be temporarily stored in page-locked memory before it is transferred to device. By explicitly allocating the data in page locked memory it can be safely used for DMA. Allocation of data in the pinned memory can be done with `cudaHostAlloc()`. It takes the same arguments as `cudaMemcpy()` except that it has an additional argument specifying the stream index into which this command is added. A stream can be created using `cudaStreamCreate()` and destroyed using `cudaStreamDestroy()` functions.

In this paper we aim to speed up the execution of CUDA programs by overlapping kernel execution with the data transfer between CPU and GPU. Using streams, we have implemented different kinds of concurrency for the execution of a set of CUDA kernels. We compared the overlapped execution time with the time taken by the non-overlapped or serial execution. Experimental results show that execution time of CUDA applications can be decreased by simultaneously performing the transferring of data and the execution of kernel.

2. RELATED WORK

Several methods have been proposed by the researchers to speed up GPU applications. These researches mainly focus on utilizing the computational power of CPU cores in addition to the GPU for kernel execution (Raju & Chiplunkar, 2018). Among such works, application specific methods to improve the performance are presented in papers (Antoniadis & Sifaleras, 2017; Fang, Chen & Mao, 2018; Siklosi, Reguly & Mudalige, 2019; Yang, Li & Li, 2017). These parallelization approaches are based on the characteristics of individual applications and cannot be generalized.

There exist some compiler frameworks and libraries that enable the programmer to offload a portion of GPU workload to CPU cores. Frameworks like FluidiCL (Pandit & Govindarajan, 2014), JAWS (Piao et al., 2015), and SKMD (Single Kernel Multiple Device) (Lee et al., 2015) are used for the execution of OpenCL kernels using both CPU and GPU cores. OpenCL allows the programmer to select a processing device, CPU or GPU, for the execution of a given kernel. The compiler automatically generates the binary code for the selected device. Since CUDA does not provide the above feature, implementation of a co-operative execution scheme on CUDA is more complex process compared to the implementation of same on the OpenCL. Cooperative Heterogeneous Computing (CHC) (Lee et al., 2014) is a prominent cooperative execution framework for CUDA which partitions an input kernel and executes the partitions concurrently on host and device.

Only a few approaches have been reported that focus on hiding the data transfer overhead by overlapping the communication and computation. A method to minimize CPU-GPU communication overhead is suggested by (Fu, Wang & Zhai, 2017), in which two or more data transfer operations of same direction are merged into a single operation. Using compiler techniques, multiple commands for data copying are moved to same location in the source code so that these operations can be merged. Merging of data transfer operations decreases the total number of data transfer operations. Hyper-Q feature supported by NVIDIA GPUs enables concurrent execution of multiple independent kernels on a single GPU. However, when the execution of multiple kernels are not properly ordered, contention for shared

resources can degrade the overall performance (Luley & Qiu, 2016). A model has been developed by (Lázaro-Muñoz et al., 2017) that determines the order of kernel execution so as to increase the possibilities of simultaneously performing the data transfer and kernel execution, and reduce the total execution time. An analytical performance model is proposed in (Werkhoven et al., 2014) for classifying the relative performance of the different techniques for overlapping computation and communication. An approach that partitions the input data into sub-blocks and overlaps the data transfer and execution of sub-blocks in a pipelined manner is proposed in (Li et al., 2017).

Gowanlock & Karsin (2019) have developed a sorting approach in which the input data is divided into multiple batches of uniform size which are sorted on the GPU. Sorted batches are merged on the host which completes the task of sorting. The process of transferring input data and sorted batches are overlapped using CUDA streams.

A method to minimize the of GPU memory consumption for training the Convolutional Neural Networks is proposed by (Hascoet et al., 2019). In their method, the GPU memory buffers are temporarily offloaded to CPU in the forward pass and transferred back to GPU as needed by the computation during the backward pass of the backpropagation algorithm. To reduce the PCI-e bottleneck the data transfers and GPU computations are overlapped using streams.

Dhake & Walunj (2019) have used GPU to check whether input data packets contain virus signature. A string-matching algorithm parallelly checks for the occurrence of different patterns of virus string in the input packet. (Patil & Kulkarni, 2021) have investigated the performance characteristics of CPU-GPU data transfer method that is based on pinned memory.

The research works presented in (Gowanlock & Karsin, 2019; Hascoet et al., 2019; Dhake & Walunj, 2019) and (Patil & Kulkarni, 2021) make use of pinned memory and streams to overcome the PCI-e transfer overhead. In our approach, in addition to overlapping the data transfer and kernel execution we have also investigated the advantage of offloading a portion of GPU workload to CPU through an approach called as 4– way and 4+ way concurrency. With this approach, the data transfer from CPU to GPU, kernel execution, data transfer from GPU to CPU, and execution on the CPU cores can occur simultaneously.

In GPU-based graph processing systems the major challenge is that the size of the input graph is so large that it cannot be fitted into the GPU memory. To manage the problem of GPU memory oversubscription (Sabet, Zhao & Gupta, 2020) have proposed a graph processing system in which a subgraph consisting of active vertices is loaded into GPU memory. This method significantly reduces the amount of data transfer between CPU and GPU and hence minimizes the overhead of data transfer. However, this approach is specific to graph processing systems and cannot be generalized.

NVLink is an emerging CPU-GPU communication link which is faster than PCI-e. (Lutz et al., 2020) have investigated the performance of NVLink with respect to processing large data sets and observed significant speedups compared to the usage of PCI-e. However, NVLink is yet to appear in the commodity hardware.

In our approach we use CUDA streams to overlap computation and communication. The input data set is divided into chunks. The execution of a kernel associated with a chunk and transferring of next chunk are overlapped using streams. Compared to the earlier research works our approach not only overlaps kernel execution and data transfer but also offloads a portion of GPU workload to CPU cores. Thereby it reduces the GPU workload as well as the volume of data to be transferred to GPU.

3. METHODOLOGY

Any CUDA program consists of three device related operations: namely, copy the input data from CPU to GPU memory, execute the kernel, and copy the results to CPU memory from the GPU memory. Even if these three steps are added to different streams, they cannot be overlapped due to the dependency between the operations. To enable the concurrency within a data parallel kernel, we divide the data set into several smaller chunks or tiles. The data transfer and kernel execution corresponding to two adjacent tiles can be overlapped by adding them into different non-default streams. CUDA streams can support different levels of concurrency based on the order in which the commands are organized in different streams. These concurrency levels are referred to as 2-way, 3-way, 4-way, and 4+ way concurrency.

The effectiveness of different levels of concurrency are tested with the following data parallel CUDA kernels:

- Vector addition,
- Vector dot product,
- 1-D stencil operation,
- Matrix transpose,
- Matrix multiplication.

In the above kernels the vectors and matrices are of integer data type. Different levels of concurrency are applied for the execution of individual kernels and also for the combined execution of all kernels. These experiments are carried out on a system having Intel Quad-Core i5-7300HQ, 2.50 GHz, 8GB host memory, NVIDIA GTX 1050 with 640 CUDA cores, 4GB device memory, CUDA compute capability of 6.0 and CUDA SDK 9.1. The operating system is Ubuntu (12.04LTS) and the GPU driver version is 24.21.13.9891. While compiling the above kernels we have not used any optimization flags supported by NVIDIA CUDA C compiler (nvcc).

In 1-D stencil and matrix multiplication kernels, each input data element is repeatedly used in multiple computations. On the GPU, the data elements are accessed from the off-chip global memory. GPUs also possess shared memory located within the chip. Access latency shared memory is lesser than that of global memory. On GTX 1050, the global memory bandwidth is 112 Giga Bytes/second, whereas the shared memory bandwidth is above 224 Giga Bytes/second. Shared memory is used in the implementation of above two kernels. The input data elements which are repeatedly used in the computation are copied from the global to shared memory. Frequent accesses to shared memory greatly reduce the access latency, thereby reducing the execution time. Since the shared memory feature is not supported on the CPU, the implementation of above kernels on the CPU are benefited by the same.

3.1. Two-way concurrency

To enable the two-way concurrency, we have divided the input data into four tiles of equal size. These tiles are loaded from the host to the device in succession. The PCI-e transfer operations are referred to as HtoD1 through HtoD4. After the data transfers are completed, the kernel function is invoked successively four times, each time with a different tile of input data as the input parameter. The kernels are referred to as K1 through K4. After each kernel

invocation, the data transfer operation is invoked to transfer the results produced by that kernel from the device to host. These data transfer operations are referred to as DtoH1 through DtoH4. It is necessary to add a kernel invocation operation and associated GPU to CPU data transfer operation into the same non-default stream. Four different pairs of kernel launch operation and the corresponding data transfer operations are added into different streams. Hence, the GPU to CPU data copying operation for a kernel can be overlapped with the execution of another kernel as shown in Figure 5. In this approach two operations from different streams are performed in parallel. Hence, this approach is referred to as 2-way concurrency.

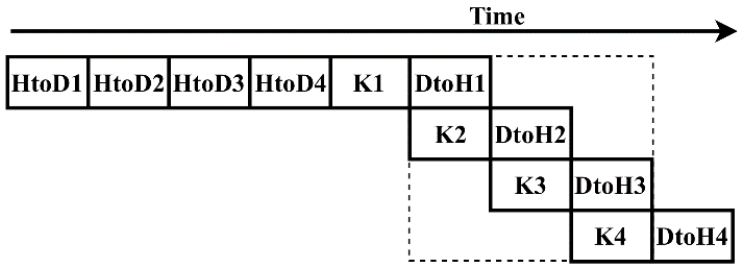

Fig. 5. 2-way concurrency

3.2. Three-way concurrency

In the previous method, kernels are launched only after all the four tiles of input data are copied from the CPU to GPU. In 3-way concurrency, the host to device data transfer for a given input data tile is followed by the corresponding kernel launch, which is then followed by the GPU to CPU data transfer of the results produced by the kernel. These three operations pertaining to a tile of input data are added to same non-default stream. The set of three operations, each pertaining to different tiles of input data are added into different non-default streams.

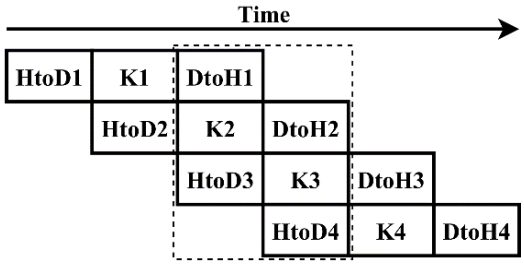

Fig. 6. 3-way concurrency

The three different operations (copying of the data from CPU to GPU, execution of the kernel, and copying of the results from GPU to CPU) corresponding to three different tiles of input data are overlapped as shown in the figure 6. Overlapping of the three operations is possible as these are issued to different CUDA streams and there exists no dependence between them.

3.3. Four-way concurrency

In the previous two concurrency approaches, the CPU cores remain idle when the kernel is being executed by the GPU. The overall execution time of the kernel can be improved if we could offload some portion of the GPU workload to CPU cores. In the 4-way concurrency approach, a portion of the input data is processed in host when the GPU is executing the kernel processing rest of the data. To process the tile allocated to the host, a routine that is functionally equivalent to the kernel is executed on the CPU. For any given application, the implementation of the CPU and GPU versions of the kernels are based on the same algorithm. Compared to OpenMP threads the thread creation and management overhead is lesser in the case of Pthreads. Hence, we have used Pthread APIs to create the CPU thread. This approach overlaps three different GPU operations and execution on the CPU as shown in figure 7. Thus, four different operations are performed simultaneously and hence the name 4-way concurrency.

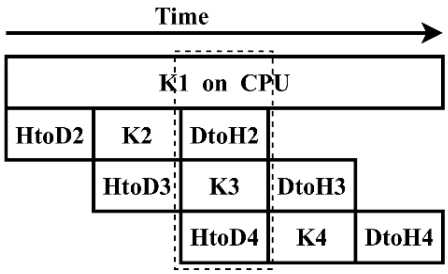

Fig. 7. 4-way concurrency

3.4. Four+ way concurrency

This type of concurrency is same as the four-way concurrency except that it uses multiple CPU threads. Each thread on the CPU can be assigned with separate tile of input data or multiple threads can be used to simultaneously process a single tile. In this way all CPU cores can be utilized for the computation. As shown in the figure 8, in the 4+way concurrency more than 4 operations can be performed in parallel.

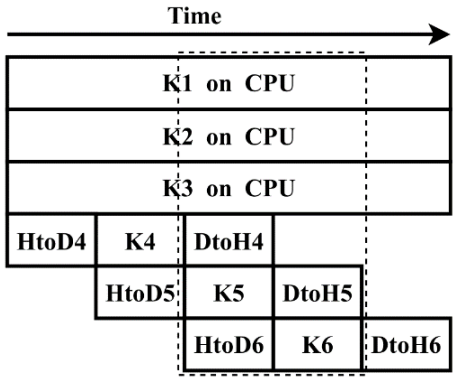

Fig. 8. 4+ way concurrency

The above different methods of concurrency can effectively decrease the time needed for the execution of a CUDA program. Moreover, even the CPU cores can also be engaged in the execution of a kernel. Hence, the overall performance of an application can be improved.

4. RESULTS AND ANALYSIS

The kernels used in our experiments are listed in section 3. The concurrency levels discussed above are applied to the execution of each individual kernel as well as the combined execution of different kernels within a single program. The results obtained from these experiments are discussed in section 4.1 and 4.2.

4.1. Applying concurrency for the execution of individual kernels

In this experiment, we have used 4 streams e for 2-way and 3-way concurrency, each one comprising of three GPU operations, namely CPU to GPU data transfer, kernel invocation, and GPU to CPU data transfer. Thus, we have executed 4 kernels, each processing the input data of uniform size. For vector-based kernels (Vector Addition, Vector Dot Product, and 1-D stencil operation), each of the kernels on different streams process a tile of 327680 integer elements. In the case of 4-way concurrency, we have used 3 streams to process three tiles of data on the GPU. The fourth tile is processed by a CPU thread. In the case of 4+ way concurrency, the fourth tile is processed in parallel by 4 different CPU threads. We have followed the same approach for matrix-based kernels (i.e. matrix transpose and matrix multiplication) also. The size of the matrix processed by each kernel and also by the CPU threads is 896×896 integer elements. Thus, for vector-based kernels the number of elements in each input vector is 1310720 elements (i.e., 5.1 MB) and for matrix-based kernels the size of each input matrix is 3211264 elements (i.e., 12.5 MB).

Tab. 1. Serial, 2, 3, and 4-way concurrent execution times (in milli seconds) for individual kernels

Kernel Name	Serial	2-way	3-way	4-way
Vector Addition	6.04	3.78	2.06	1.76
Vector Dot Product	3.31	3.21	1.78	1.42
Matrix Transpose	9.67	5.19	2.97	2.12
1-D Stencil Operation	4.57	2.28	1.28	6.77
Matrix Multiplication	73.14	68.66	62.33	4486.88

Table 1 lists the serial and concurrent execution time for the above kernels with 2, 3, and 4-way concurrency. While executing any application we have ensured that no other user applications are executing as background processes. However, we observe a slight difference in the two successive execution times of any CUDA program. This difference in execution times is attributed to the background processes of operating system. To protect the results of our experiments from this factor, each application is executed several times until no further improvement in the execution time is observed. Thus, for any application the execution time presented in this table and all the subsequent tables represent the lowest of multiple runs of the given application.

The serial execution uses the default stream as depicted in Figure 2(a). For vector addition, vector dot product, and matrix transpose, the execution time with 4-way concurrency is optimal and the speedup compared to the serial execution is 3.43, 2.33, and 4.56 respectively. These kernels involve less computation and require less time to execute. The 4+way execution time for different kernels is shown in the Table 2. The second and third columns of this table show the execution time when 2 threads and 4 threads respectively are used to process the tile allocated to the CPU. When a tile of input data is processed using multiple CPU threads, the overhead of creating multiple threads subsides the benefit of time saved by the parallelization of processing that tile on CPU. Hence, we do not observe performance gain with 4+ way concurrency either using two or four CPU threads.

Tab. 2. 4+way concurrent execution times (in milli seconds) for individual kernels

Kernel Name	2 CPU Threads	4 CPU Threads
Vector Addition	1.83	1.84
Vector Dot Product	1.47	1.54
Matrix Transpose	2.41	2.65
1-D Stencil Operation	3.65	2.15
Matrix Multiplication	2535.96	1387.31

Tab. 3. GPU-only and CPU-only execution time of individual kernels

Kernel Name	GPU Execution Time (milli seconds)	CPU Execution Time (milli seconds)
Vector Addition	1.33	1.02
Vector Dot Product	0.69	1.07
1-D Stencil	1.03	6.87
Matrix Transpose	2.27	3.73
Matrix Multiplication	20.07	95152.0

The CPU-only execution time for matrix multiplication and stencil operation is much higher than that of GPU-only time as shown in Table 3. This is due to the fact that these two kernels are computation intensive compared to the rest of the kernels. As GPUs consist of large number of cores, compared to the multicore CPUs they exhibit optimal performance for compute intensive data parallel kernels. Hence, as shown in the Table 3, for the above two kernels and even for any other kernels the GPU-only execution time is much lesser than the CPU-only time. The vast difference between the CPU-only and GPU-only time of matrix multiplication is also due to the fact that the algorithm used for multiplication on the CPU is not cache friendly. The simple matrix multiplication operation to multiply matrices A and B to produce matrix C is defined as $c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj}$, where A, B, and C are m×n, n×p, and m×p matrices respectively. The above method of multiplying two matrices can be implemented using three level nested loops. If we consider all matrices to be of order n×n, the size of the cache is B bytes, and the size of a cache line is b bytes, then the cache consists of $\frac{B}{b}$ cache lines. C-language stores matrices in row major order. When the inner loop accesses an element of matrix B, a cache line containing that element must be available in

the cache. According to the algorithm, the inner loop accesses only one element from that cache line. When $n > \frac{B}{b}$, every access to matrix B causes a cache miss. The algorithm results in $\theta(n^3)$ cache misses in the worst case.

GPU provides shared memory which has higher bandwidth than the global memory. We use tile-based algorithm for the matrix multiplication algorithm on the GPU. In this algorithm, a row of matrix A and a column of matrix B is accessed by different threads of a thread-block. By placing the tile containing these rows and columns in the shared memory, repeated long latency global memory accesses can be avoided. Moreover, when the threads of a warp access adjacent elements in the shared memory, GPU can coalesce these different accesses into a single access. Due to these features, the performance of matrix multiplication algorithm on GPU is significantly better than the performance of the same on the CPU. Hence, the 4-way and 4+way concurrencies which offload matrix multiplication workload to CPU increase the execution time compared to the 2-way and 3-way concurrencies which use only GPU.

Therefore, as shown in Table 1, the 3-way concurrency proves to be ideal for 1-D stencil operation and matrix multiplication with a speedup of 3.57 and 1.17 *respectively* compared to the serial execution. However, with 4+way concurrency it can be observed that for both kernels the execution time decreases as the number of CPU threads are increased based on the number of cores available. The decrease in the execution time in this case is attributed to distribution of huge amount of computational load among multiple threads.

4.2. Applying concurrency for the combined execution of different kernels

In this experiment we have overlapped the kernel execution and data transfer operations of five different kernels. Size of the vector for a vector-based application is 327680 elements and the size of the matrix for a matrix-based application is 896×896 elements. Table 4 shows the combined execution time with serial execution, 2-way, and 3-way concurrent execution of 5 different kernels. Compared to the serial execution (using default stream), the 2-way and 3-way concurrency results in a speedup of 1.15 and 1.28 respectively.

Tab. 4. Combined execution time of all kernels with serial execution, 2, and 3-way concurrency

Level of concurrency	Execution Time (in milli seconds)
1-way (single stream)	25.63
2-way	22.33
3-way	20.03

The performance of 4-way concurrency is tested by executing 4 different kernels on the GPU and remaining one on the CPU. In this way, the execution time for five different combinations of kernels are tested. Table 5 lists the execution time for the same. Except for matrix multiplication, speedup is observed when any of the kernel is executed on the CPU and rest are executed on the GPU. The speedup is calculated with respect to the serial execution time (i.e. 25.63ms). As discussed, GPU is suitable for matrix multiplication compared to CPU. Hence executing it on the CPU will not improve the performance.

The 4+ way concurrency is realized by executing vector addition and matrix transpose kernels on the host and the remaining 3 kernels on the device. These two kernels are chosen for execution on the host as their execution time on the host is minimum as shown in the Table 5. The two kernels on the host are executed in parallel by separate threads. Execution time for the above combination of 4+ way concurrency is 19.16 milliseconds which is slightly better than the execution time of matrix transpose kernel executed on the CPU using 4-way concurrency (i.e. 19.45 milliseconds).

Tab. 5. Combined execution time of all kernels with 4-way concurrency

Kernel on the CPU	Execution time (in milli seconds)	Speedup
Vector Addition	19.71	1.3
Vector Dot Product	20.43	1.25
1-D Stencil	20.05	1.28
Matrix Transpose	19.45	1.32
Matrix Multiplication	96169.88	0.0003

5. CONCLUSIONS

CUDA streams enable the overlapping of CPU-GPU communication and execution of kernel. The results of applying concurrency for the execution of individual kernels and combination of different kernels show a significant improvement in the execution time over serial or non-overlapped execution. However, while applying 4-way or 4+ way concurrency care must be taken so that the CPU thread creation overhead does not dominate the performance benefits of offloading the workload to CPU. In this regard, this work can be extended to device a mechanism that can dynamically select a suitable amount workload for the CPU based on its hardware features and the computational characteristics of the given application.

REFERENCES

Antoniadis, N., & Sifaleras, A. (2017). A hybrid CPU-GPU parallelization scheme of variable neighborhood search for inventory optimization problems. *Electronic Notes in Discrete Mathematics*, *58*, 47–54. https://doi.org/10.1016/j.endm.2017.03.007

Dhake, A.A., & Walunj, S.M. (2019). Transfer Time Optimization Between CPU and GPU for Virus Signature Scanning. In A. Luhach, D. Jat, K. Hawari, X.Z. Gao & P. Lingras (Eds.), *Advanced Informatics for Computing Research. ICAICR 2019. Communications in Computer and Information Science* (vol. 1076 pp. 70–78). Springer Singapore. https://doi.org/https://doi.org/10.1007/978-981-15-0111-1_6

Fang, J., Chen, H., & Mao, J. (2018). Understanding data partition for applications on CPU-GPU integrated processors. In *Communications in Computer and Information Science* (vol. 747). Springer Singapore. https://doi.org/10.1007/978-981-10-8890-2_32

Fu, C., Wang, Z., & Zhai, Y. (2017). A CPU-GPU Data Transfer Optimization Approach Based on Code Migration and Merging. *Proceedings - 2017 16th International Symposium on Distributed Computing and Applications to Business, Engineering and Science, DCABES 2017, 2018-Septe* (pp. 23–26). IEEE. https://doi.org/10.1109/DCABES.2017.13

Gowanlock, M., & Karsin, B. (2019). A hybrid CPU/GPU approach for optimizing sorting throughput. *Parallel Computing*, *85*, 45–55. https://doi.org/10.1016/j.parco.2019.01.004

Gregg, C., & Hazelwood, K. (2011). Where is the Data ? Why You Cannot Debate CPU vs. GPU Performance Without the Answer. *IEEE International Symposium on Performance Analysis of Systems and Software*. (pp. 134–144). IEEE. https://doi.org/10.1109/ISPASS.2011.5762730

Hascoet, T., Zhuang, W., Febvre, Q., Ariki, Y., & Takiguchi, T. (2019). Reducing the Memory Cost of Training Convolutional Neural Networks by CPU Offloading. *Journal of Software Engineering and Applications*, *12*(08), 307–320. https://doi.org/10.4236/jsea.2019.128019

Huang, W., Yu, L., Ye, M., Chen, T., & Hu, T. (2012). A CPU-GPGPU scheduler based on data transmission bandwidth of workload. *Parallel and Distributed Computing, Applications and Technologies, PDCAT Proceedings* (pp. 610–613). IEEE. https://doi.org/10.1109/PDCAT.2012.15

Lázaro-Muñoz, A.J., González-Linares, J.M., Gómez-Luna, J., & Guil, N. (2017). A tasks reordering model to reduce transfers overhead on GPUs. *Journal of Parallel and Distributed Computing*, *109*, 258–271. https://doi.org/10.1016/j.jpdc.2017.06.015

Lee, C., Woo, W.R., & Gaudiot, J. (2014). Boosting CUDA Applications with CPU – GPU Hybrid Computing. *International Journal of Parallel Programming*, *42*, 384–404. https://doi.org/10.1007/s10766-013-0252-y

Lee, J., Samadi, M., Park, Y., & Mahlke, S. (2015). SKMD: Single kernel on multiple devices for transparent CPU-GPU collaboration. *ACM Transactions on Computer Systems*, *33*(3). https://doi.org/10.1145/2798725

Li, T., Dong, Q., Wang, Y., Gong, X., & Yang, Y. (2017). Dual buffer rotation four-stage pipeline for CPU – GPU cooperative computing. *Soft Computing*, *23*, 859–869. https://doi.org/10.1007/s00500-017-2795-0

Luley, R.S., & Qiu, Q. (2016). Effective utilization of CUDA hyper-Q for improved power and performance efficiency. *Proceedings – 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016* (pp. 1160–1169). IEEE. https://doi.org/10.1109/IPDPSW.2016.154

Lutz, C., Breß, S., Zeuch, S., Rabl, T., & Markl, V. (2020). Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 1633–1649). ACM Digital Library. https://doi.org/10.1145/3318464.3389705

NVIDIA TITAN V. (n.d.). *NVIDIA Corporation*. Retrieved May 8, 2021 from https://www.nvidia.com

NVIDIA. (2015). *CUDA C Programming Guide v 9.1*. NVIDIA.

Pandit, P., & Govindarajan, R. (2014). Fluidic kernels: Cooperative execution of openCL programs on multiple heterogeneous devices. *Proceedings of the 12th ACM/IEEE International Symposium on Code Generation and Optimization, CGO 2014* (pp. 273–283). ACM Digital Library. https://doi.org/10.1145/2544137.2544163

Patil, S.V., & Kulkarni, D.B. (2021). Data transfer optimization in CPU/GPGPU Communication. *Turkish Journal of Computer and Mathematics Education*, *12*(13), 1920–1923.

Piao, X., Kim, C., Oh, Y., Li, H., Kim, J., Kim, H., & Lee, J.W. (2015). JAWS: A JavaScript framework for adaptive CPU-GPU work sharing. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP, 2015-Janua* (pp. 251–252). ACM Digital Library. https://doi.org/10.1145/2688500.2688525

Raju, K., & Chiplunkar, N.N. (2018). A survey on techniques for cooperative CPU-GPU computing. *Sustainable Computing: Informatics and Systems*, *19*, 72–85. https://doi.org/10.1016/j.suscom.2018.07.010

Sabet, A.H.N., Zhao, Z., & Gupta, R. (2020). Subway: Minimizing data transfer during out-of-GPU-memory graph processing. *Proceedings of the 15th European Conference on Computer Systems, EuroSys 2020* (pp. 1–16). ACM Digital Library. https://doi.org/10.1145/3342195.3387537

Siklosi, B., Reguly, I.Z., & Mudalige, G.R. (2019). Heterogeneous CPU-GPU execution of stencil applications. *Proceedings of P3HPC 2018: International Workshop on Performance, Portability and Productivity in HPC, Held in Conjunction with SC 2018: The International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 71–80). IEEE. https://doi.org/10.1109/P3HPC.2018.00010

Werkhoven, B. Van, Maassen, J., Seinstra, F.J., & Bal, H.E. (2014). Performance models for CPU-GPU data transfers. *Proceedings – 14th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2014* (pp. 11–20). IEEE. https://doi.org/10.1109/CCGrid.2014.16

Yang, W., Li, K., & Li, K. (2017). A hybrid computing method of SpMV on CPU–GPU heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, *104*, 49–60. https://doi.org/10.1016/j.jpdc.2016.12.023