Parallel Processing Letters
Vol. 26, No. 4 (2016) 1640002 (18 pages)

DOI: 10.1142/S0129626416400028

World Scientific
www.worldscientific.com

Evaluating Multiple Streams on Heterogeneous Platforms

Jianbin Fang*, Peng Zhang, Zhaokui Li, Tao Tang, Xuhao Chen,
Cheng Chen and Canqun Yang

Software Institute, College of Computer,
National University of Defense Technology,
Changsha, 410073, China
**j.fang@nudt.edu.cn.*

Received October 2016
Revised October 2016
Communicated by Guest Editors
Published 21 December 2016

ABSTRACT

Using *multiple streams* can improve the overall system performance by mitigating the data transfer overhead on heterogeneous systems. Prior work focuses a lot on GPUs but little is known about the performance impact on (Intel Xeon) Phi. In this work, we apply multiple streams into six real-world applications on Phi. We then systematically evaluate the performance benefits of using multiple streams. The evaluation work is performed at two levels: the microbenchmarking level and the real-world application level. Our experimental results at the microbenchmark level show that data transfers and kernel execution can be overlapped on Phi, while data transfers in both directions are performed in a serial manner. At the real-world application level, we show that both overlappable and non-overlappable applications can benefit from using multiple streams (with an performance improvement of up to 24%). We also quantify how task granularity and resource granularity impact the overall performance. Finally, we present a set of heuristics to reduce the search space when determining a proper task granularity and resource granularity. To conclude, our evaluation work provides lots of insights for runtime and architecture designers when using multiple streams on Phi.

Keywords: Performance evaluation; multiple streams; resource partitioning; pipelining.

1. Introduction

Heterogeneous platforms are increasingly popular in many application domains [1]. 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 addition to procurement cost, significant programming and porting effort is required to realized the performance benefit [2]. 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 [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 [6], OpenCL Command Queues [7], and Intel's hStreams [8]. These implementations of *multiple streams* spawn more than one streams/pipelines so that the data movement stage of one pipeline overlap the kernel execution stage of another.[a]

Prior works on multiple streams mainly focus on GPUs and the potential of using multiple streams on GPUs is shown to be significant [9–12]. Liu et al. give a detailed study into how to achieve optimal task partitions within an analytical framework for AMD GPUs and NVIDIA GPUs [11]. In [10], the authors model the performance of asynchronous data transfers of CUDA streams to determine the optimal number of streams. However, little is known about how multiple streams behave on the OS-enabled coprocessor such as Intel Xeon Phi. For such coprocessors, programmers can explicitly map streams to different groups of cores, i.e., they have control of *resource granularity*. This control on GPUs is not exposed to programmers. Thus, how resource granularity impacts the overall performance and how to determine a proper resource granularity on Phi is unknown.

To answer these questions and gain insights of using multiple streams on Phi, we provide a systematic performance evaluation. Specifically, we evaluate the performance impact of multiple steams at two levels: the microbenchmarking level and the real-world application level. At the microbenchmarking level, we measure the overlapping capability of Phi with multiple streams including ① the overlapping of data transfers in both directions, ② the overlapping of data transfers with kernel execution, and ③ performance potentials from resource partitioning. At the real-world application level, we first give a performance comparison of applications with and without using multiple streams, and then provide an in-depth analysis of the performance factors by using six different real-world applications. Also, we present a set of heuristics to reduce the huge search space when selecting a proper task granularity and resource granularity. Our preliminary results on multiple Phis show a significant performance improvement (over 1 Phi) without code changes and we conclude that using multiple streams is a promising programming tool for multiple devices.

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

To summarize, our contributions are as follows.

- We apply the streaming mechanism to 7 (6 + 1) applications representative of different domains and patterns. With them, we systematically evaluate the capability of multiple streams on Phi.
- We quantify how each performance factor impacts the application performance and perform an in-depth analysis on the performance changes.
- We present a set of guidelines to significantly reduce the search space when determining task granularity and/or resource granularity, based on our observations.
- We give a preliminary performance evaluation on multiple MICs with multiple streams.

The rest of this paper is organized as follows: Section 2 gives a brief description of multiple streams in terms of temporal resource sharing and spatial resource sharing, a prototype implementation of multiple streams, and the related work. Section 4 introduces the experimental setup and the benchmarks. We provide a systematic performance evaluation of multiple streams with microbenchmarks in Section 5 and with real-world applications in Section 6. Section 7 discusses the performance of multiple streams on multiple devices and Section 8 concludes the work.

2. Background

In this section, we introduce multiple streams in terms of temporal sharing and spatial sharing, describe a multiple stream prototype (`hStreams`), and give the related work.

2.1. *Multiple streams*

2.1.1. *Temporal sharing*

Using multiple streams can overlapping computation and communication (data transfers), thus realizing the *temporal sharing* of resources. On the whole, we divide the offloading process of heterogeneous applications into three stages: (1) move data from host to device (`H2D`), (2) kernel execution (`EXE`), and (3) move data from device to host (`D2H`). Overlapping these three stages will significantly improve the overall performance. Figure 1 shows an illustrative comparison between single stream and multiple streams. Suppose that these stages consume equal amount of time for a given task. When using a single stream (i.e., the steps run in a serial manner), the code takes 6 time units to finish two tasks. Meanwhile, the streamed code will finish four tasks (using four streams) in the same amount of time.

2.1.2. *Spatial sharing*

Using multiple streams also enjoys the idea of resource partitioning. That is, to partition the resource (e.g., processing cores) into multiple groups and map each

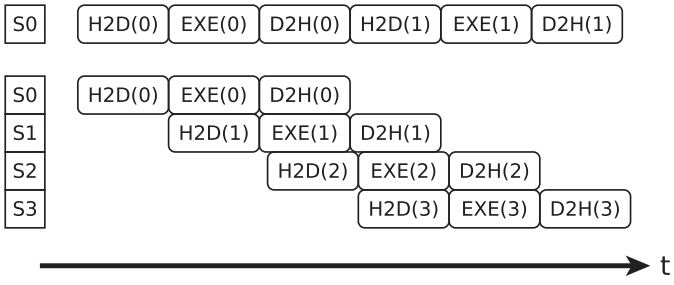


Fig. 1. Temporal sharing (S_x represents a stream labeled with x).

stream onto a partition. Therefore, different streams can run on different partitions concurrently, i.e., resource *spatial sharing*. Nowadays accelerators have a large number of processing units that some applications cannot efficiently exploit them for a given task. Typically, we offload a task and let it occupy all the processing cores. When using multiple streams, we divide the processing cores into multiple groups (each group is named as a *partition*). Figure 2 shows that a device has 16 cores and is logically divided into four partitions (`P0`, `P1`, `P2`, `P3`). Then different tasks are offloaded onto different partitions, e.g., `T0`, `T1`, `T2`, `T3` runs on `P0`, `P1`, `P2`, `P3`, respectively. In this way, we aim to improve the device utilization.

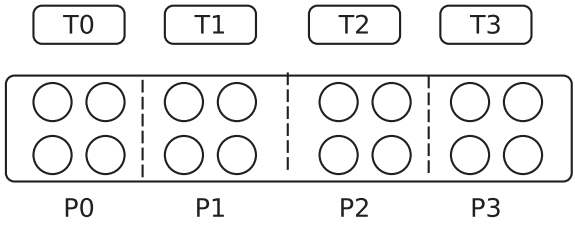


Fig. 2. Spatial sharing. The circles represent processing cores, `Tx` represents a task, and `Px` represents a partition.

2.2. *hStreams*

`hStreams` is an open-source streaming implementation from Intel [13]. At its core is the resource partitioning mechanism [8]. Figure 3 shows the mapping between logical concepts and a physical machine (e.g., Intel Xeon Phi). At the physical level, the whole device is partitioned into multiple groups and thus each group has several processing cores. At the logical level, a device can be seen as one or more *domains*. Each domain contains multiple *places*, each of which then has multiples *streams*. The logical concepts are visible to programmers, while the physical ones are transparent to them and the mapping between them are automatically handled by the runtime.

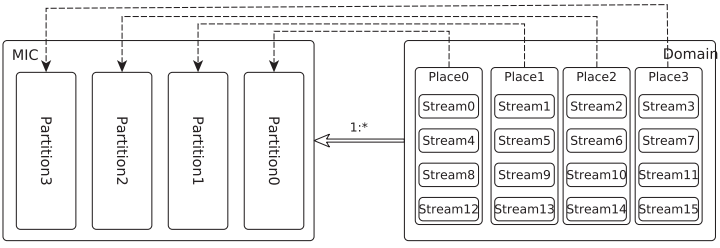


Fig. 3. `hStreams` resource view.

`hStreams` is implemented as a library and provides users with APIs to access coprocessors/accelerators efficiently. Programming with `hStreams` resembles that in `CUDA` or `OpenCL`. Programmers have to create the streaming context, move data between host and device, and invoke kernel execution. And they also have to split tasks to use multiple streams. Further, `hStreams` cannot be used alone, but has to used with other programming models such as `OpenMP` or Intel `TBB`.

3. Related work

Our work relates to pipelining, multi-tasking, partitioning workloads between the host and devices or among devices, modeling streams, and offloading necessity.

Pipelinining is widely used in modern computer architectures [14]. 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 slices, thus improving the overall system throughput. Likewise, a heterogeneous application is divided into stages (`H2D`, `EXE`, `D2H`), and can exploit the idea of software pipelining on the heterogeneous platforms (as mentioned in Section 2.1).

Multi-tasking provides concurrent execution of multiple applications (kernels) on a single device. In [15], Adriaens et al. 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 [16], 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 [17]. 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. Advanced streaming also supports multi-tasking, which is beyond the scope of this paper.

Workload Partition: 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 [18, 19] or during program execution [20, 21]. Partitioning

workloads aims to use unique architectural strength of processing units and improve resource utilization [22]. 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. Partitioning workloads is also required among coprocessors [23]. In [24], Planas et al. present AMA (Asynchronous Management of Accelerators) that combines several ideas of managing and scheduling computations to multiple accelerators. Their main target is a task-based programming framework of improving the management of multi-accelerator systems with a minimized overhead. The streaming prototype used in this paper also supports multi-device programming, which requires no code modification and thus is transparent to programmers. Using this prototype, we give our preliminary performance results on a MIC-based heterogeneous platform with multiple Phis (Section 7).

Multiple Streams Modeling: In [10], 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 [9], 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 is capable of correctly classifying the relative performance of the different implementations. In [11], 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 discuss the heuristics of reducing the search space when determining the factors. Using a model on Phi 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 [2]. Using automatically extracted application signatures and a machine profile based on benchmarks, they aim to predict the application running time rapidly and accurately without going to the considerable effort of porting and tuning. Evaluating offloading necessity is a former step of applying multiple streams. In this work, we focus on applying the streaming mechanism and evaluating its performance impact.

To summarize, using multiple streams enjoys the idea of pipelining and resource management. Selected applications have been evaluated on GPGPUs, while very few works are noticed on Intel Xeon Phi. Our work evaluates the streaming mechanism on such platforms and its performance impact. In addition, we measure the overlapping capability with microbenchmarks. To the best of our knowledge, this is the first systematic performance evaluation of multiple streams on the MIC-based heterogeneous platform with both microbenchmarks and real-world applications.

4. Experimental methodology

In this section, we first introduce the hardware and software environment, and then describe the used benchmarks.

4.1. *Platform configurations*

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 (v3.6)`.

4.2. *Benchmarks*

4.2.1. *Microbenchmark*

`hBench` is a microbenchmark used to quantify the capability of multiple steams in terms of temporal sharing and spatial sharing. The basic operation of the microbenchmark is $B[i] = A[i] + \alpha$. Before kernel execution, we need to move array A to the coprocessor, and move the output array B back once the kernel finishes execution. The compute complexity of the kernel execution is controlled by *iterations*. Simply, more iterations consume more computational time. The data movements and kernel execution can be either synchronous or asynchronous. With this microbenchmark, we aim to evaluate the capability of multiple streams on the Phi coprocessor.

4.2.2. *Real-world applications*

We use nine benchmarks, among which, `Matrix Multiplication (MM)` and `Cholesky Factorization (CF)` are from the hStreams SDK while the others are written by ourselves. `Kmeans`, `Hotspot`, `Nearest Neighbour (NN)`, `SRAD` from the Rodinia benchmark suite, `DCT`, `PrefixSum (PS)` from the AMD SDK, and `Dot Product (DP)` from the NVIDIA SDK are used in our work [4]. When porting them in hStreams, we partition the whole dataset into tiles, each of which represents a task. Then at least one task is mapped to a stream. The OpenMP regions are coded as kernels (like CUDA or OpenCL), which will be offloaded onto the coprocessor/sink side. We run each benchmark for 11 iterations, ignore the first iteration, and calculate the mean results.

5. Evaluating multiple streams with microbenchmarks

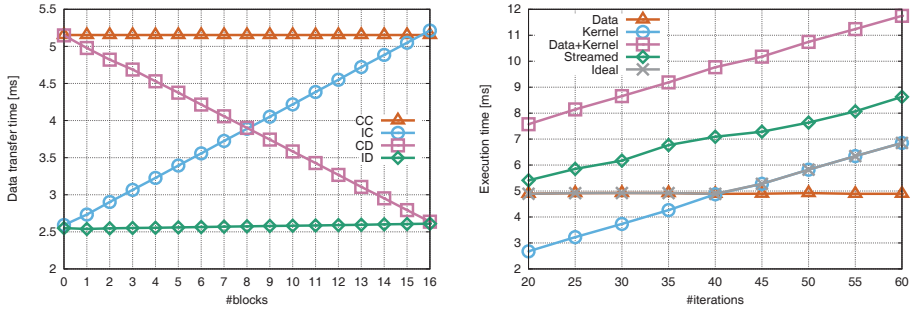
In this section, we evaluate the performance impact of using multiple streams from the perspective of temporal sharing and spatial sharing. First, we will evaluate the overlapping of data transfers and computations, and between different data transfers. Then, we will quantify the performance impact of using the resource partitioning (i.e., spatial sharing).

5.1. *Temporal sharing*

5.1.1. *Overlapping data transfers*

Data transfers can be performed from host to device and vice verse. However, whether data can be transferred from both directions concurrently depends on the target platform. Thus, we use `hBench` to measure the overlapping capability of data transfers. Specifically, it moves *hd* blocks of data elements from host to device, and moves *dh* blocks of data elements from device to host.

Figure 4(a) shows four cases when *dh* and *hd* are assigned with different values, and the block size is 1 MB. For `CC`, $hd = dh = 16$. We first transfer 16 data blocks from host to device, and then transfer another 16 blocks from device to host. Since the total amount of data blocks remains constant, the data transfer time does not change (5.2 ms). For `IC`, *hd* increases from 0 to 16 and $dh = 16$, while for `CD`, $hd = 16$ and *dh* decreases from 16 to 0. We observe that the data transfer time increases linearly over blocks for IC, while it decreases linearly for CD. For `ID`, *hd* increases from 0 to 16, *dh* decreases from 16 to 0, and $hd + dh = 16$. In this case, the total amount of transferred data keeps constant, and we notice the transfer time also remains around 2.5 ms. If moving data from host to device could overlap the data transfers in the other direction, the total transferring time will be dominated by the one with more data blocks, other than the sum of transferring time. Therefore, we conclude that data transfers from both directions are performed in a serial manner.



(a) (b)

Fig. 4. Temporal sharing: (a) How the data transfer time over the number of transferred blocks. (b) The overlapping extent of data transfers and computation when changing the number of kernel iterations.

5.1.2. *Overlapping data transfers with computation*

We then measure the overlapping potential of data transfers and computation with `hBench`. Apart from data transfers, we measure the kernel execution time. The kernel computes $B[i] = A[i] + \alpha$, where array A is transferred from host to device before kernel execution while array B is transferred from device to host as the

output. We control the computation amount by iterating the addition operation while keeping the size of array A and array B fixed (16 MB).

Figure 4(b) shows the overlapping of data transfers and computation. *Data* (the line with triangular marks) represents the data transferring time from both directions, and *Kernel* (the line with circle marks) represents the kernel execution time. Since the size of array A and B is constant, the total transferring time does not change with the number of kernel iterations. By contrast, the kernel execution time increases linearly as shown in Figure 4(b). These two lines intersect when using 40 iterations in the kernel. When using less than 40 iterations, the performance is dominated by data transfers (i.e., *dominant transfers*), and when using over 40 iterations, the performance is dominated by kernel execution (i.e., *dominant kernel*) [10]. We expect that the line with cross marks in Figure 4(b) represents a full overlap of data transfers and kernel execution. However, the measured execution time (the line with diamond marks) is longer than the expected execution time, illustrating the difficulty of achieving a full overlap.

5.2. *Spatial sharing*

We further use `hBench` to evaluate the performance impact of resource granularity. Specifically, we partition array A and B into 128 blocks, and use 100 kernel iterations. Figure 5 shows how the kernel execution (excluding the data transferring time) changes over resource granularity. We observe that, for a given task granularity, the execution time first decreases and then increases when changing the number of partitions. This is because partitioning resources can lead to a better utilization. Meanwhile, using too many partitions (and streams) also introduces extra management overheads.

The `ref` bar of Figure 5 shows the execution time of the non-streamed non-tiled code. We see that it is lower than that of the tiled streamed code. In other words, simply partitioning the hardware resources brings no performance improvement for the kernel execution. The root reason is that we explicitly make a synchro-

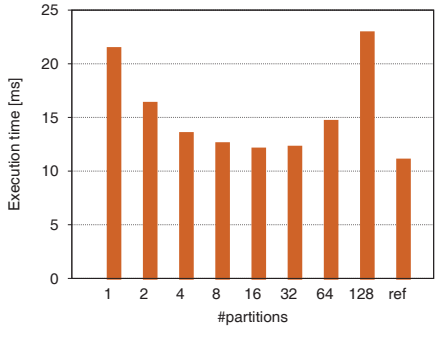


Fig. 5. How resource granularity impacts the overall performance. The `ref` bar represents the execution time of the non-streamed non-tiled code.

nization between data transfers and kernel execution, and the application is ***non-overlappable***. Our experimental results in Figure 4(b) and Figure 5 show that using multiple streams is beneficial only when the target application is ***overlappable***. Among the aforementioned applications in Section 4.2, `MM`, `CF`, `NN`, `DCT`, `PS`, and `DP` are overlappable, while `Hotspot`, `Kmeans`, and `SRAD` are non-overlappable. Therefore, we expect that the first three applications can benefit from temporal and spatial sharing of hardware resources.

6. Evaluating multiple streams with real-world applications

We use nine real-world applications to evaluate the performance impact of multiple streams. We first give an overall performance comparison of non-streamed code and streamed code. Then we analyze the performance gaps between them and the impacting factors.

6.1. *An overall performance comparison*

In this section, we will give an overall comparison between the streamed code (`w/`) and the non-streamed version (`w/o`). Note that the non-streamed version uses a single stream and a single tile/task. Meanwhile, the whole datasets of streamed code are partitioned into a large number of tasks, where the task granularity is controlled by the number of tiles. Then the tasks are mapped to streams and each stream runs multiple tasks. At the low level, the hardware resources (i.e., processing cores) are partitioned into group, and the number of processing cores per partition is referred to be as resource granularity. In the experiments, we empirically enumerate all the possible values of task granularity and resource granularity to obtain the optimal performance.

Figure 6 shows the overall performance comparison. For `MM`, `CF`, `Kmeans`, `NN`, `DCT`, `PS` and `DP`, we see that the streamed code outperforms the non-streamed code for all the used datasets, with an average performance improvement of 8.3%, 24.1%, 24.1%, 9.2%, 4.5%, 10% and 6.1%, respectively. Among the seven applications, six (`MM`, `CF`, `NN`, `DCT`, `PS`, `DP`) can overlap the data transfer stage and the kernel execution stage, i.e., they are overlappable. Although `Kmeans` is an non-overlappable application, it can benefit from the reduced memory allocation and deallocation during kernel execution by employing multiple streams.

Moreover, we see that using multiple streams brings no performance change for `Hotspot`. This is because data transfers and kernel execution of this application cannot be overlapped. Partitioning a large workload into several small workloads which are then mapped onto different resource partitions, gives no performance boost. Also, due to the overheads of managing streams, we notice that the streamed code runs slightly slower than the non-streamed code for the small datasets.

For `SRAD`, the streamed code runs slower than the non-streamed code for small datasets. The reason resembles that of the `Hotspot`. While the streamed version of `SRAD` outperforms the non-streamed one for large datasets. This case is out of our

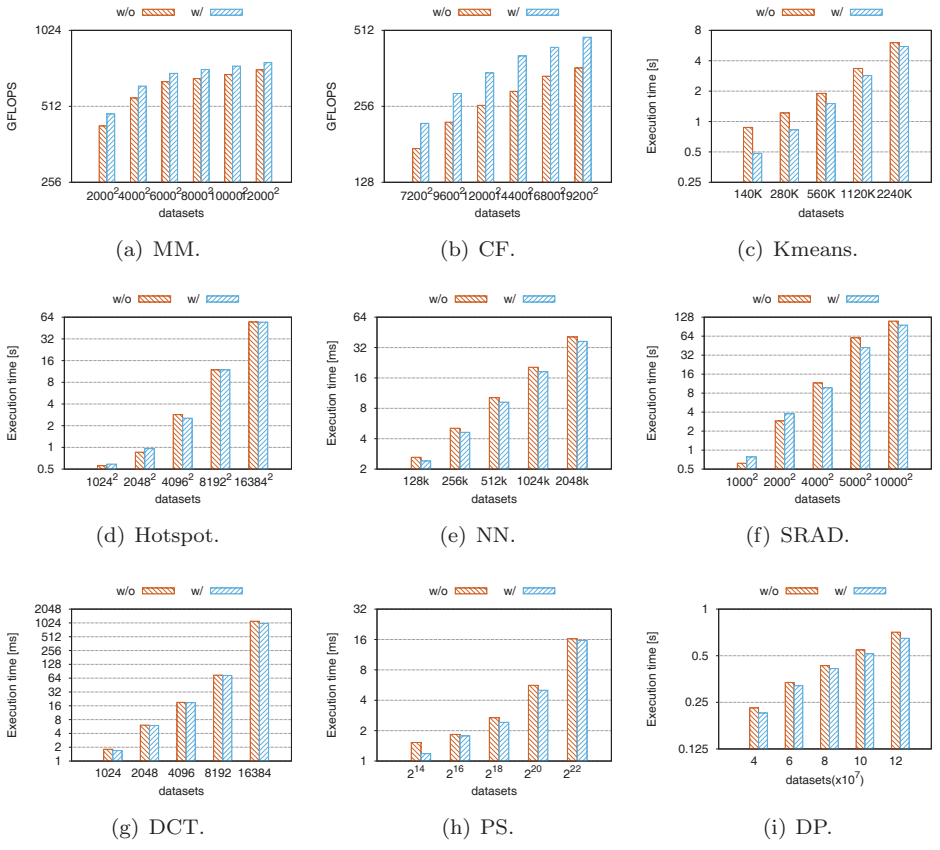


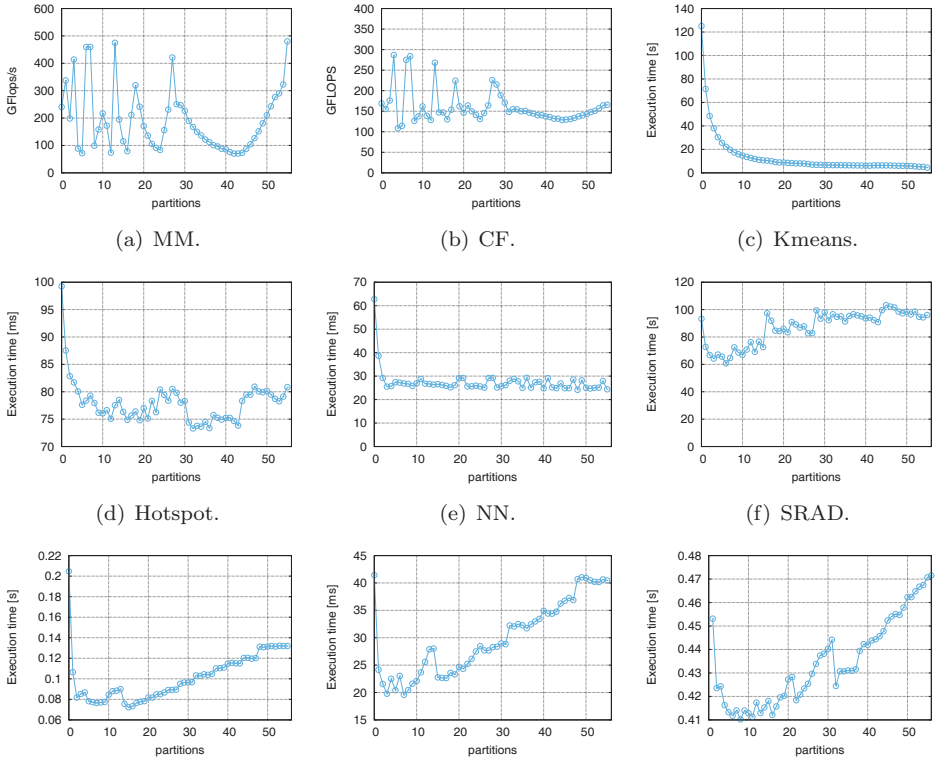
Fig. 6. A performance comparison between using a single stream and multiple streams. For Kmeans, the number of centroid is 8, and we run 100 iterations before reaching convergence. For Hotspot, we run 50 simulation iterations. For NN, the target coordination is (40, 120), and the number of nearest neighbors to find is 10. For SRAD, $\lambda = 0.5$, and we run the kernel for 100 iterations.

expectation. Theoretically, it should not occur due to its non-overlappable feature. The reason is still under investigation.

6.2. *Performance analysis*

6.2.1. *How the number of partitions impacts performance?*

Figure 7 shows how the overall performance changes with the number of partitions (P) when fixing task granularity (T). We observe that the performance varies significantly over P for the nine applications. For `MM` and `CF`, the benchmarks run much faster on some points than the others. On these points, 56 is a multiple of P, i.e., $P \in \{2, 4, 7, 8, 14, 28, 56\}$. Each 31SP Phi has 57 cores and one is reserved for the uOS. Thus, we have 224 (56×4) available threads. When mapping N streams onto a Phi, each stream will occupy $224/N$ threads. It is possible that two streams would



(a) MM. (b) CF. (c) Kmeans.

(d) Hotspot. (e) NN. (f) SRAD.

(g) DCT. (h) PS. (i) DP.

Fig. 7. How the performance changes with the number of partitions. For Matrix Multiplication, $D = 6000$, and $T = 500 \times 500$; For Cholesky Factorization, $D = 9600$, $T = 800 \times 800$, and we use the *column-major* layout. For Kmeans, $D = 1120000$, $T = 20000$, and we run 100 iterations before reaching convergence. For Hotspot, the grid size D is 16384×16384, $T = 1024 \times 1024$, and we run 50 iterations. For NN, the number of records D is 5242880, $T = 512$, and the number of nearest neighbors to find is 10. For SRAD, $D = 10000 \times 10000$, $T = 20 \times 20$, $\lambda = 0.5$, and we run the kernel for 100 iterations. For DCT, $D = 8192$ and $T = 16$. For PrefixSum, $D = 2^{22}$ and $T = 64$, for DotProduct, $D = 8 \times 10^7$ and $T = 64$.

share the same processing core and thus incur contention for shared resources such as caches. When the Phi is partitioned into groups, using these values within the set can avoid that the threads from the same core are partitioned into different streams. Therefore, we recommend using the number of partitions within the set $\{2, 4, 7, 8, 14, 28, 56\}$ for such applications.

For `Kmeans`, the execution time drops over the number of partitions (shown in Figure 7(c)). When looking into the code, we observe `Kmeans` has to allocate and free temporal memory space dynamically in each iteration. This overhead increases linearly with the number of threads and decreases with the number of streams accordingly. Thus, the performance trend of the non-overlappable `Kmeans` does not fit the one shown in Figure 5.

For `Hotspot`, we see that the execution time roughly matches the trend shown in Figure 5. However, there are some fluctuations when changing the number of partitions. Particularly, when the number of partitions ranges from 33 to 37, the simulation time reaches its lowest points. At this time, the number of threads per partition is 6 or 7, and each partition will use threads on at most two processing cores. We believe this configuration will lead to a good cache utilization.

Figure 7(e) shows how the number of partitions impacts the overall performance of `NN`. We see that the execution time first decreases sharply over partitions (and streams) til $P = 4$. This is due to the fact that using more streams will create more opportunities of overlapping data transfers and kernel execution. Thereafter, the execution time remains around 25 ms. This overlappable application can use both temporal sharing and spatial sharing.

Figure 7(f) shows how the execution time changes over partitions for `SRAD`. On the whole, we notice that the performance first increases and then decreases, which roughly fits the trend presented in Figure 5 of Section 5.2. This is because this application consists of several kernels between which an explicit synchronization is needed. Thus, the application can only exploit spatial sharing of multiple streams.

From Figures 7(g)–7(i), we see that execution time first decreases then increases over P, which fits the trend shown in Figure 5.2. For `DCT`, we reach the lowest point when $P = 16$. This is because $T = 16$ and some partitions will stay idle when using more than streams. The streamed `PrefixSum` reaches the optimal performance when $P = 4$. In such a case, both tasks and resources can be evenly partitioned, which leads to a good load balance. For `DotProduct`, the performance varies slightly when changing the number of resource partitions.

6.2.2. *How the number of tiles impacts performance?*

Figure 8 shows how the performance changes with the number of tiles (T) for the nine applications. Overall, the achieved performance first increases and then decreases (note the different metrics between `MM`, `CF` and the other four applications). In particular, we observe most applications run the fastest when $T = 4$. With one task, the performance decreases sharply. This is due to the fact that we partition the 56 cores of a Phi into four groups ($P = 4$), and mapping the tile to a partition will leave the other partitions idle. Further, using a larger T (i.e., more tiles but each tile is smaller) will have more pipelining opportunities to overlap stalls. But using a large T introduces extra control overheads and incurs a relatively low resource utilization. Therefore, further increasing the number of tasks leads to a worse performance as shown in Figure 8.

In addition, selecting T for `CF` and `SRAD` differs from the other applications. Their achieved performance reaches the optimal when $T = 100$, and $T = 400$, respectively. Different from other applications, these two contain several kernels which could introduce context interference. Furthermore, we see that `NN` obtains a similar performance between $T = 1$ and $T = 4$. This is because `NN`'s performance

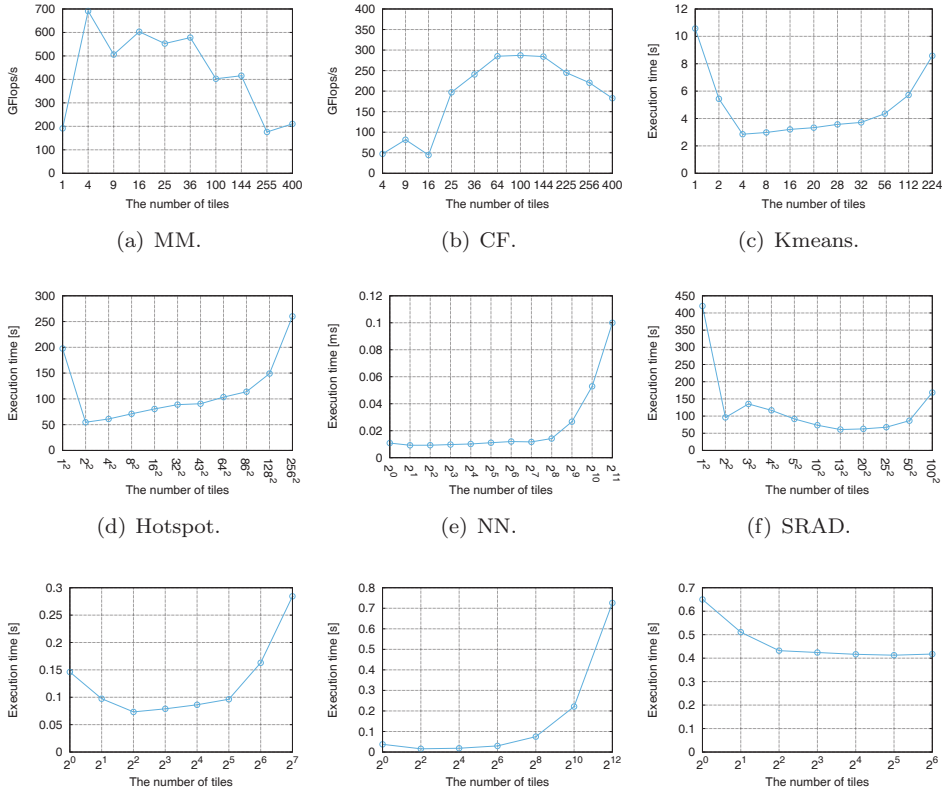


Fig. 8. How the performance changes with the number of tiles. For Matrix Multiplication, $D = 6000$, and $P = 4$; For Cholesky Factorization, $D = 9600$, $P = 4$, and we use the *column-major* layout. For Kmeans, $D = 1120000$, $P = 4$, the number of centroid is 8, and we run 100 iterations before reaching convergence. For Hotspot, the grid size D is 16384×16384, $P = 4$, and we run 50 iterations. For NN, the number of records D is 5242880, $P = 512$, the target coordination is (40, 120), and the number of nearest neighbors to find is 10. For SRAD, $D = 10000 \times 10000$, $P = 4$, $\lambda = 0.5$, and we run the kernel for 100 iterations. For DCT, $D = 8192$, $P = 4$. For PrefixSum, $D = 2^{22}$ and $P = 4$. For DotProduct, $D = 8 \times 10^7$ and $P = 4$.

is bounded by data transfers and creating multiple streams to achieve overlapping brings a slight difference to the overall performance. For `DP`, we notice that the execution time decreases with over T, but we expect an increase with more tiles.

6.3. *Discussion*

6.3.1. *Using occasion*

Our experimental results show that using multiple streams is beneficial only when the applications are *overlappable*. For such applications, exploiting temporal sharing of hardware resources will overlap data transfers and kernel execution, and thus

speedup the execution process. Further, using spatial sharing of hardware resources will increase the resource utilization. Note that only leveraging spatial sharing might not lead to a performance improvement for the non-overlappable applications.

Moreover, we observe several special cases that using multiple streams is beneficial for the non-overlappable applications (e.g., `Kmeans` and `SRAD`). This is because of the extra kernel overheads, e.g., allocating/deallocating temporal memory space. Therefore, we have to consider such application-specific characteristics when using multiple streams.

6.3.2. *Reducing the search space*

As can be seen from Section 6.2, resource granularity (P) and task granularity (T) have a significant impact on the overall performance. For a given application, maximizing the overall performance need search for the optimal value for each factor. This will consume a huge amount of time. Hereby we discuss how to prune the search space when selecting a proper value for P and T.

As indicated in Figures 7(a) and 7(b), we obtain that $P \in \{2, 4, 7, 8, 14, 28, 56\}$ and such values will avoid that the threads from the same core are mapped to different streams. The results of the other overlappable application (`NN`) show that when $P \geq 4$, the performance remains around 25 ms. Therefore, we should focus our attention on these special numbers.

When determining the number of tiles, the first priority is to guarantee load balancing. This is particularly true when $T < P$, i.e., the resource is under-utilized (Figure 8). Therefore, we guarantee that $T = m{\cdot}P$, where $m \in \{1, 2, 3, \ldots\}$. Besides, T should not be too large to achieve a good resource utilization, and it should not be too small to exploit the pipelining potentials.

To summarize, to achieve the optimal performance for will incur a huge search space. Our guidelines reduce the search space significantly. To further reduce the search space, we need a fine analytical performance model [10, 9, 11]. Alternatively, we plan to use machine learning techniques to obtain a proper value for P and T.

7. Preliminary results on multiple MICs

Current large-scale computing systems often employ multiple accelerators to guarantee its peak performance. Thus, how to use multiple devices simultaneously becomes an issue. Using multiple streams seems a promising tool. For example, `hStreams` provides a unified resource management layer of all the Phis (MICs) and its runtime automatically map the streams to the underlying hardware domain. In this way, a streamed code can run on multiple Phis without code modifications. In this section, we discuss the preliminary results on the heterogeneous platforms with multiple Phis.

Figure 9 shows the `CF`'s performance on one and two Phis. We see that the achieved performance increases significantly with two devices than with one device, but the performance is still lower than the projected performance of two Phis. This

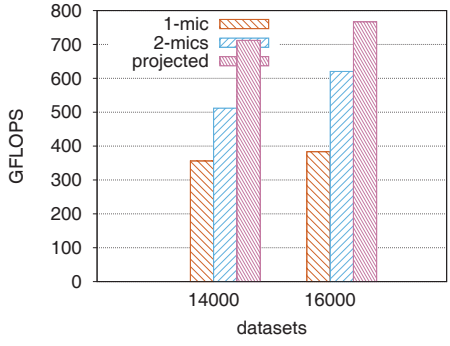


Fig. 9. How Cholesky Factorization performs on multiple MICs for datasets 14000 × 14000 and 16000 × 16000.

is because partitioning workloads among devices with separate memory space needs to transfer more data blocks than that using only one device. Also, `CF` contains several kernels and explicit synchronizations are required between them. When using multiple Phis, synchronizations between streams from different Phis might introduce extra overheads. To gain more insights, we would like to run more experiments with a wide range of applications in future.

8. Conclusion

The potential of using multiple streams on the heterogeneous platforms is expected to be significant for a wide range of applications. In this paper, we perform a systematic performance evaluation of multiple streams on the MIC-based heterogeneous platforms. Our experimental results at the microbenchmarking level and the real-world application level lead to the following observations/conclusions: (1) The data transfers in both directions on Phi cannot run concurrently; (2) Data transferring on Phi overlaps kernel execution, but the full overlap seems not achievable; (3) Using multiple streams might not lead to a performance increase only in the presence of spatial resource sharing; (4) Being overlappable is a must for benefits when using multiple streams; (5) Both task granularity and resource granularity have a large impact on the overall performance; (6) Some non-overlappable application still enjoy a performance improvement by using multiple streams.

In the future, we would like to investigate how to transform the non-overlappable applications to overlappable applications. Further, we will leverage machine learning techniques to obtain a proper task and resource granularity. Also, we plan to further evaluate the performance impact on multiple Phis.

Acknowledgment

We would like to thank the authors from the Rodinia benchmark suite for their valuable benchmarks. We are also thankful to the reviewers for their constructive

comments. This work was partially funded by the National Natural Science Foundation of China under Grant Nos. 61402488, 61502514 and 61602501, the National Key Research and Development Program of China under Grant No. 2016YFB0200400.

References

[1] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, pp. 879–899, May 2008.

[2] M. R. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden, and S. Poole, "Modeling and predicting performance of high performance computing applications on hardware accelerators," *International Journal of High Performance Computing Applications*, vol. 27, pp. 89–108, May 2013.

[3] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate CPU vs. GPU performance without the answer," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pp. 134–144, IEEE, Apr. 2011.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, IEEE, Oct. 2009.

[5] M. Boyer, J. Meng, and K. Kumaran, "Improving GPU performance prediction with data transfer modeling," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pp. 1097–1106, IEEE, May 2013.

[6] NVIDIA Inc., *CUDA C Best Practices Guide Version 7.0*, March 2015.

[7] The Khronos OpenCL Working Group, "OpenCL – The open standard for parallel programming of heterogeneous systems." http://www.khronos.org/opencl/, January 2016.

[8] Intel Inc., *hStreams Architecture document for Intel MPSS 3.5*, April 2015.

[9] B. Van Werkhoven, J. Maassen, F. J. Seinstra, and H. E. Bal, "Performance models for CPU-GPU data transfers," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pp. 11–20, IEEE, May 2014.

[10] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, "Performance models for asynchronous data transfers on consumer graphics processing units," *Journal of Parallel and Distributed Computing*, vol. 72, pp. 1117–1126, Sept. 2012.

[11] B. Liu, W. Qiu, L. Jiang, and Z. Gong, "Software pipelining for graphic processing unit acceleration: Partition, scheduling and granularity," *International Journal of High Performance Computing Applications*, DOI: 10.1177/1094342015585845, June 2015.

[12] F. Ino, S. Nakagawa, and K. Hagihara, "Gpu-chariot: A programming framework for stream applications running on multi-gpu systems," *IEICE Transactions*, vol. 96-D, no. 12, pp. 2604–2616, 2013.

[13] C. J. Newburn, G. Bansal, M. Wood, L. Crivelli, J. Planas, A. Duran, P. Souza, L. Borges, P. Luszczek, S. Tomov, J. Dongarra, H. Anzt, M. Gates, A. Haidar, Y. Jia, K. Kabir, I. Yamazaki, and J. Labarta, "Heterogeneous streaming," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*, pp. 611–620, 2016.

[14] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, Sept. 2006.

[15] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for GPGPU

spatial multitasking," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pp. 1–12, IEEE, Feb. 2012.

[16] F. Wende, T. Steinke, and F. Cordes, "Multi-threaded kernel offloading to GPGPU using Hyper-Q on kepler architecture," Tech. Rep. 14-19, ZIB, Takustr.7, 14195 Berlin, 2014.

[17] F. Wende, T. Steinke, and F. Cordes, "Concurrent kernel execution on xeon phi within parallel heterogeneous workloads," in *Euro-Par 2014 Parallel Processing* (F. Silva, I. Dutra, and V. Santos Costa, eds.), vol. 8632 of *Lecture Notes in Computer Science*, pp. 788–799, Springer International Publishing, 2014.

[18] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu, "Adaptive optimization for petascale heterogeneous CPU/GPU computing," in *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pp. 19–28, IEEE, 2010.

[19] C. Yang, W. Xue, H. Fu, L. Gan, L. Li, Y. Xu, Y. Lu, J. Sun, G. Yang, and W. Zheng, "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 '13, (New York, NY, USA), pp. 1–12, ACM, 2013.

[20] H. Takizawa, K. Sato, and H. Kobayashi, "SPRAT: Runtime processor selection for energy-aware computing," in *Cluster Computing, 2008 IEEE International Conference on*, pp. 386–393, IEEE, 2008.

[21] J. A. Pienaar, A. Raghunathan, and S. Chakradhar, "MDR: Performance model driven runtime for heterogeneous parallel platforms," in *Proceedings of the International Conference on Supercomputing*, ICS '11, (New York, NY, USA), pp. 225–234, ACM, 2011.

[22] S. Mittal and J. S. Vetter, "A survey of CPU-GPU heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, July 2015.

[23] K. Spafford, J. Meredith, and J. Vetter, "Maestro: Data orchestration and tuning for OpenCL devices," in *Euro-Par 2010 – Parallel Processing* (P. D'Ambra, M. Guarracino, and D. Talia, eds.), vol. 6272 of *Lecture Notes in Computer Science*, pp. 275–286, Springer Berlin Heidelberg, 2010.

[24] J. Planas, R. M. Badia, E. Ayguade, and J. Labarta, "AMA: Asynchronous management of accelerators for task-based programming models," *Procedia Computer Science*, vol. 51, pp. 130–139, 2015.