Streaming Applications on Heterogeneous Platforms

Zhaokui Li(✉), Jianbin Fang, Tao Tang, Xuhao Chen, and Canqun Yang

Software Institute, College of Computer,
National University of Defense Technology, Changsha, China
zhaokuili@yeah.net, {j.fang,tangtao84,chenxuhao,canqun}@nudt.edu.cn

Abstract. Using multiple streams can improve the overall system performance by mitigating the data transfer overhead on heterogeneous systems. Currently, very few cases have been streamed to demonstrate the streaming performance impact and a systematic investigation of *streaming necessity* and *how-to* over a large number of test cases remains a gap. In this paper, we use a total of 56 benchmarks to build a statistical view of the data transfer overhead, and give an in-depth analysis of the impacting factors. Among the heterogeneous codes, we identify two types of *non-streamable* codes and three types of *streamable* codes, for which a streaming approach has been proposed. Our experimental results on the CPU-MIC platform show that, with multiple streams, we can improve the application performance by up 90 %. Our work can serve as a generic flow of using multiple streams on heterogeneous platforms.

Keywords: Multiple streams · Heterogeneous platforms · Performance

1 Introduction

Heterogeneous platforms are increasingly popular in many application domains [13]. The combination of using a host CPU combined with a specialized processing unit (e.g., GPGPUs or Intel Xeon Phi) has been shown in many cases to improve the performance of an application by significant amounts. Typically, the host part of a heterogeneous platform manages the execution context while the time-consuming code piece is offloaded to the coprocessor. Leveraging such platforms can not only enable the achievement of high peak performance, but increase the *performance per Watt ratio*.

Given a heterogeneous platform, how to realize its performance potentials remains a challenging issue. In particular, programmers need to explicitly move data between host and device over `PCIe` before and/or after running kernels. The overhead counts when data transferring takes a decent amount of time, and determines whether to perform offloading is worthwhile [2,3,5]. To hide this overhead, overlapping kernel executions with data movements is required. To this end, *multiple streams* (or *streaming mechanism*) has been introduced, e.g., CUDA Streams [12], OpenCL Command Queues [16], and Intel's

G.R. Gao et al. (Eds.): NPC 2016, LNCS 9966, pp. 116–129, 2016.
DOI: 10.1007/978-3-319-47099-3_10

hStreams [8]. These implementations of *multiple streams* spawn more than one streams/pipelines so that the data movement stage of one pipeline overlaps the kernel execution stage of another[1].

Prior works on multiple streams mainly focus on GPUs and the potential of using multiple streams on GPUs is shown to be significant [4,7,9,17]. Liu et al. give a detailed study into how to achieve optimal task partition within an analytical framework for AMD GPUs and NVIDIA GPUs [9]. In [4], the authors model the performance of asynchronous data transfers of CUDA streams to determine the optimal number of streams. However, these studies have shown very limited number of cases, which leaves two questions unanswered: (1) whether each application is required and worthwhile to use multiple streams on a given heterogeneous platform?, (2) whether each potential application is *streamable* or *overlappable*? If so, how can we stream the code?

To systematically answer these questions, we (1) build a statistical view of the data transfer (`H2D` and `D2H`) fraction for a large number of test cases, and (2) present our approach to stream different applications. Specifically, we statistically show that more than 50 % test cases (among 223) are not worthwhile to use multiple streams. The fraction of `H2D` varies over platforms, applications, code variants, and input configurations. Further, we identify two types of *non-streamable* codes (`Iterative` and `SYNC`) and three categories of *streamable* code based on task dependency (`embarrassingly independent`, `false dependent`, and `true dependent`). Different approaches are proposed to either eliminate or respect the data dependency. As case studies, we stream 13 benchmarks of different categories and show their streaming performance impact. Our experimental results show that using multiple streams gives a performance improvement ranging from 8 % to 90 %. To the best of our knowledge, this is the first comprehensive and systematic study of multiple streams in terms of both *streaming necessity* and *how-to*. To summarize, we make the following contributions:

- We build a statistical view of the data transfer fraction (R) with a large number of test cases and analyze its impacting factors (Sect. 3).
- We categorize the heterogeneous codes based on task dependency and present our approach to stream three types of applications (Sect. 4).
- We show a generic flow of using multiple streams for a given application: calculating R and performing code streaming (Sects. 3 and 4).
- We demonstrate the performance impact of using multiple streams on the CPU-MIC platform with 13 streamed benchmarks (Sect. 5).

2 Related Work

In this section, we list the related work on pipelining, multi-tasking, workload partitioning, multi-stream modeling, and offloading necessity.

[1] In the context, the streaming mechanism is synonymous with *multiple streams*, and thus we refer the *streamed code* as *code with multiple streams*.

Pipelinining is widely used in modern computer architectures [6]. Specifically, the pipeline stages of an instruction run on different functional units, e.g., arithmetic units or data loading units. In this way, the stages from different instructions can occupy the same functional unit in different time steps, thus improving the overall system throughput. Likewise, the execution of a heterogeneous application is divided into stages (`H2D`, `KEX`, `D2H`), and can exploit the idea of software pipelining on the heterogeneous platforms.

Multi-tasking provides concurrent execution of multiple applications on a single device. In [1], the authors propose and make the case for a GPU multitasking technique called *spatial multitasking*. The experimental results show that the proposed spatial multitasking can obtain a higher performance over cooperative multitasking. In [19], Wende et al. investigate the concurrent kernel execution mechanism that enables multiple small kernels to run concurrently on the Kepler GPUs. Also, the authors evaluate the Xeon Phi offload models with multi-threaded and multi-process host applications with concurrent coprocessor offloading [18]. Both multitasking and multiple streams share the idea of spatial resource sharing. Different from multi-tasking, using multiple streams needs to partition the workload of a single application (rather than multiple applications) into many tasks.

Workload Partitioning: There is a large body of workload partitioning techniques, which intelligently partition the workload between a CPU and a coprocessor at the level of algorithm [20,21] or during program execution [14,15]. Partitioning workloads aims to use unique architectural strength of processing units and improve resource utilization [11]. In this work, we focus on how to efficiently utilize the coprocessing device with multiple streams. Ultimately, we need to leverage both workload partitioning and multiple streams to minimize the end-to-end execution time.

Multiple Streams Modeling: In [4], Gomez-Luna et al. present performance models for asynchronous data transfers on different GPU architectures. The models permit programmers to estimate the optimal number of streams in which the computation on the GPU should be broken up. In [17], Werkhoven et al. present an analytical performance model to indicate when to apply which overlapping method on GPUs. The evaluation results show that the performance model are capable of correctly classifying the relative performance of the different implementations. In [9], Liu et al. carry out a systematic investigation into task partitioning to achieve maximum performance gain for AMD and NVIDIA GPUs. Unlike these works, we aim to evaluate the necessity of using multiple streams and investigate how to use streams systematically. Using a model on Phi to determine the number of streams will be investigated as our future work.

Offloading Necessity: Meswani et al. have developed a framework for predicting the performance of applications executing on accelerators [10]. Using automatically extracted application signatures and a machine profile based on benchmarks, they aim to predict the application running time before the application is ported. Evaluating offloading necessity is a former step of applying multiple streams. In this work, we evaluate the necessity of using multiple streams with a statistical approach.

3 A Statistical View

In this section, we give a statistical view of how many applications are worthwhile to be streamed on heterogeneous platforms, and analyze the factors that impact the streaming necessity.

3.1 Benchmarks and Datasets

As shown in Table 1, we use a large number of benchmarks that cover a broad range of interesting applications domains for heterogeneous computing. These benchmarks are from the Rodinia Benchmark Suite, the Parboil Benchmark Suite, the NVIDIA SDK, and the AMD APP SDK. In total, we employ 56 benchmarks and 223 configurations. The details about how applications are configured are summarized in Table 1. Note that we remove the redundant applications among the four benchmark suites when necessary.

Table 1. Applications, inputs and configurations

Suite	Applications	Input	Applications	Input
Rodinia (18)	backprop	$10 \times \{2^{16}, 2^{17}, 2^{18}, 2^{19}, 2^{20}\}$	bfs	graph{512K, 1M, 2M, 4M, 8M}
	b+tree	Kernel1, Kernel2	cfd	0.97K, 193K, 0.2M
	dwt2d,gaussian, lud	$2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}$	myocyte, srad	100, 200, 300, 400, 500
	hearwall, lavaMD, leukocyte	10, 20, 30, 40, 50	hotspot	$2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}$
	kmeans	$\{1, 3, 10, 30, 100\} \times 100000$	nn, nw	$2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}$
	pathfinder	$(\{1, 2, 4\} \times 10^5, \{100, 200, 400\})$	streamcluster	$100 \times \{2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}\}$
Parboil (9)	spmv	small, medium, large	stencil	small, default
	mri-gridding	small	cutcp	small, large
	tpacf	small, medium, large	bfs	1M, NY, SF, UT
	sgemm	small, medium	mri-q	small, large
	lbm	short, long		
NVIDIA SDK (17)	BlackScholes	$10^6 \times \{4, 8, 12, 16, 20\}$	ConvolutionSeparable	$2^{10} \times \{1, 2, 3, 4, 8\}$
	DCT8x8	$2^{10} \times \{1, 2, 3, 4, 8\}$	DotProduct	$2^{10} \times 10^3 \times \{1, 2, 3, 4, 8\}$
	DXTCompression	lena	FDTD3d	10, 20,30, 40, 50
	Histogram	1, 2, 3, 4, 5	MatrixMul	6, 7, 8, 9, 10
	MatVecMul	256, 128, 64, 32, 16	QuansiRandomGenerator	$2^{10} \times 10^3 \times \{1, 2, 3, 4, 8\}$
	Reduction	$2^{10} \times 10^3 \times \{1, 2, 3, 4, 8\}$	Reduction-2	$2^{10} \times 10^3 \times \{1, 2, 3, 4, 8\}$
	Transpose	$2^{10} \times \{1, 2, 3, 4, 8\}$	Tridiagonal	32, 64, 128, 256, 512
	VectorAdd	$2^{10} \times \{1, 2, 4, 8, 16\}$	FastWalshTransform	8M
			ConvolutionFFT2D	62500k
AMD SDK (12)	BinomialOption	$2^{10} \times \{1, 2, 4, 8, 16\}$	BitonicSort	$2^{20} \times \{1, 2, 4, 8, 16\}$
	BoxFilter	BoxFilter_Input	DwtHaar1D	$2^{10} \times 10^3 \times \{1, 2, 3, 4, 8\}$
	FloydWarshall	$2^{10} \times \{1, 2, 3, 4, 5\}$	MonteCarloAsian	$2^{10} \times \{1, 2, 3, 4, 5\}$
	RadixSort	$2^{12} \times \{12, 13, 14, 15, 16\}$	RecursiveGaussian	default
	ScanLargeArrays	$2^{10} \times \{1, 2, 4, 8, 16\}$	StringSearch	1, 2, 3, 4, 5
	URNG	1, 2, 3, 4, 5	PrefixSum	1024k

3.2 Experimental Platforms

The heterogeneous platform used in this work includes a dual-socket Intel Xeon CPU (12 cores for each socket) and an Intel Xeon 31SP Phi (57 cores for each card). The host CPUs and the cards are connected by a `PCIe` connection. As for the software, the host CPU runs Redhat Linux v7.0 (the kernel version is 3.10.0-123.el7.x86_64), while the coprocessor runs a customized uOS (v2.6.38.8). Intel's MPSS (v3.6) is used as the driver and the communication backbone between the host and the coprocessor. Also, we use Intel's multi-stream implementation `hStreams (v3.5.2)` and Intel's OpenCL SDK (v14.2). Note that the applications in Table 1 are in OpenCL, while the pipelined versions are in `hStreams`.

3.3 Measurement Methodology

A typical heterogeneous code has three parts: (1) transferring data from host to device (`H2D`), (2) kernel execution (`KEX`), and (3) moving data from device back (`D2H`). To measure the percentage of each stage, we run the codes in a strictly stage-by-stage manner. Moreover, we perform 11 runs and calculate the median value. Before uploading datasets, buffer allocation on the device side is required. Due to the usage of the *lazy allocation policy*, the allocation overhead is often counted into `H2D`. Thus, we argue that `H2D` might be larger than the actual host-to-device data transferring time.

3.4 Results and Analysis

We define *data transfer ratio (R)* as the fraction of the data transfer time to the total execution time, and take this metric (R) as an indicator of whether it is necessary to use multiple streams. Figure 1 shows the CDF distribution of the `H2D` and `D2H` duration versus the overall execution time (R_{H2D} and R_{D2H}). We observe that the CDF is over 50 % when $R_{H2D} = 0.1$. That is, the `H2D` transfer time takes less than 10 % for more than 50 % configurations. Meanwhile, the number is even larger (around 70 %) for the `D2H` part. In the remaining contents, we will focus on R_{H2D} and use R (instead of R_{H2D}) for clarity.

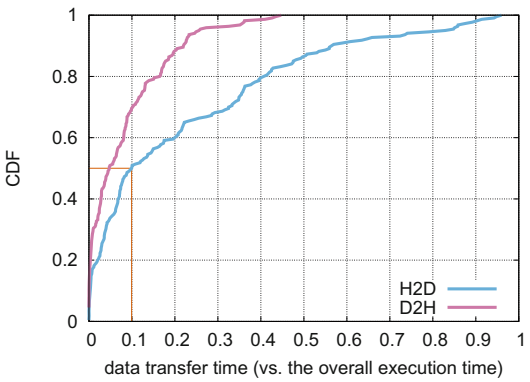
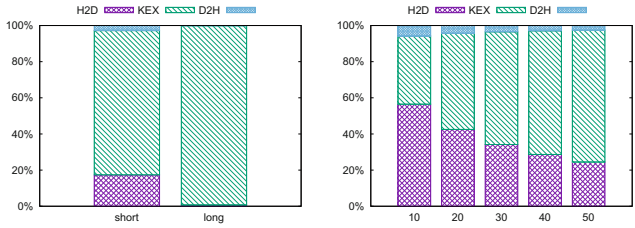


Fig. 1. The CDF curve for data transfers between the host and the accelerator.

The Impact of Input Datasets. Typically, the `H2D` ratio will remain when changing the input datasets. This is because the computation often changes linearly over the data amount. But this is not necessarily the case. Figure 2 shows how R changes with the input datasets for `lbm` and `FDTD3d`, respectively. We note that, for `lbm`, using the `short` configuration takes a decent amount of time to move data from host to device, while the data amount takes a much smaller proportion for the `long` configuration. For `FDTD3d`, users have to specify

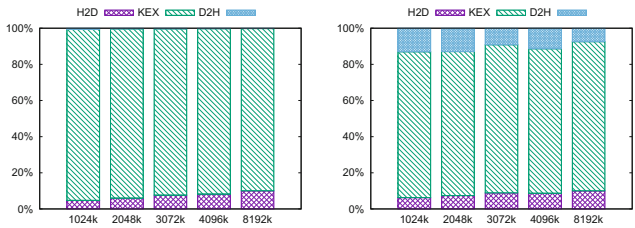
the number of time steps according to their needs. We note that the kernel execution time increases over time steps. When streaming such applications, it is necessary to focus on the commonly used datasets.



(a) lbm. (b) FDTD3d.

Fig. 2. `R` changes over datasets for `lbm` and `FDTD3d`.

The Impact of Code Variants. Figure 3 shows how data transfers change with the two code variants for `Reduction`. `Reduction v1` performs the whole reduction work on the accelerator, thus significantly reducing the data-moving overheads. Meanwhile, `Reduction v2` performs the final reduction on the host side, and thus needs to transfer the intermediate results back to host. Therefore, different code variants will generate different data transferring requirements, which is to be taken into account when streaming such code variants.

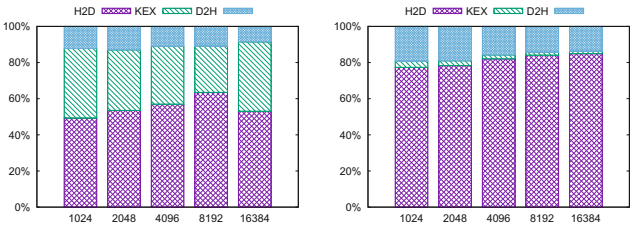


(a) reduction v1. (b) reduction v2.

Fig. 3. `R` changes over code variants of `NVIDIA Reduction`.

The Impact of Platform Divergence. Figure 4 shows how R changes on MIC and a K80 GPU [2]. We see that the kernel execution time (of `nn`) on the MIC occupies 33 % on average while the number is only around 2 % on the GPU. This is due to the huge processing power from NVIDIA K80, which reduces the `KEX` fraction significantly. Ideally, using the streaming mechanism can improve the overall performance by 2 % on the GPU. In this case, we argue that it is unnecessary to use multiple streams on GPU.

[2] Note that the only difference lies in devices (Intel Xeon 31SP Phi versus NVIDIA K80 GPU) and all the other configurations are the same.



(a) on MIC. (b) on GPU.

Fig. 4. `R` changes over platforms of `Rodinia nn`.

To summarize, we observe that R varies over platforms, benchmarks, code variants, and input configurations. Each benchmark has a unique balance between computation and memory accesses. Different code variants lead to the differences in transferred data amounts and kernel execution time. Also, input configurations can incur changes in transferred data amounts and/or kernel execution time. Furthermore, R depends on hardware capabilities (e.g., the PCIe interconnect and the accelerator).

We use R as an indicator of deciding whether the target application is worthwhile to be streamed. Figure 1 shows that `H2D` takes only 10 % of the total execution time for over 50 % test cases. We argue that, on the one hand, the applications are not worthwhile to be streamed when R is small. This is due to two factors: (1) code streaming introduces overheads for filling and emptying the pipeline, and (2) streaming an application requires extra programming efforts from reconstructing data structures and managing streams. Thus, streaming such applications might lead to a performance degradation compared with the non-streamed code. On the other hand, when R is too large (e.g., 90 %), it is equally not worthwhile to apply streams. When the fraction of `H2D` is too large, using accelerators may lead to a performance drop (when comparing to the case of only using CPUs), not to mention using streams. In real-world cases, users need make the streaming decision based on the value of R and the coding effort.

4 Our Streaming Approach

4.1 Categorization

After determining the necessity to apply the pipelining/streaming mechanism, we further investigate *how-to*. Generally, we divide applications into tasks which are mapped onto different processing cores. As we have mentioned above, each task includes the subtasks of data transfers and kernel execution. To pipeline codes, we should guarantee that *there exist independent tasks* running concurrently. Once discovering independent tasks, we are able to overlap the execution of `H2D` from one task and `KEX` from another (Fig. 5(a)). For a single task, `H2D` is dependent on `KEX`.

In practice, more than one `H2D` may depend on a single kernel execution (Fig. 5(b)). Thus, we need to analyze each `H2D`–`KEX` dependency to determine whether each pair can be overlapped. Moreover, an application often has more than one kernel. Implicitly, each kernel is synchronized at the end of its execution. Therefore, the kernel execution order is strictly respected within a single task. Figure 5(c) shows that `H2D(1)` is depended by `KEX(1)`, but the data is not used til the execution of `KEX(2)`. Thus, this data transfer can be delayed right before `KEX(2)` when analyzing dependency and/or streaming the code.

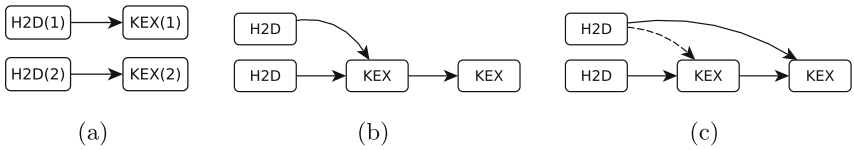


Fig. 5. The dependent relationship between `H2D` and `KEX` (The number in the parenthesis represents stages from different tasks).

Based on the dependency analysis, we categorize the codes listed in Table 1 as *streamable codes* (Sect. 4.2) and *non-streamable codes*. The first pattern (`SYNC`) of *non-streamed codes* is when the `H2D` data is shared by all the tasks of an application. In this case, the whole data transfer has to be finished before kernel execution. The second non-streamable pattern is characterized as `Iterative`, for which `KEX` will be invoked in an iterative manner once the data is located on device. Although such cases can be streamed by overlapping the data transfer and the first iteration of kernel execution, we argue that the overlapping brings no performance benefit for a large number of iterations.

We analyze the heterogeneous codes listed in Table 1 and categorize them in Table 2. Each nonstreamable code is labeled with `SYNC` or `Iterative`. We note that the kernel of `hearwall` has such a large number of lines of codes and complex structures that its execution takes a major proportion of the end-to-end execution time. It is unnecessary to stream such code on any platform. Due to the multiple `H2D`–`KEX` dependency pairs, an application might fall into more than one category (e.g., `streamcluster`). Also, the kernel of `myocyte` runs sequentially and thus there are no concurrent tasks for the purpose of pipelining. For the streamable codes, we group them into three categories, which are detailed in Sect. 4.2.

4.2 Code Streaming

We divide the streamable/overlappable applications into three categories based on task dependency: (1) embarrassingly independent, (2) false dependent, and (3) true dependent. Tasks are generated based on input or output data partitioning, and thus task dependency shows as a form of data dependency. We will explain them one by one.

Table 2. Application categorization.

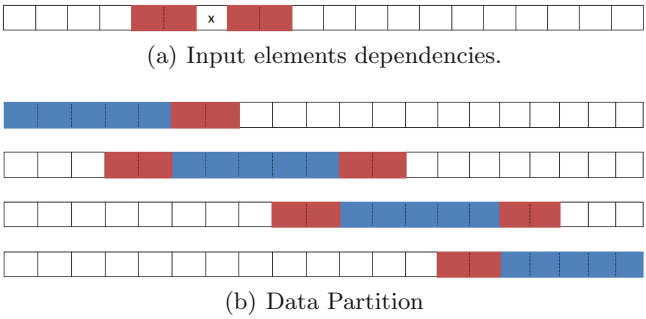
	Nonstreamable		Streamable		
	SYNC	**Iterative**	**Independent**	**False-dependent**	**True-dependent**
Rodinia	backprop, bfs, b+tree, Kmeans, streamcluster	hotspot, pathfinder	backprop, dwt2d, nn, srad, streamcluster	lavaMD, leukocyte	gaussian, lud nw
Parboil	spmv, tpacf, bfs, mri-q	mri-gridding, cutcp	sgemm	stencil, lbm	
NVIDIA SDK	Reduction-2,	FDTD3d, DXTCompression	BlackScholes, DCT8x8, DotProduct, Histogram, MatrixMul, MatVecMul, QuansirandomGenerator, Reduction, Transpose, Tridiagonal, VectorAdd	ConvolutionSeparable FastWalshTransform ConvolutionFFT2D	
AMD SDK		BitonicSort, FloydWarshall, RadixSort	BinomialOption, MonteCarloAsian, RecursiveGaussian, ScanLargeArrays, URNG, PrefixSum	BoxFilter, DwtHaar1D, StringSearch	

Embarrassingly Independent. Tasks from such overlappable applications are completely independent. Thus, there is no data dependency between tasks. Taking `nn` (nearest neighbor) for example, it finds the k-nearest neighbors from an unstructured data set. The sequential `nn` algorithm reads in one record at a time, calculates the Euclidean distance from the target latitude and longitude, and evaluates the k nearest neighbors. By analyzing its code, we notice no dependency between the input data. Figure 6 shows how to partition the input data. Assuming 16 elements in the set, we divide them into 4 groups, which represent 4 tasks. Then we spawn streams to run the tasks. Due to no dependency, data transferring from one task can overlap kernel execution from another. More `Embarrassingly Independent` applications are shown in Table 2.

Fig. 6. Nearest neighbor data partition (16 elements and 4 groups).

False Dependent. There exist data dependencies in such overlappable applications, but the dependencies come from read-only data (i.e., *RAR* dependency). In this case, two tasks will share common data elements. A straightforward solution to this issue is that each task moves the shared data elements separately. For example, `FWT` (fast walsh transform) is a class of generalized Fourier transformations. By analyzing the code, we find that there are dependencies between the input data elements: As shown in Fig. 7(a), calculating element x is related to the 4 neighbors which are marked in red. The input elements are read-only, so we can eliminate the relationship by redundantly transferring boundary elements (Fig. 7(b)). We first divide the total elements into four blocks, corresponding

to four tasks (in blue). Then, we additionally transfer the related boundary elements (in red) when dealing with each data block. More `False Dependent` applications are shown in Table 2.



(a) Input elements dependencies.

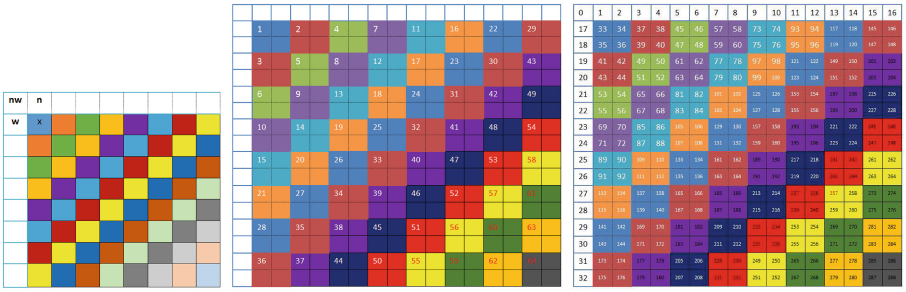
(b) Data Partition

Fig. 7. Task dependency and data partition for `FWT`

True Dependent. The third category of overlappable applications is similar to the second one in that there exist data dependencies between tasks. The difference is that the dependency is true (i.e., *RAW*). This is complicated for programmers not only because there is a dependence between the input data elements, but because they need to update input data in the process of calculation. Thus, the output elements depend on the updated input data, and we must control the order of calculation. The key for this pattern is to discover concurrency while respecting the dependency.

`NW`, Needleman-Wunsch is a nonlinear global optimization method for DNA sequence alignments. The potential pairs of sequences are organized in a 2D matrix. In the first step, the algorithm fills the matrix from top left to bottom right, step-by-step. The optimum alignment is the pathway through the array with maximum score, where the score is the value of the maximum weighted path ending at that cell. Thus, the value of each data element depends on the values of its northwest-, north- and west-adjacent elements. In the second step, the maximum path is traced backward to deduce the optimal alignment. As shown in Fig. 8(a), calculating element x is related with three elements: 'n' (north-element), 'w' (west-element), and 'nw' (northwest-element). We must calculate output elements diagonal by diagonal (in the same color), and the elements on the same diagonal can be executed concurrently. Figure 8(b) shows how we divide the data: we number all blocks from the top-left diagonal to the bottom-right one (the first row and first column are the two DNA sequences, marked in number 0), and then change the storage location to let elements from the same block stored contiguously. Figure 8(c) shows the storage pattern, and the numbers represent the relative location. By controlling the execution in the order of diagonal from top-left to bottom-right, we can respect the dependencies between tasks.

Further, the tasks on the same diagonal can run concurrently with multiple streams. Note that the number of streams changes on different diagonals. More `True Dependent` applications can be found in Table 2.



(a) Dependencies. (b) Block Number. (c) Storage Pattern.

Fig. 8. `NW` input elements dependencies and how to partition.

5 Experimental Results

In this section, we discuss the performance impact of using multiple streams. We use the CPU-MIC heterogeneous platform detailed in Sect. 3.2. Due to the limitation in time and space, we port 13 applications from Table 1 with `hStreams`. As shown in Table 2, these 13 benchmarks are characterized as different categories and thus we use the corresponding approach to stream them.

Figure 9 shows the overall performance comparison. We see that using multiple streams outperforms using a single stream, with a performance improvement of 8 %–90 %. In particular, for `nn`, `FastWalshTransform`, `ConvolutionFFT2D`, and `nw`, the improvement is around 85 %, 39 %, 38 %, and 52 %, respectively. However, for applications such as `lavaMD`, we cannot obtain the expected performance improvement with multiple streams, which will be discussed in the following.

Also, we notice that the performance increase of using multiple streams varies over benchmarks and datasets. This is due to the differences in data transfer ratio (R): a larger R leads to a greater performance improvement. For example, for `ConvolutionSeparable` and `Transpose`, the average performance improvement is 45 % and 11 %, with R being 19 % and 14 %, respectively. Further, when selecting two datasets (400M and 64M) for `Transpose`, we can achieve a performance increase of 14 % and 8 %, with R being 20 % and 10 %, respectively.

For the `False Dependent` applications, if the extra overhead of transferring boundary elements is nonnegligible, code streaming is not beneficial. For `FWT`, one element is related to 254 elements which is far less than the subtask data size of 1048576. Therefore, although having to transfer extra boundary values, the overall streaming performance impact is positive. However, when the boundary elements are almost equal to the subtask size, the overhead introduced by

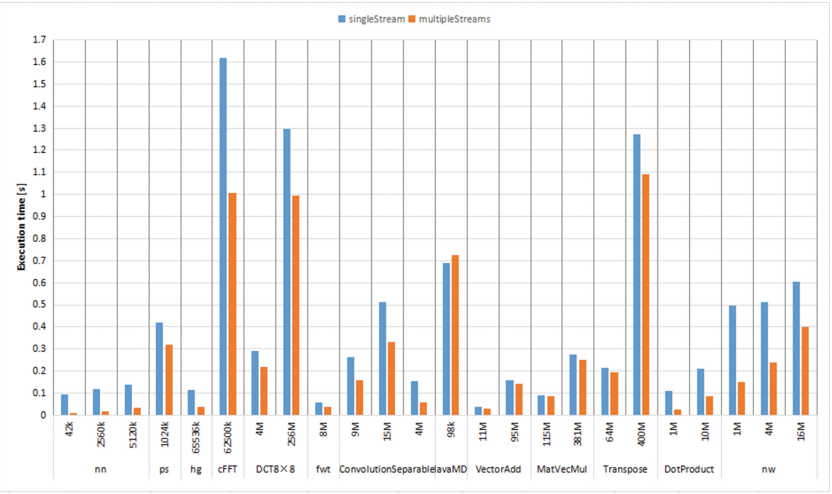


Fig. 9. A performance comparison between single stream and multiple streams. For each application, we employ different configuration, corresponding to different data size. Note that `ps`, `hg`, `cFFT` and `fwt` represents `PrefixSum`, `Histogram`, `ConvolutionFFT2D` and `FastWalshTransform`, respectively.

boundary transmission can not be ignored. `LavaMD` calculates particle potential and relocation due to mutual forces between particles within a large 3D space. In the experiment, one element for lavaMD depends on 222 elements, in which 111 elements are lying before the target element and the other half behind. The task data size is 250, which is close to the boundary element number. Thus, we cannot get the expected performance improvement, and the experimental results confirm our conclusion. Specifically, when the task is of 250 and remains unchanged, for single stream, the `H2D` and `KEX` time is 0.3476 s and 0.3380s, respectively. When using multiple streams, the overall execution time is 0.7242s. Therefore, it is not beneficial to stream the overlappable applications like `lavaMD`.

6 Conclusion

In this paper, we summarize a systematic approach to facilitate programmers to determine whether the application is required and worthwhile to use streaming mechanism, and how to stream the code. (1) obtaining the ratio R: run the codes in stage-by-stage manner, record the `H2D` and `KEX` time, and calculate R; (2) judging whether the application is overlappable; (3) streaming the codes by either eliminating or respecting data dependency. Our experimental results on 13 streamed benchmarks show a performance improvement of upto 90 %.

The process of analyzing whether a code is streamable and transforming the code is manually performed. Thus, we plan to develop a compiler analysis and tuning framework to automate this effort. Based on the streamed code, we will

further investigate how to get optimal performance by setting a proper task and/or resource granularity. Ultimately, we plan to autotune these parameters leveraging machine learning techniques. Also, we want to investigate the streaming mechanism on more heterogeneous platforms, other than the CPU-MIC one.

Acknowledgment. We would like to thank the reviewers for their constructive comments. This work was partially funded by the National Natural Science Foundation of China under Grant No. 61402488, No. 61502514 and No. 61602501, the National High-tech R&D Program of China (863 Program) under Grant No. 2015AA01A301, the National Research Foundation for the Doctoral Program of Higher Education of China (RFDP) under Grant No. 20134307120035 and No. 20134307120031.

References

1. Adriaens, J.T., Compton, K., Kim, N.S., Schulte, M.J.: The case for GPGPU spatial multitasking. In: 2012 IEEE 18th International Symposium on High Performance Computer Architecture (HPCA), pp. 1–12. IEEE, February 2012
2. Boyer, M., Meng, J., Kumaran, K.: Improving GPU performance prediction with data transfer modeling. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW), pp. 1097–1106. IEEE, May 2013
3. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.-H., Skadron, K.: Rodinia: a benchmark suite for heterogeneous computing. In: IEEE International Symposium on Workload Characterization, 2009. IISWC 2009, pp. 44–54. IEEE, October 2009
4. Gómez-Luna, J., González-Linares, J.M., Benavides, J.I., Guil, N.: Performance models for asynchronous data transfers on consumer graphics processing units. J. Parallel Distrib. Comput. **72**(9), 1117–1126 (2012)
5. Gregg, C., Hazelwood, K.: Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In: 2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 134–144. IEEE, April 2011
6. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 4th edn. Morgan Kaufmann, Burlington (2006)
7. Ino, F., Nakagawa, S., Hagihara, K.: GPU-chariot: a programming framework for stream applicationsrunning on multi-GPU systems. IEICE Trans. **96–D**(12), 2604–2616 (2013)
8. Intel Inc. hStreams Architecture document for Intel MPSS 3.5, April 2015
9. Liu, B., Qiu, W., Jiang, L., Gong, Z.: Software pipelining for graphic processing unit acceleration: partition, scheduling and granularity. Int. J. High Perform. Comput. Appl. **30**(2), 169–185 (2015)
10. Meswani, M.R., Carrington, L., Unat, D., Snavely, A., Baden, S., Poole, S.: Modeling and predicting performance of high performance computing applications on hardware accelerators. Int. J. High Perform. Comput. Appl. **27**(2), 89–108 (2013)
11. Mittal, S., Vetter, J.S.: A survey of CPU-GPU heterogeneous computing techniques. ACM Comput. Surv. **47**(4), 36 (2015)
12. NVIDIA Inc. CUDA C Best Practices Guide Version 7.0, March 2015
13. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. Proc. IEEE **96**(5), 879–899 (2008)

14. Pienaar, J.A., Raghunathan, A., Chakradhar, S.: MDR: performance model driven runtime for heterogeneous parallel platforms. In: Proceedings of the International Conference on Supercomputing, ICS 2011, pp. 225–234. ACM, New York (2011)
15. Takizawa, H., Sato, K., Kobayashi, H.: SPRAT: runtime processor selection for energy-aware computing. In: 2008 IEEE International Conference on Cluster Computing, pp. 386–393. IEEE (2008)
16. The Khronos OpenCL Working Group. OpenCL - The open standard for parallel programming of heterogeneoussystems, January 2016. http://www.khronos.org/opencl/
17. Werkhoven, B.V., Maassen, J., Seinstra, F.J., Bal, H.E.: Performance models for CPU-GPU data transfers. In: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 11–20. IEEE, May 2014
18. Wende, F., Steinke, T., Cordes, F.: Concurrent kernel execution on xeon phi within parallel heterogeneous workloads. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014. LNCS, vol. 8632, pp. 788–799. Springer, Heidelberg (2014). doi:10.1007/978-3-319-09873-9_66
19. Wende, F., Steinke, T., Cordes, F.: Multi-threaded kernel offloading to GPGPU using hyper-Q on kepler architecture. Technical report 14–19, ZIB, Takustr. 7, 14195 Berlin (2014)
20. Yang, C., Wang, F., Du, Y., Chen, J., Liu, J., Yi, H., Lu, K.: Adaptive optimization for petascale heterogeneous CPU/GPU computing. In: 2010 IEEE International Conference on Cluster Computing (CLUSTER), pp. 19–28. IEEE (2010)
21. Yang, C., Xue, W., Fu, H., Gan, L., Li, L., Xu, Y., Lu, Y., Sun, J., Yang, G., Zheng, W.: A peta-scalable CPU-GPU algorithm for global atmospheric simulations. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2013, pp. 1–12. ACM, New York (2013)