HPerf: A Lightweight Profiler for Task Distribution on CPU+GPU Platforms

Joo Hwan Lee Nimit Nigania Hyesoon Kim
School of Computer Science
Georgia Institute of Technology
{*joohwan.lee, nnigania3, hyesoon*}*@gatech.edu*

Bevin Brett
Software and Services Group
Intel Corporation
bevin.brett@intel.com

***Abstract*—Heterogeneous computing has emerged as one of the major computing platforms in many domains. Although there have been several proposals to aid programming for heterogeneous computing platforms, optimizing applications on heterogeneous computing platforms is not an easy task. Identifying which parallel regions (or tasks) should run on GPUs or CPUs is one of the critical decisions to improve performance.**

In this paper, we propose a profiler, HPerf, to identify an efficient task distribution on CPUs+GPUs system with low profiling overhead. HPerf is a hierarchical profiler. First it performs lightweight profiling and then if necessary, it performs detailed profiling to measure caching and data transfer cost. Compared to a brute-force approach, HPerf reduces the profiling overhead significantly and compared to a naive decision, HPerf improves the performance of OpenCL applications up to 25%.

I. Introduction

These days, many computing systems from mobile platforms to server systems have both CPUs and GPUs. In the last few years, GPUs have evolved to execute not only graphics applications but also general-purpose computing workloads [1]. Even mobile platforms such as the one from Qualcomm [2] utilize GPUs for general-purpose computation. Programming for heterogeneous computing that uses both CPUs and GPUs is everywhere.

OpenCL aims to increase programmability and portability on heterogeneous computing [3]. Lately, several other programming environments such as OpenACC [4], Rigel [5], and COMIC [6] have been proposed for the same purpose. Nonetheless, programming on heterogeneous computing still requires a significant amount of programmers' efforts to achieve the best performance. Identifying which task should run on CPUs or GPUs is one of the critical decisions, which is not trivial.

A common practice of task distribution on CPUs and GPUs considers two factors: "*thread-level parallelism*" (TLP) and "*the amount of data transfer between CPUs and GPUs*" [7], [8], [9], [10]. However, these two factors might not be sufficient. Also, identifying the amount of data to be transferred between CPUs and GPUs might not be obvious. Furthermore, caching behavior might also change depending on task distribution (or offloading scenario).

Nonetheless, no previous work has considered all these effects that change depending on offloading scenario. In this paper, we propose a new profiler, *HPerf*, to consider computing power differences, data transfer time cost and caching effects together. The input to HPerf is parallelized code running on CPUs and the outcome is a suggested offloading scenario to achieve the best estimated performance.

The rest of the paper is organized as follows. First, we explain the limitation of previous approaches and then summarize our contributions in Section II. We propose our profiling algorithms in Section III and describe the implementation of the profiler in Section IV. Section V presents our evaluation results and Related works are presented in Section VI. Finally, Section VII concludes the paper.

II. Background and Motivation

A. Previous Approaches on Task Distribution on CPUs and GPUs

A candidate application for heterogeneous platform with CPUs and GPUs consists of two kinds of code regions; "*parallel region*" and "*sequential region*". Parallel regions are the code regions that can be parallelized, and sequential regions are the ones that cannot be parallelized or have very limited parallelism.

While where to execute the sequential region is straightforward (on CPUs), it is nontrivial where to execute the parallel region. There are three types of previous static-time task distribution decision mechanisms. [1]

1) *All-CPU-or-All-GPU*
   The simplest decision mechanism is to execute all tasks either on CPUs or GPUs. Programmer can measure total execution time by running all tasks on CPUs or on GPUs and then compare both and choose one. [11]
2) *Measure-Separately*
   Instead of running all tasks on CPUs or GPUs, the Measure-Separately approach measures execution time and data transfer time for each task separately, *one at a time* [12], [13], [14]. Although this approach

[1] Run-time systems also have many different scheduling algorithms to distribute tasks. Our work is task distribution at static-time to take advantage of compilation time optimizations. Run-time scheduling algorithm can improve performance further with the static-time optimized code.

sounds simple and reasonable, the execution time of each task is affected by other task execution decisions because of different data movement costs and different caching behaviors. We explain this in more detail in Section II-B.

3) *Brute-Force*
The Brute-Force approach is to profile all the possible scenarios. Obviously, this method will find the best-performing scenario, but this method is not scalable with multiple tasks. Hence, we need an efficient profiling algorithm that does not require all scenarios to be profiled but still identifies the best performing scenario.

Before we propose our new algorithms, we explain the limitation of the *Measure-Separately* approach in more detail.

B. Limitations of Measure-Separately

The Measure-Separately approach has three limitations. First, this approach assumes that GPU code is available so that the profiler can measure the execution time on GPUs. Second, the data transfer cost for each parallel region is assumed to be the same regardless of offloading decisions (i.e., task distribution decision) for other parallel regions. Third, the execution time for each parallel region is assumed to remain the same regardless of offloading decisions.

1) Existence of GPU code: Obviously, GPU code is not always available. Even though the same OpenCL code can run on CPUs and GPUs, programmers generally need to optimize the code for different architectures.

2) Data Transfer Cost: Figure 1 shows two different offloading scenarios. Parallel regions A, B, C, and D are sequentially executed. Arrows indicate possible data movement between parallel regions. Data movement between parallel regions can occur either through (1) *a CPU-GPU communication interface such as a PCI-E bus (solid line)* or (2) *within the same computing platform (dotted line)*. Depending on offloading decisions, data movement can be done differently. In scenario (a), arrows ❶, ❸, ❺, ❻, and ❽ are CPU-GPU communications. But, in scenario (b), arrows ❷, ❸, ❹, ❼, and ❽ require data transfer between the CPU and the GPU.

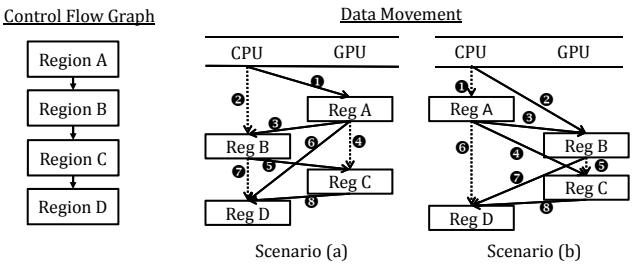

Figure 1: Different offloading scenarios and data movement.

For example, the data movement from parallel region B to parallel region C (❺) requires a CPU-GPU communication in scenario (a), but it becomes internal communication in scenario (b). The Measure-Separately approach decides whether to offload parallel region B without considering this implication. Parallel region A could generate results that parallel regions C and D use. Depending on whether or not parallel region A is executed on a GPU, the data movement method for ❹ and ❻ can be a CPU-GPU communication or just an internal communication. Considering all the communication overhead with regard to other parallel regions is not possible in the Measure-Separately approach.

3) Cache-to-Cache Influx: The data movement within the same computing platform is often done through caches. We call this data movement through the cache "*cache-to-cache influx*" in this paper.

When a parallel region is offloaded to a GPU, the performance of other parallel regions is also changed. A parallel region's offloading decision will affect other parallel region's cache behaviors. The performance of parallel regions on CPUs can be *improved* compared to the base scenario in which all parallel regions are running on CPUs, if offloading reduces the working set size of parallel regions on CPUs. The performance can be *degraded* because data has to be copied from a GPU. This data could have been inside the CPU cache if the offloaded parallel region ran on CPUs.

Tables I and II show cache behavior changes for the two scenarios in Figure 1. [2] The performance column shows the performance of each scenario over the base scenario where all parallel regions run on CPUs. The performance can either be improved or degraded. Interesting scenarios (shaded in the tables) are when the performance improves because of reduced effective working set size on CPUs by offloading some regions on GPUs. Please note that we consider cache-to-cache influx only for CPU caches because the performance on CPUs is heavily affected by caching behavior but not on GPUs.

C. Contributions

As Section II-B shows, the performance of each task can be affected by other task distribution decisions. Nonetheless, no previous decision mechanism estimates these costs efficiently. Even though Brute-Force approach can find the best performing scenario, it has scalability problem with multiple parallel regions. Measure-Separately can be used to solve the scalability problem of Brute-Force, but it misses many important aspects that Brute-Force can capture.

Hence, in this paper, we propose *HPerf* to estimate cache behavior changes and data movement costs depending on offloading scenarios. HPerf is a lightweight profiler to help programmers by achieving two contradicting goals:

[2] The table does not discuss cache misses due to conflict/capacity misses by its own parallel region.

Table I: Region D's cache behavior for Scenario (a).

Data from	Baseline (ALL-CPU)	Scenario (a)	Performance
Region A (❻)	Hit if B & C don't evict data from A	Miss	Degrade
	Miss if B & C evict data from A	Miss	Same
Region B (❼)	Hit if C doesn't evict data from B	Hit	Same
	Miss if C evicts data from B	**Hit**	**Improve**
Region C (❽)	Hit	Miss	Degrade

Table II: Region D's cache behavior for Scenario (b).

Data from	Baseline (ALL-CPU)	Scenario (b)	Performance
Region A (❻)	Hit if B & C don't evict data from A	Hit	Same
	Miss if B & C evict data from A	**Hit**	**Improve**
Region B (❼)	Hit if C doesn't evict data from B	Miss	Degrade
	Miss if C evicts data from B	Miss	Same
Region C (❽)	Hit	Miss	Degrade

(1) *finding a well-performing offloading scenario* with (2) *reasonable profiling overhead*.

III. Profiling Algorithms

In this section, we explain our basic profiling algorithms. Detailed discussions of the mechanisms are explained in later sections.

A. Overview of Profiling Stages

HPerf is a hirarchical profiler consisting of five profiling stages as follows.

- *Stage-1. Check to see if All-CPU or All-GPU is good enough.*
  First, we do one-time profiling similar to the Measure-Separately approach. This stage decides if further profiling stages are required in an approximated manner. We measure the execution time of each parallel region on CPUs. Here, we ignore cache-to-cache influx. The execution time on GPUs is approximated using the heuristics in Section IV-B. We estimate the maximum data transfer time of ALL-GPU execution with the maximum data transfer overhead. If GPU execution time is significantly lower than CPU execution time even with data transfer cost then HPerf chooses All-GPU at this stage and finishes profiling.

- *Stage-2. Check to see if cache-to-cache influx is important.*
  To determine whether the cache-to-cache influx can significantly change the performance, HPerf measures the cache performance assuming that *every parallel region starts from a cold state*. If cache-hit ratio with cold-start is high enough, the profiler bypasses Stage-4.

- *Stage-3. Estimate data transfer costs for all offloading scenarios.*
  In this stage, HPerf estimates more precise data transfer costs for all possible offloading scenarios and therefore total execution time of them. HPerf identifies which data need to be transferred between CPUs and GPUs without requiring detailed simulation or profiling. The detailed algorithm is presented in Section III-B.

- *Stage-4. Capture cache-to-cache influx effect.*
  To capture performance changes due to cache-to-cache influx, HPerf measures execution time on CPUs for selected offloading scenarios. The performance change of each parallel region on CPU depending on offloading scenarios are applied on estimated total execution time of offloading scenarios from Stage-3. We propose a new algorithm (Section III-C) to reduce the number of offloading scenarios that need to be profiled. The algorithm considers multiple workload characteristics such as the memory footprint size for each parallel region and the data sharing patterns between parallel regions. HPerf identifies the important offloading scenarios that require detailed profiling.

- *Stage-5. Suggest the final decision.*
  Finally, HPerf selects the offloading scenario that is estimated to have the best performance. At this stage, data transfer time and execution time on CPUs and GPUs are updated based on the previous stages' outcomes.

Now we highlight our proposed profiling algorithms: *DataProfiler* and *InfluxProfiler*. To improve readability, the implementation details are explained in later sections.

B. DataProfiler: Data Transfer Overhead Estimation

DataProfiler estimates the data transfer time overhead for each offloading scenario by computing (1) *the number of API calls for data transfer between CPUs and GPUs* and (2) *the amount of data for each transfer*. In the ALL-CPU scenario

ALL-CPU	CPU-GPU
C_1. $O_\alpha \leftarrow$ Reg A_{CPU} (I_α)	C_1. $O_\alpha \leftarrow$ Reg A_{CPU} (I_α)
	D_1. Write (I_β) : CPU → GPU
C_2. $O_\beta \leftarrow$ Reg B_{CPU} (I_β)	C_2. $O_\beta \leftarrow$ Reg B_{GPU} (I_β)
	D_2. Read(O_β) : CPU ← GPU

Figure 2: A code example comparing the ALL-CPU scenario and a CPU-GPU scenario.

of Figure 2, data transfer is not needed, but in the CPU-GPU scenario, the number of API calls for data transfer is *two*; *one* for sending I_β from a CPU to a GPU (D_1) and *one* for sending O_β from a GPU to a CPU (D_2). DataProfiler also estimates the amount of data for each transfer, which is the size of I_β and O_β in this example.

The key to accurately estimate the number of data transfers and the amount of data transferred for each data transfer is to identify the minimum data that must be transferred between CPUs and GPUs. Only the data not currently present in the computing platform needs to be transferred. In order to know which data should be transferred, DataProfiler constructs a state-machine to keep track of the location (on the CPU or the GPU) and its dirty state (clean, dirty, or invalid) for each memory object. DataProfiler computes the state transitions of each memory object for all possible offloading scenarios. Data should be moved from one location to another (CPUs or GPUs) only if the latest data is in the other location. [3] The detailed mechanism to compute the data transfer time is presented in Section IV-E.

C. InfluxProfiler: Cache-to-Cache Influx Profiling

To measure performance change due to cache-to-cache influx, HPerf performs profiling for important offloading scenarios. To reduce the number of scenarios to be profiled, InfluxProfiler identifies potentially cache-to-cache influx existing scenarios considering following factors.

1) Shared data group between parallel regions
2) Memory footprint size for each parallel region
3) Memory-write within each parallel region
4) Data movement from sequential regions to parallel regions

In this section, we only discuss how considering data sharing between parallel regions and memory footprint size can reduce the number of scenarios to be profiled. The detailed mechanism is discussed in Section IV-F.

Terminology. In this paper, we use parallel regions running on CPUs to indicate different offloading scenarios. For example, $\{A, B\}$ represents running parallel regions A and then B on CPUs and running other parallel regions on GPUs.

Data Sharing and Memory Footprint Size. InfluxProfiler utilizes data sharing between parallel regions and the memory footprint size of each parallel region in order to reduce the number of scenarios to be profiled. In Figure 3, there are four parallel regions in an application, and all parallel regions can be potentially offloaded. All four parallel regions A, B, C, and D are sequentially executed.

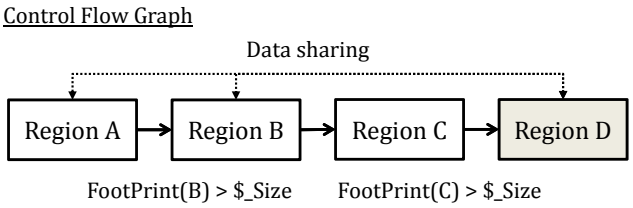

Figure 3: Considering data sharing between parallel regions and memory footprint size.

Suppose we want to find the offloading scenario that might increase the cache-hit ratio on parallel region D running on CPUs. The Brute-Force approach will suggest profiling the scenarios of executing parallel regions $\{A, D\}$, $\{B, D\}$, $\{C, D\}$, $\{A, B, D\}$, $\{A, C, D\}$, $\{B, C, D\}$, and $\{A, B, C, D\}$ on CPUs. However, if we know parallel region D shares data only with parallel regions A and B, executing parallel region C on CPUs will not increase cache hits on parallel region D. Please note that the output of InfluxProfiler is the list of potentially cache-to-cache influx existing scenarios. Hence, the scenario of executing parallel regions $\{C, D\}$ on CPUs need not be profiled.

After that, we check the memory footprint size of parallel region B. If parallel region B's memory footprint size is greater than the cache size, any cache block that is inserted before executing parallel region B will be evicted by parallel region B. [4] This means $\{A, B, D\}$ and $\{B, D\}$ show the same cache-to-cache influx on parallel region D. This holds also true for $\{B, C, D\}$ and $\{A, B, C, D\}$. The sets can be reduced further if we consider the memory footprint size of parallel region C. Hence, the final sets to profile will be only executing parallel regions $\{A, D\}$, and $\{B, D\}$ on CPUs.

Instead of profiling seven scenarios, we now have to profile only two scenarios. We perform a similar task for each parallel region and accumulate the scenarios for each region to get the total number of unique scenarios to be profiled. Formal definitions of the findings are as follows.

- *Finding 1 (Data Sharing). When the parallel region of interest is R_d, for any parallel region R that has no data sharing with R_d, there exists no cache-to-cache influx from region R to region R_d in any offloading*

[3] The FSM of clean, dirty and invalid is the same as FSM in MSI cache coherence protocol.

[4] We assume LRU cache replacement policy.

scenario.

- *Finding 2 (Memory Footprint Size). When the parallel region of interest is R_d, for any parallel region R whose memory footprint size is greater than cache size, there exists no cache-to-cache influx from the parallel region R_p (preceding R and then R_d) to region R_d in any offloading scenario.*

IV. Implementation

Now, we describe our profiler, *HPerf*, which implements profiling algorithms in the previous section.

A. Overview

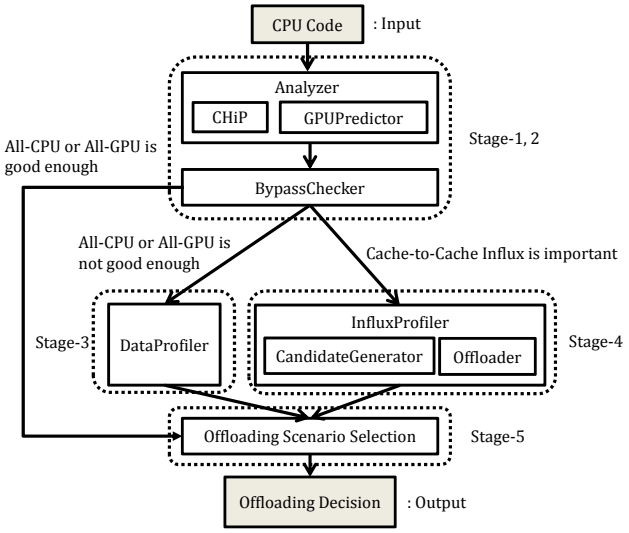

Figure 4: The flow of the HPerf.

Figure 4 shows the flow of the profiler. First, *Analyzer* measures execution time and captures the memory behavior of the application. Then, *BypassChecker* checks whether or not later profiling stages are required. This corresponds to *Stage-1* and *Stage-2*. *DataProfiler* (*Stage-3*) computes the data transfer overhead and therefore total execution time of all possible offloading scenarios. In Stage-3, we ignore performance changes of parallel regions on CPUs due to cache-to-cache influx. The performance change is captured by *InfluxProfiler* (*Stage-4*) by profiling reasonable number of offloading scenarios. Different execution time of parallel regions on CPUs depending on offloading scenarios are applied on estimated total execution time of offloading scenarios from DataProfiler. Finally, the estimated optimal offloading scenario is suggested at *Stage-5*.

B. Analyzer(Stage 1)

Analyzer collects application characteristics through profiled execution. It is implemented as a runtime library based on OpenCL to automate benchmark analysis. The current implementation version targets OpenCL applications, but the implementation can be easily extended to other programming platforms such as x86 + CUDA, and the same proposed algorithms should apply. [5] Analyzer in Stage 1 extracts following application characteristics.

1) *The execution time of each parallel region on the CPU*
2) *The estimated performance of each parallel region on the GPU*
3) *The data movement between sequential and parallel regions in the application*
4) *The memory access pattern in each parallel region*
5) *Whether each parallel region's memory access patterns change on different iterations of loops*

The extracted information is used in later profiling stages. To extract these information, Analyzer gathers OpenCL API call traces.

1) Tracing OpenCL APIs: Analyzer traces data movement OpenCL API calls such as `clEnqueueWriteBuffer`, `clEnqueueReadBuffer` to extract the data movement between sequential and parallel regions. The range ($[start\_addr : end\_addr]$) of each OpenCL buffer involved by data movement APIs is extracted by Analyzer tracking arguments passed to those APIs.

Analyzer also calculates the memory footprint size for each parallel region by capturing OpenCL buffer objects accessed by each parallel region. [6] Analyzer gathers call traces of API calls for OpenCL buffer creation (`clCreateBuffer`) to extract the number of memory objects and size of each memory object. The list of OpenCL buffer objects accessed by each parallel region is extracted by gathering the trace of argument setting API function calls (`clSetKernelArg`).

2) GPU Performance Prediction: GPUPredictor estimates the computation time of the parallel region on GPUs using the CPU version of code. Compared to previous GPU analytical models [15], [16] requiring actual GPU code, we use a simple approximation to get the first-order approximation of GPU code performance. It should be noted that even when the application is written in OpenCL, programmers generally need to optimize the code for different architectures, which becomes the burden for programmers. In this paper, we consider only the machines' computing power differences to estimate GPU code performance.

[5] Currently, Analyzer and Offloader are implemented as OpenCL API, but those can be changed in different styles such as annotation-based approach.

[6] HPerf considers GPU's memory capacity and offloading scenarios that requires more memory space than the capacity is eliminated from possible offloading scenarios. In this paper, we assume all offloading scenarios do not exceed GPUs total memory capacity for brevity.

$$Machine\_Scale = \frac{Peak\_GPU\_FLOPS}{Peak\_CPU\_FLOPS} \quad (1)$$

$$Machine\_Scale\_SIMD = \begin{cases} \frac{Machine\_Scale}{SIMD\_width} & vectorized \\ Machine\_Scale & otherwise \end{cases} \quad (2)$$

$$GPU\_time = \frac{CPU\_time}{Machine\_Scale\_SIMD} \quad (3)$$

Equation (3) presents how HPerf predicts GPU computation time. We utilize peak performance differences between CPUs and GPUs and take into account whether the CPU code is vectorized, as shown in Equations (1), (2).

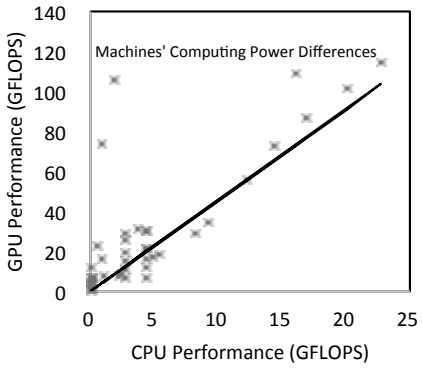

Figure 5: Scatter plot of CPU performance vs. GPU performance.

Figure 5 shows a scatter plot of CPU performance on the X-axis and GPU performance on the Y-axis for the same benchmark running on CPUs and GPUs. Benchmarks include several OpenCL applications with different degrees of parallelism and memory requirements. Although there are some outliers, many benchmarks approximate the machines' computing power differences.

An appliction can either be GPU-friendly or not. It can be GPU-friendly if the memory latencies are hidden when offloaded to the GPU. On the other hand, an application can be CPU-friendly if the application has limited thread-level parallelism or is limited by memory bandwidth on GPUs while it benefits from high instruction-level parallelism on the CPU. However, since the profiler only needs a first-order approximated execution time, our approximation is effective, as the figure shows. Future work will detect the outlier cases and improve GPU performance prediction.

C. BypassChecker

BypassChecker classifies applications and decides whether or not further profiling is required. In Stage 1, *BypassChecker* compares the data transfer cost between the CPU and the GPU and the benefit on computation time by offloading. Analyzer provides the measured computation time for each parallel region on a CPU and the list of memory objects for each parallel region and their sizes. The predicted computation time of each parallel region on a GPU is also given as input.

$CompTime\_CPU$ and $CompTime\_GPU$ are the sum of all parallel regions' computation time on CPUs and GPUs and $TransferTime\_GPU$ is the data transfer time when running all parallel regions on GPUs. We compare the following values.

1) $CompTime\_CPU$
2) $CompTime\_GPU$ + $TransferTime\_GPU$

If (2) is smaller than (1) by more than $five$ times, HPerf selects All-GPU execution as the offloading decision.[7] If (2) is larger than (1) by more than $five$ times, then HPerf selects All-CPU execution. When (1) and (2) do not belong to both cases, HPerf utilizes DataProfiler.

$TransferTime\_GPU$ is a first-order data transfer overhead estimation that is the maximum for all possible offloading scenarios. To calculate the maximum data transfer overhead, we assume "*all memory objects (i.e. OpenCL buffer objects) used in each parallel region are transferred from a CPU to a GPU before executing the parallel region and are transferred back from a GPU to a CPU after executing the parallel region regardless of how each buffer is accessed by each parallel region*". The number of API calls for data transfer between CPUs and GPUs and the amount of data for each transfer are maximal with this assumption.

D. Stage 2

In Stage 2, we check whether the cache-hit ratio for CPU computation will significantly change depending on offloading decision because of the change of cache-to-cache influx. If we predict that caching behavior will not be affected, InfluxProfiler profiling can be skipped. To get the first-order approximation, Analyzer profiles the cache-hit ratio of parallel regions when cache status is assumed to cold-start for each parallel region.

Cache-to-cache influx helps to reduce the number of cold cache misses. The cache-hit ratio for any offloading scenario must be better than the cache-hit ratio with cold-start. So, if the cache still shows a high cache-hit ratio even when all memory accesses have cold misses, the cache performance will not be affected much by offloading decisions, and the performance change on CPUs will be small. BypassChecker decides not to execute InfluxProfiler if the cache-hit ratio with cold-start is already high (*75.0%* as threshold).

1) Profiling Memory Access Pattern of Parallel Regions: To extract the memory access pattern of each parallel region during profiled execution, Analyzer employs a fast cache profiler called *CHiP* [17] based on Pin [18]. Analyzer resets

[7] $five$ is emprically modeled.

the cache in CHiP at the beginning of each parallel region to collect the cache-hit ratio with cold-start.

CHiP also collects memory address signatures while it collects cache hit information. The signature of memory accesses generated by CHiP represents the memory access pattern of each parallel region that is used by InfluxProfiler to identify data sharing between parallel regions.

$$hashed\_addr = addr[25:10] \oplus addr[41:26] \quad (4)$$

$$signature[hashed\_addr] = 1 \quad (5)$$

Equations (4) and (5) are used to generate signatures similar to the one in [19]. We omit the LSB 10 bits to consider only the page address and omit the MSB 22 bits since most applications do not use more than 40 bits in the address space.

Analyzer differentiates the access type of memory references of each parallel region. It differentiates `read-only`, `write-only`, and `read-write` accesses for each buffer object. Analyzer performs static analysis on parallel regions to extract this information.

E. DataProfiler

DataProfiler computes the data transfer overhead for different offloading scenarios by emulation. DataProfiler utilizes extracted workload information from Analyzer. Analyzer provides call traces of OpenCL API calls so that DataProfiler constructs a state-machine for each memory object. The state-machine inspired from cache coherence protocol is used to compute the minimum data that must be transferred between CPU and GPU. System memory on a CPU and device memory on a GPU are considered as each one's private cache. CPU's system memory is also considered as backing store in cache coherence protocol.

Call traces of OpenCL API calls are used as primitives for state transition in the state-machine. DataProfiler converts OpenCL API calls for (1) *data movement between the host and OpenCL compute devices* and (2) *kernel launch on OpenCL compute devices* as primitives. DataProfiler uses the following primitives that change the state of each memory object. By emulating the change of state for each memory object in different offloading scenarios, DataProfiler identifies the number of data of data transfers and amount of each data transfer.

- *W_SEQ: Data movement from a sequential region*
- *R_SEQ: Data movement to a sequential region*
- *(W/R/RW)_CPU: Write-Only/Read-Only/Read-Write access from a parallel region executed on CPUs*
- *(W/R/RW)_GPU: Write-Only/Read-Only/Read-Write access from a parallel region executed on GPUs*

$$TransferTime(sec) = \alpha \times TransferAmount(KB) + \beta \quad (6)$$

To convert the number of data transfers and amount of each data transfer into time, we use a linear model. Equation (6) represents the model we use to calculate the data transfer time of a single data transfer. The data transfer time of each data transfer is determined with following two values.

1) *The bandwidth of the data transfer* and
2) *The initialization cost of each transfer*.

The coefficients of Equation (6) depend on hardware configuration and are empirically determined. We run several microbenchmarks that have different data transfer sizes to get the coefficient variable values. So, the coefficient is α = 3.0 $\times$ 10^{-7}, β = 2.0 $\times$ 10^{-4} for a data transfer from a CPU to a GPU and α = 6.0 $\times$ 10^{-7}, β = 4.0 $\times$ 10^{-4} for a data transfer from a GPU to a CPU in our evaluation.

F. InfluxProfiler

1) Data Sharing and Memory Footprint Size: InfluxProfiler uses data sharing between parallel regions and the memory footprint size of each parallel region to reduce the number of scenarios to be profiled. In order to know whether different parallel regions share data, InfluxProfiler computes signature differences similar to the Hamming distance as in Equation (7). The signature of each parallel regions is generated by Analyzer.

$$sig\_diff = \frac{|(sig_i \cup sig_j) - (sig_i \cap sig_j)|}{|(sig_i \cup sig_j)|} \quad (7)$$

When sig_diff is 0, it means that parallel regions share data, while sig_diff being 1 means no sharing. If sig_diff is less than a threshold (*0.5*), InfluxProfiler concludes that parallel regions i and j share data.

2) Data Movement from Sequential Region: Data movement from sequential regions can also affect the CPU's cache behavior. For example, even when two consecutive parallel regions access identical data, the sequential region between these parallel regions can overwrite cached data from the previous region using say `clEnqueueWriteBuffer` in the case of OpenCL. Since the data is flushed from the cache, there is no reuse in the later parallel region. InfluxProfiler decides that there is no cache-to-cache influx for such scenario and uses it to reduce the number of scenarios to be profiled.

3) Memory-Write within Parallel Region: In many heterogeneous applications, a common pattern found is that a memory object is shared between multiple parallel regions, but the access types from different parallel regions are different. One common usage is using a memory object as an index array for other memory objects. The indexes are calculated and written in one parallel region, but only read by other parallel regions.

InfluxProfiler considers memory-write within each parallel region to reduce the number of scenarios to be profiled. In Figure 6, parallel regions A and C read D_α and are executed on the CPU. Parallel region B reads D_γ and writes D_α and is executed on the GPU. The footprint size of parallel region B is larger than the CPU cache size. If InfluxProfiler does not consider that memory object D_α is written on parallel region B, it would suggest that offloading parallel region B will increase the cache-hit ratio on parallel region C. However, since D_α is modified on the GPU, D_α from parallel region A cannot be used by region C. InfluxProfiler identifies such scenarios and removes them from the potential offloading scenarios.

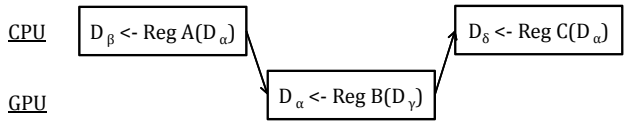

Figure 6: Considering memory-write within parallel regions.

4) Handling Loops: Until now, we have assumed that all parallel regions are executed only once. However, in many applications, parallel regions are executed multiple times in a loop. A naive way of handling loops is to treat all invocations of a parallel region as separate parallel regions. However, this method is not practical since the number of offloading scenarios is largely dependent on the number of parallel regions.

InfluxProfiler utilizes the finding that the memory access pattern of a parallel region does not change between different loop iterations in general. Even though it is still possible for an application to have less uniform loop structure, many applications with large TLP that can benefit from offloading share this characteristic of uniform behavior. The data access pattern of the first execution of a parallel region remains the same on other executions of the region, which is true for most of our evaluated benchmarks. If non-uniform behavior appears, we split a loop into multiple loops to assure uniform behavior in a single loop. Figure 7 shows an example with control divergence in a loop. The outcome of the branch on parallel region A is different on different iterations. We split this loop into two separate loops.

InfluxProfiler transforms the loop into a linear form in order to apply the mechanism described in previous sections. Figure 8 shows an example when parallel regions A, B, and C are inside a loop and are executed N times. Left side of Figure 8 shows the original CFG and right side of Figure 8 shows the different transformed CFGs depending on the parallel region of interest. CFG (a) in Figure 8 represents when we want to see whether the number of cache misses can be reduced on parallel region A due to cache-to-cache influx. The parallel region of interest is moved to the successor of all other parallel regions in the loop. The parallel region out of the loop is not considered.

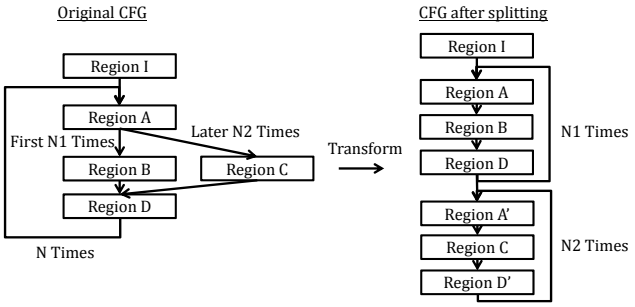

Figure 7: Loop splitting.

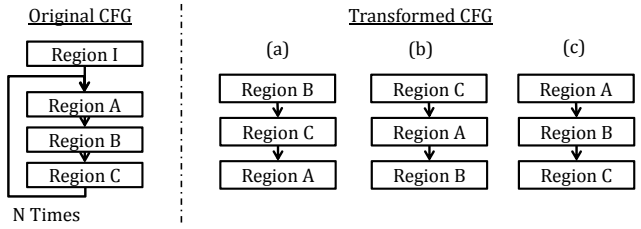

Figure 8: Different loop transformation for different parallel regions of interest.

InfluxProfiler also includes the scenario of executing only a single parallel region in a loop on the CPU as the offloading scenario to be profiled. It is because cached data by previous invocation of a parallel region can be reused in later invocations of the same parallel region.

5) Compression: To reduce the number of offloading scenarios, InfluxProfiler also merges different offloading scenarios into a single offloading scenario. For example, if scenarios to be profiled before compression are executing parallel regions $\{C, D\}$, $\{A, C, D\}$ on CPUs, they can be reduced to $\{A, C, D\}$ without any loss of accuracy. This is because scenario $\{C, D\}$ captures the cache-to-cache influx from region C to D and scenario $\{A, C, D\}$ captures the cache-to-cache influx from region A to C and the one from regions A and C to D. The cache-to-cache influx that scenario $\{C, D\}$ captures is included in scenario $\{A, C, D\}$.

6) Offloader: Offloader is the module that executes the OpenCL code with different offloading scenarios. Potential offloading scenarios with potentially reduced cache misses on the CPU are passed to Offloader. Offloader offloads different parallel regions on GPUs depending on offloading scenarios and measures performance changes of parallel regions on CPUs.

The main goal of Offloader is to preserve the cache behaviors of CPUs on different offloading scenarios, The performance of parallel regions offloaded on GPUs is not measured, which removes the needs to have an optimized GPU code. Only the functional outcomes of offloaded parallel regions are used to continue computations on CPUs. The target of offloading is not just limited to the GPU, which can

be other CPUs that do not alter the behavior of the CPU to be measured.

Since current implementation of HPerf is based on OpenCL, Offloader is built as an OpenCL delegator library that invokes OpenCL libraries from vendors. It implicitly initializes GPUs and manages data transfer between the CPU and the GPU. Data transfer between the CPU and GPU is minimized by only transferring missing data in the computing platform that will execute the parallel region.

V. Experimental Evaluation

A. Methodology

We evaluate HPerf's benefit with the Parboil benchmarks [20], [21]. [8] To evaluate the profiling overhead, we also evaluate applications from NPB [22]. We use the NPB benchmark only to evaluate InfluxProfiler. [9] Table III summarizes the characteristics of the benchmarks. We measure the wall-clock execution time on the system in Table IV.

Table III: Workload characteristics.

Benchmark	Suite	# of Regions	Problem Size
mri-fhd	Parboil [20]	2	default
mri-q(1)	Parboil [20]	2	default
rpes	Parboil [20]	2	default
histogram	Parboil [21]	4	default
mri-q(2)	Parboil [21]	2	default
sad	Parboil [21]	3	default
BT	NPB [22]	22	class A
FT	NPB [22]	8	class S, W, A
SP	NPB [22]	26	class W
IS	NPB [22]	6	class W
CG	NPB [22]	32	class W, A

Table IV: Experimental machine configuration.

CPU	Intel(R) Core(TM) i5-3550 CPU
Cache	L1D/L2/L3: 128K/1M/6M
FP peak performance	105.6 GFLOPS
Core frequency	3.30 GHz
DRAM	4GB
GPU	NVidia Quadro 2000
FP peak performance	480 GFLOPS
Shader Clock frequency	1250 MHz
DRAM	1GB
O/S	Ubuntu 12.04.1 LTS
Platform	AMD OpenCL Platform for CPU NVidia OpenCL Platform for GPU

[8]Single kernel benchmarks in Parboil are not included in this study.

[9]The OpenCL NPB benchmark has huge performance differences between CPUs and GPUs. The performance delta between CPUs and GPUs is greater than 10x. Both Measure-Separately and HPerf decide ALL-GPUs for this benchmark. So we only use the NPB benchmark to demonstrate the scalability of InfluxProfiler algorithm since it has many number of regions to effectively demonstrate the benefits of the InfluxProfiler algorithm.

B. Evaluation of BypassChecker (Stages-1, 2)

1) Stage-1: The outcome of Stage-1 is whether All-CPU or All-GPU is good enough. It should be noted that the decision is the approximated decision by BypassChecker. BypassChecker categorizes applications into simple applications and complex ones and maps all parallel regions on CPUs or GPUs for the simple ones. In our evaluation, the categorization of BypassChecker is identical to that of the oracle decision by actual measurement for the evaluated benchmarks. Following is the outcome of Stage-1.

- *Simple*: mri-fhd, mri-q(1), rpes, mri-q(2), sad
- *Complex*: histogram

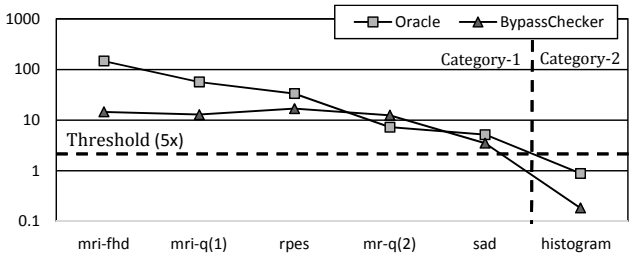

Figure 9: Predicted speedup from BypassChecker and measured speedup of ALL-GPU execution over ALL-CPU execution.

Figure 9 compares predicted speedup from BypassChecker and measured speedup of ALL-GPU execution over ALL-CPU. ALL-GPU execution includes data transfer cost. The horizontal dotted line represents the threshold (*5 times*) of BypassChecker deciding whether a later profiling stage is required. In general, BypassChecker's estimation approximates actual speedup to categorize applications correctly. For simple applications, the benefit of offloading is very high even with the data transfer cost, so HPerf can bypass Stages 2-4 and offload all parallel regions on GPUs. As for histogram, it can benefit from utilizing the detailed mechanism of the later profiling stage.

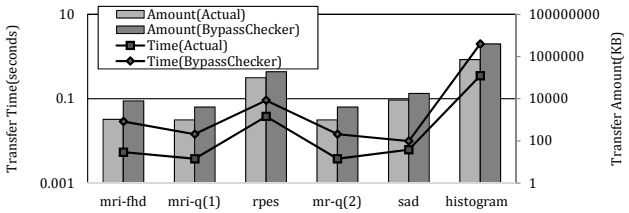

Figure 10: Estimated data transfer time and data transfer amount from BypassChecker and measured value.

To check that All-CPU or All-GPU is good enough, BypassChecker estimates the maximal data transfer time. We also evaluate how effective BypassChecker's estimation works. Figure 10 compares the estimated data transfer time and the amount of data transfer from BypassChecker and

measured value. In general, BypassChecker's estimation approximates the measured value. It should be noted that the estimated value from BypassChecker is greater than the actual value since it assumes the maximum data transfer overhead, as described in Section IV-C.

2) Stage-2: The outcome of Stage-2 is whether the cache-to-cache influx changes cache behavior significantly. Figure 11 compares the cache-hit ratio when the cache status is assumed to cold-start (*Cold-Start*) or not (*All-CPU*). The results of mri-fhd, mri-q(1), rpes, and mri-q(2) are omitted since ALL-CPU execution is the only scenario that InfluxProfiler decides to profile and the only offloading scenario when the cache-to-cache influx helps to reduce the number of cache misses. The horizontal dotted line represents the threshold (*75.0 %*) of BypassChecker. Little difference is found between *Cold-Start* and *All-CPU* since the cache-hit ratio of *Cold-Start* is still high even though the cache-hit ratio of *Cold-Start* is the worst among all possible offloading scenarios.

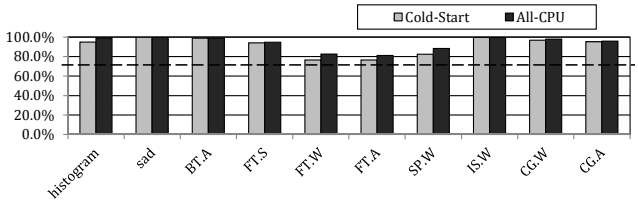

Figure 11: Cache-hit ratio of Cold-Start and All-CPU execution.

C. DataProfiler Evaluation (Stage-3)

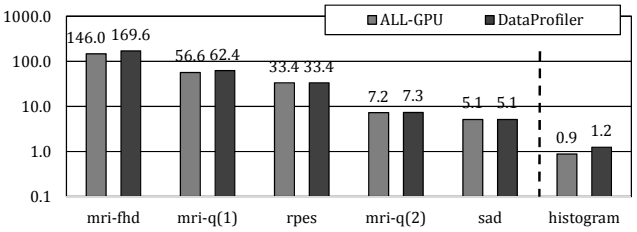

Figure 12: Speedup of DataProfiler based offloading decision and ALL-GPU execution over ALL-CPU execution. DataProfiler based offloading decision is identical to oracle offloading decision.

In order to demonstrate the benefit of DataProfiler, we compare the offloading decision from DataProfiler and the decision from Measure-Separately. For all evaluated applications, DataProfiler based offloading decision is identical to oracle offloading decision, and Measure-Separately always decides that "*offloading all computations to GPU is the optimal offloading decision*".

Figure 12 compares the speedup of the offloading decision from DataProfiler and ALL-GPU execution over the ALL-CPU execution. For *mri-fhd*, *mri-q(1)*, *mri-q(2)* and *histogram*, DataProfiler makes a different offloading decision from Measure-Separately and shows performance improvement. For *rpes* and *sad*, the offloading decisions from DataProfiler is ALL-GPU execution. For histogram, offloading all computation on GPUs is even worse than ALL-CPU execution (*0.88* times speedup). DataProfiler makes a different offloading decision from ALL-GPU execution, and achieves a *1.25* times speedup over ALL-CPU execution.

D. InfluxProfiler Evaluation (Stage-4)

1) Profiling Overhead Reduction: InfluxProfiler reduces the number of offloading scenarios that need to be profiled without missing the opportunities to identify potentially cache-to-cache influx existing scenarios. Table V represents how each component in InfluxProfiler reduces the number of scenarios. Even though we only consider parallel regions in the hot spot loop of the application, the maximum number of offloading scenarios that can be profiled (the number from Brute-Force) is still high. The number of offloading scenarios that InfluxProfiler decides to profile is significantly reduced from the maximum number of offloading scenarios, especially for the benchmarks that have many parallel regions (up to *96.4%* for *SP.W*). An exception is *IS.W*, but the maximum number of scenarios is quite small for this workload, which makes the profiling overhead negligible.

On the evaluated benchmarks, consideration of the shared data groups and footprint sizes (*SHARE/SIZE*) is the most important factor to reduce offloading scenarios. Compression (*COMP*) contributes on *SP.W*, *CG.W*, and *CG.A*. Considering data movement from sequential regions and memory-write within parallel regions (*SEQ/WRITE*) helps for *FT.S*.

2) Verification of InfluxProfiler's Outcome: As previously described, InfluxProfiler identifies potential offloading scenarios that will improve the cache hit on CPUs. Figure 13 shows the reduction of cache misses of regions running on CPUs with the identified scenario over the same regions with the ALL-CPU scenario. Please recall that we are focusing on the cache performance of parallel regions that are not offloaded and executing on CPUs. Among all the evaluated benchmarks, eight benchmarks show that offloading computation on GPUs can indeed reduce cache misses on CPUs. For *FT.W* and *FT.A*, offloading parallel regions on GPUs can reduce cache misses on CPUs to *21.2%* and *19.9%* respectively.

E. Discussion

1) Applying to other programming languages: Even though our evaluation utilize OpenCL code, it isn't the fundamental requirement of our work. HPerf's mechanism can be applied on programs written in different programming languages such as OpenMP without major modification. The only necessary modification is tracing OpenCL APIs from

Table V: How each component in InfluxProfiler affects the number of offloading scenarios.

Benchmark	MAX	SHARE/SIZE	SHARE/SIZE & COMP	SHARE/SIZE & COMP & SEQ/WRITE (Final Outcome)
mri-fhd	3	1	1	1
mri-q(1)	3	1	1	1
rpes	3	1	1	1
histogram	15	8	8	8
mri-q(2)	3	1	1	1
sad	7	3	3	3
BT.A	1023	43	43	43
FT.S	31	13	13	12
FT.W	31	10	10	10
FT.A	31	10	10	10
SP.W	16383	691	582	582
IS.W	7	7	7	7
CG.W	1023	374	200	200
CG.A	1023	374	200	200

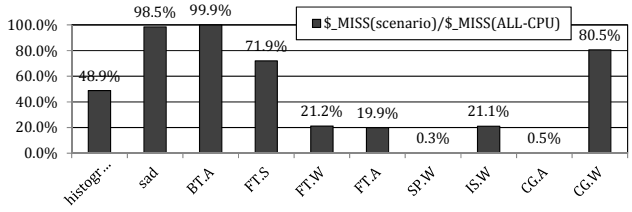

Figure 13: The ratio of the cache misses of an offloading decision from HPerf to the cache misses of ALL-CPU execution.

Analyzer, which can be changed to different styles such as annotation-based approach utilizing CHiP.

HPerf doesn't assume already implemented code to perform data transfers between CPUs and GPUs for all parallel regions either. The size and range of data movement between sequential regions and parallel regions can be easily obtained by programmers annotations, which is straightforward for many applications.

2) Input Sensitivity: In this paper, we have not considered the input sensitivity issue. Profiler-based mechanisms are not free from the input-sensitive problem. Compilers or programmers can use HPerf to decide which parallel regions should be offloaded for various input sizes. During execution, the OpenCL runtime library or any other runtime system can monitor input sizes and decide whether a parallel region is worth offloading. HPerf can also provide hints to help make these decisions. The detailed runtime mechanism is beyond the scope of this paper.

3) Future Work: Two main interference between parallel regions are "*cache-to-cache influx*" and "*memory contention between CPUs and GPUs*". For simplicity, HPerf assumes discrete memory systems for CPUs and GPUs, so memory contention does not exist on this platform. However, memory contention can have performance impact on fused architecture, so our future work will cover this.

Currently, the unit of task for offloading by HPerf is an OpenCL kernel. For certain workloads, a finer granularity of task than a kernel can enable a better task distribution decision. HPerf also assumes that the regions that could potentially be offloaded are known a priori. We will extend HPerf regarding these challenges in future work.

HPerf can be extended to other parallel programming paradigms and is not just limited to heterogeneous computing applications with CPUs and GPUs. For example, the programmer or the O/S can utilize HPerf for different thread scheduling algorithms on multi-socket CPUs considering data movement between sockets.

4) Cache-to-Cache Influx: Though the evaluated benchmarks in this paper show already high cache-hit ratio even with *Cold-Start* and therefore little performance change due to cache-to-cache influx, we believe that, depending on workloads, cache behavior can change the optimal offloading decision, which previous works have ignored. Our evaluation shows that InfluxProfiler makes it easy to identify offloading scenarios that reduce the number of cache misses by limiting the scenarios to be profiled, which is helpful for these workloads. The mechanism of InfluxProfiler is general so that it can be applied on other parallel programming paradigms without heavy modifications.

While CHiP is used as a module in Analyzer in HPerf, programmer can also verify the change of cache misses by integrating CHiP with Offloader. By creating a single cache structures in CHiP for all parallel regions executed on CPU, CHiP can mimic cache state between parallel regions on different offloading scenarios.

VI. Related work

While many run-time systems proposed also have different task distribution mechanisms, they have multiple limita-

tions such as focusing on single parallel region or relying on programmers. Our work is static-time task distribution to take advantage of compilation-time optimizations and run-time scheduling can improve performance further with the static-time optimization. In this section, we compare HPerf to previous proposals.

A. Anaytical Modeling

Analytical modeling has been proposed as a way of helping programmers to port their applications for heterogeneous platforms. Hong and Kim [15] and Sim et al. [16] assert that the key for understanding GPU performance is identifying the Memory Warp Parallelism (MWP), and Computation Warp Parallelism (CWP). Understanding cache behavior on GPU is also important on their work, because it change the CWP, MWP, therefore change the performance of GPU kernel, since recent GPGPU architectures have a hardware-managed cache.

B. GPGPU Profiling Tools

The importance of the trade-off between computation and communication has been emphasized by several GPU programming optimization guidelines [23]. Even though multiple tools such as ATI Stream Profiler [24] and NVIDIA Visual Profiler [25] are provided by vendors, these tools are to help programmers to analyze performance bottlenecks for already offloaded computations.

C. CPU-GPU Partitioning

Several frameworks on heterogeneous platforms are introduced to utilize both CPUs and GPUs. Qilin [26] uses curve-fitting to construct linear equations and dynamically maps computations on CPUs and GPUs, taking the application, the input size, and hardware resources into account. Similarly, Grewe and O'Boyle [20] statically partition work based on machine learning based performance prediction with an off-line profiler. However, these works focused on dividing only one single parallel region, while we study the interactions between multiple parallel regions.

D. Cross-Platform Performance Prediction

Several techniques have been proposed for cross-platform performance prediction. Lee and Brooks [27] use regression modeling to estimate application performance on different hardware configurations. GROPHECY [28] is an analytical model based GPU performance projection framework utilizing a code skeleton for CPU code that is a high-level code representation. GROPHECY explores different optimizations on GPUs to achieve the best GPU execution time. These works mainly focus on computation time on different architectures.

E. Automatic Code Generation

In order to reduce the programmer's burden of implementing different versions of code on different architectures, multiple automatic code generation frameworks have been proposed. Ocelot [29] provides an automatic code conversion tool between CPUs and GPUs. OpenACC [4] encapsulates the explicit accelerator management including data transfer and allows programmers to specify code regions to be offloaded by compiler directives that are automatically translated to GPU code. SnuCL [30] extends OpenCL to clusters and utilizes the CPU for kernel execution by serializing work-items. Although these automatic code generation frameworks reduce the programmer's burden of writing multiple versions of code, these works still rely on programmers for offloading decisions.

F. Data Movement Optimization

Several optimization techniques with trade-off analysis between communication overhead and the benefit of computation for heterogeneous computing systems have been introduced lately. Becchi et al. [7] study data layout optimization techniques to reduce data transfer cost and also utilize a data-aware scheduling algorithm to hide data transfer latency. CGCM [9] consists of a runtime library and compiler transformation for communication optimization between the CPU and GPU. DymanD [10] replaces static alias analysis and type inference of CGCM by runtime library. Even though these works tried to minimize the data transfer overhead with different optimizations, they do not answer the initial question of which parallel regions to offload. Our techniques can be applied on top of other techniques to minimize the amount of data transfer.

G. Decision Mechanism

Several decision mechanisms for offloading decisions have been proposed. uCLbench [11] is a microbenchmark suite for hardware characterization. It reflects different characteristics of the same workload on different architectures. Depending on the workload, the decision of where to execute the parallel region is changed. O'Boyle et al. [14] propose an automatic code generation framework from OpenMP code to OpenCL code and a prediction model to decide whether to execute the code on GPUs or on CPUs. Their prediction and offloading decisions are based on the static code features of each parallel region without considering the interactions between multiple parallel regions. Merge [12] is a map-reduce application framework with dynamic scheduling relying on users to provide the suitability of a kernel for the device. Jiménez et al. [13] study the multi-application scheduling policies on a heterogeneous architecture. They run each application on all devices and use the measured performance data to make a decision. This is the same as the Brute-Force approach. Even though multiple applications can benefit from these approaches, they have limitations

such as the assumption of the existence of GPU codes; they also assume each individual region's execution time remains constant regardless of the offloading decisions of other regions. To the best of our knowledge, our work is the first work that studies interactions of data movement and data reuse between multiple parallel regions in order to decide which parallel regions to offload.

VII. Conclusions

Programming for heterogeneous platforms is not an easy task. Even though programmers can reduce execution time by offloading computation, understanding performance behavior and finding the optimal way to utilize different computing resources have been challenging in many parallel programs.

In this paper, we propose a profiler, HPerf, to identify which parallel regions to offload from CPU applications with low profiling overhead. The proposed profiler considers interactions between multiple parallel regions, which were previously not studied much. HPerf identifies data transfer cost and cache behavior changes on CPUs depending on offloading scenarios. With the InfluxProfiler, profiling overhead is reduced significantly. DataProfiler identifies different data transfer costs depending on offloading scenarios. We demonstrate the benefits of HPerf with OpenCL applications.

References

[1] I. Buck, "Gpu computing with nvidia cuda," in *ACM SIGGRAPH 2007 Courses*, ser. SIGGRAPH '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1281500.1281647

[2] Qualcomm, "Adreno, qualcomm's integrated graphics solution," http://www.qualcomm.com/products_services/chipsets/multimedia/graphics.html, 2008.

[3] OpenCL, "The open standard for parallel programming of heterogeneous systems," http://www.khronos.org/opencl, 2009.

[4] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openacc: First experiences with real-world applications," in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 859–870. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32820-6_85

[5] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Rigel: An Architecture and Scalable Programming interface for a 1000-core accelerator," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, Jun. 2009.

[6] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, and S. Han, "Comic: A coherent shared memory interface for cell be," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 303–314. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454157

[7] M. Becchi, S. Byna, S. Cadambi, and S. Chakradhar, "Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 82–91. [Online]. Available: http://doi.acm.org/10.1145/1810479.1810498

[8] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 134–144. [Online]. Available: http://dx.doi.org/10.1109/ISPASS.2011.5762730

[9] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic cpu-gpu communication management and optimization," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 142–151. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993516

[10] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August, "Dynamically managed data for cpu-gpu architectures," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012, pp. 165–174. [Online]. Available: http://doi.acm.org/10.1145/2259016.2259038

[11] P. Thoman, K. Kofler, H. Studt, J. Thomson, and T. Fahringer, "Automatic opencl device characterization: Guiding optimized kernel design," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 438–452. [Online]. Available: http://dl.acm.org/citation.cfm?id=2033408.2033459

[12] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: A programming model for heterogeneous multi-core systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 287–296. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346318

[13] V. J. Jiménez, I. Gelado, M. Gil, G. Fursin, N. Navarro *et al.*, "Predictive runtime code scheduling for heterogeneous architectures," in *HiPEAC 2009-High Performance and Embedded Architectures and Compilers*, 2009.

[14] M. F. P. O'Boyle, Z. Wang, and D. Grewe, "Portable mapping of data parallel programs to opencl for heterogeneous systems," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, ser. CGO '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/CGO.2013.6494993

[15] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 152–163. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555775

[16] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in gpgpu applications," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 11–22. [Online]. Available: http://doi.acm.org/10.1145/2145816.2145819

[17] B. Brett, P. Kumar, M. Kim, and H. Kim, "Chip: A profiler to measure the effect of cache contention on scalability," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE,

2013, pp. 1565–1574.

[18] Pin, *A Binary Instrumentation Tool*, http://www.pintool.org.

[19] A. S. Dhodapkar and J. E. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ser. ISCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 233–244. [Online]. Available: http://dl.acm.org/citation.cfm?id=545215.545241

[20] D. Grewe and M. F. P. O'Boyle, "A static task partitioning approach for heterogeneous systems using opencl," in *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, ser. CC'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 286–305. [Online]. Available: http://dl.acm.org/citation.cfm?id=1987237.1987259

[21] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," *Center for Reliable and High-Performance Computing*, 2012.

[22] S. Seo, G. Jo, and J. Lee, "Performance characterization of the nas parallel benchmarks in opencl," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, ser. IISWC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 137–148. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2011.6114174

[23] C. Cuda, "Programming guide," *NVIDIA Corporation (July 2012)*, 2012.

[24] B. Purnomo, N. Rubin, and M. Houston, "Ati stream profiler: A tool to optimize an opencl kernel on ati radeon gpus," in *ACM SIGGRAPH 2010 Posters*, ser. SIGGRAPH '10. New York, NY, USA: ACM, 2010, pp. 54:1–54:1. [Online]. Available: http://doi.acm.org/10.1145/1836845.1836904

[25] NVIDIA Corporation, "NVIDIA Visual Profiler," http://developer.nvidia.com/content/nvidia-visual-profiler, 2011.

[26] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 45–55. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669121

[27] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: ACM, 2006, pp. 185–194. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168881

[28] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram, "Grophecy: Gpu performance projection from cpu code skeletons," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 14:1–14:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063402

[29] G. Diamos, "The Design and Implementation Ocelot's Dynamic Binary Translator from PTX to Multi-Core x86," Georgia Institute of Technology, Tech. Rep. GIT-CERCS-09-18, 2009.

[30] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "Snucl: An opencl framework for heterogeneous cpu/gpu clusters," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 341–352. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304623