On-The-Fly Workload Partitioning for Integrated CPU/GPU Architectures

Younghyun Cho
Seoul National University

Florian Negele
ETH Zürich

Seohong Park
Seoul National University

Bernhard Egger
Seoul National University

Thomas R. Gross
ETH Zürich

ABSTRACT

Integrating CPUs and GPUs on the same die provides new opportunities for optimization, especially for irregular data-parallel workloads that fail to fully exploit the computational power of the GPU. Such workloads benefit from a proper partitioning between the CPU and the GPU. This paper presents an on-the-fly workload partitioning technique for irregular workloads on integrated architectures. Unlike existing work, no prior analysis of the workload is required. GPU kernels and input data are analyzed and optimized at runtime. The technique executes work chunks of similar load on the GPU and assigns irregular chunks to the CPU. Evaluated with various irregular workloads, the method achieves a 1.4x–7.1x speedup over GPU execution on AMD and Intel processors.

CCS CONCEPTS

• **Computer systems organization** → *Heterogeneous (hybrid) systems*; • **Computing methodologies** → *Parallel computing methodologies*;

KEYWORDS

Integrated architecture, workload partitioning, OpenCL

ACM Reference Format:
Younghyun Cho, Florian Negele, Seohong Park, Bernhard Egger, and Thomas R. Gross. 2018. On-The-Fly Workload Partitioning for Integrated CPU/GPU Architectures. In *International conference on Parallel Architectures and Compilation Techniques (PACT '18), November 1–4, 2018, Limassol, Cyprus.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3243176.3243210

PACT '18, November 1–4, 2018, Limassol, Cyprus

ACM ISBN 978-1-4503-5986-3/18/11…$15.00
https://doi.org/10.1145/3243176.3243210

1 INTRODUCTION

Integrating CPUs and GPUs on the same die is common in modern processor architectures. AMD's accelerated processing units (APU) [2] and Intel's Skylake processor and its successors [6] integrate multi-core CPUs and GPUs into a single chip. Thanks to the shared off-chip memory, such integrated architectures promise easy collaboration between the CPU and the GPU because the costly data copy operations from main memory to the GPU's internal memory can be avoided [7, 26].

A common strategy for CPU-GPU collaboration is to partition the work of data-parallel workloads. Programming models such as OpenCL [12] and CUDA [22] abstract data-parallel workloads with small work units, termed work-items in OpenCL. Work-items are grouped into work-groups that constitute the smallest unit of workload distribution. On integrated architectures, work-groups can be assigned to CPU cores and the GPU's compute units to increase application performance without requiring data copying [11].

The single instruction, multiple thread (SIMT) [17] execution model of today's GPUs does not cope well with irregular applications. Under SIMT, the work-items of a work-group are executed in lock-step, and execution a work-group ends when the longest-running work-item terminates. For work-items with an irregular computational load, this severely affects the througput of the GPU. Many important workloads that consist of nested loop structures, such as sparse matrix-vector multiplication and graph algorithms, are examples of such GPU-unfriendly applications.

In this context, a promising approach to avoid underutilization of the GPU for irregular workloads is to reshape the workload by grouping work-items with a similar computational load into work-groups. Figure 1 illustrates how an irregular workload can be reshaped. The four darkblue work-items with a larger computational load are grouped into one work-group such that all work-groups consist of work-items with a similar computational load. This technique is particularly well suited for integrated CPU/GPU architectures since work-items can be executed on the CPU or the GPU without requiring any movement of data.

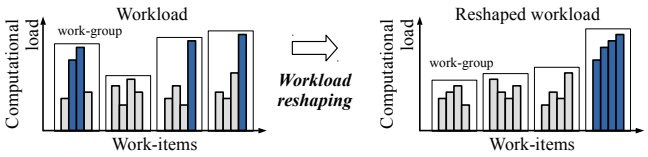

Figure 1: Reshaping an irregular workload.

Several studies have shown the effectiveness of reshaping work-items for execution on the CPU and the GPU [25, 27]. Shen et al. [25] partition an irregular workload for the CPU and the GPU based on a prior analysis of the workload. The CPU assists the GPU by executing work-groups with an irregular computational load. The required prior offline anaylsis, however, limits the practicality of the approach because of the dynamic nature of the input data. Zhang et al. [27] perform workload reshaping during the initialization of an application's input data based on a machine-learned performance model. This approach is not ideal because the performance model needs to be trained offline for each system and each application with several input data sets. Moreover, the method can only reshape workloads during initialization but is unable to handle dynamic workloads that are generated during an application's lifetime. These limitations call for an automatic approach that can handle dynamic workloads at runtime for various hardware platforms while not depending on a prior analysis or training of a model.

To this end, this paper presents an on-the-fly workload partitioning technique for irregular data-parallel workloads on integrated architectures. The technique neither requires an offline analysis nor training with workloads or input data. For an OpenCL kernel, a source-to-source compiler dynamically creates profiling code that allows the runtime system to collect information about the computational load of the work-items immediately before the kernel is launched. Based on this profile information, the workload is reshaped such that all work-items with a high loop iteration count above a dynamically determined threshold are executed on the CPU cores while the GPU only executes only work-items below that threshold and with a similar computational load. To hide the overhead of work-item profiling and reshaping, a single workload is divided into several jobs that are launched dynamically. While one job is being executed, work-item reshaping is performed for the next job.

We have implemented the presented technique into an OpenCL framework and evaluate it on integrated architectures. Code generation and workload partitioning are performed automatically and transparently for annotated irregular loops by intercepting calls to the OpenCL API. An evaluation of irregular workloads from sparse matrix-vector multiplication, page rank, graph coloring, and a BFS algorithm on an AMD A10-7850K APU (Kaveri) [1] and an Intel Skylake i5-6600 processor [10] shows that, for various input data, the on-the-fly workload partitioning achieves a 1.4x–7.1x speedup over execution on the GPU without workload partitioning. The performance gap compared to an oracle partitioning lies within 10% for the sparse matrix-vector multiplication and page rank workloads.

To summarize, the contributions of this work are

- an approach to workload partitioning based on workload reshaping that leverages code generation and runtime interaction (Section 3).
- an on-the-fly workload partitioning technique that detects workload irregularities at runtime and reshapes the workload with a very low overhead (Section 4).
- an OpenCL framework that automatically performs the presented workload partitioning for OpenCL GPU applications (Section 5).
- an evaluation of the on-the-fly workload partitioning technique for various irregular workloads on two CPU/GPU integrated processors (Section 6).

2 BACKGROUND

In this section, we explain how workload partitioning is implemented on integrated architectures and discuss optimization issues of irregular workloads.

2.1 Workload Partitioning

A runtime system partitions a data-parallel workload for multiple devices to increase performance. It calculates the partitioning information (i.e. work-group spaces for different devices) and copies the data buffers to each device. On integrated CPU/GPU architectures, workload partitioning is much simpler compared to systems with separate memories because the CPU and the GPU share the same off-chip memory resources.

Figure 2 shows how to partition an OpenCL workload for CPU and GPU execution on (a) a discrete memory system and (b) a unified memory system found in integrated architectures. For discrete memory systems, a data buffer object is created and `clEnqueueWriteBuffer()` is called to link this buffer with data buffers on the CPU and the GPU (lines 1–3). The actual data copy to GPU memory occurs in line 3. Once the copy operation has completed, the kernel is launched using `clEnqueueNDRangeKernel()` on both devices. The information about the partitioning is given by `cpu/gpu_work_offset` and `cpu/gpu_work_size` (lines 6–7). At the end, a barrier has to be inserted to make sure both devices have finished execution of the kernel (line 8). The output data from the different devices need to be merged accordingly during the synchronization step to obtain the correct result. On a unified memory architecture, on the other hand, workload partitioning is much simpler. By setting the `CL_MEM_USE_HOST_PTR` flag for the data buffer in

```
void workload_partitioning(float* A, size_t size_A)
{
#if defined(DISCRETE_MEMORY)      // (a) discrete memory system

1:  cl_mem buf = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                size_A, NULL, NULL);
2:  clEnqueueWriteBuffer(cpu_queue, buf, CL_TRUE, 0,
                         size_A, A, cpu_offset, NULL, NULL);
3:  clEnqueueWriteBuffer(gpu_queue, buf, CL_TRUE, 0,
                         size_A, A, gpu_offset, NULL, NULL);

#elif defined(UNIFIED_MEMORY)     // (b) unified memory system

4:  cl_mem buf = clCreateBuffer(context,
                                CL_MEM_READ|CL_MEM_USE_HOST_PTR,
                                size_A, A, NULL);
#endif

5:  clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&buf);
6:  clEnqueueNDRangeKernel(cpu_queue, kernel, work_dim,
                           cpu_work_offset, cpu_work_size,
                           local_work_size, 0, NULL, NULL);
7:  clEnqueueNDRangeKernel(gpu_queue, kernel, work_dim,
                           gpu_work_offset, gpu_work_size,
                           local_work_size, 0, NULL, NULL);
8:  synchronization();
}
```

Figure 2: Programming workload partitioning.

line 4, the OpenCL runtime is instructed that both CPU and GPU access the same data. Each work-item can be executed on the CPU or the GPU immediately without requiring any data copying. Also, no merging of output data is necessary during synchronization (line 8) on integrated architectures.

This work targets integrated architectures with a unified memory system. The evaluation is performed on two concrete platforms, an AMD A10-7850K APU (codename Kaveri) [1] and an Intel Skylake i5-6600 processor [10]. Both architectures support the workload partitioning method for unified memories as shown in Figure 2. This paper assumes applications are written in OpenCL. Today's OpenCL and CUDA programming models commonly support almost the same functionalities [14]; the presented techniques, in principle, can also be applied to CUDA runtimes and applications.

2.2 Irregular Workloads

For irregular workloads such as sparse matrix-vector multiplication (SpMV) and graph algorithms (e.g. PageRank), the individual work-items exhibit a different computational load. In the case of SpMV, each work-item processes the column elements of one row. The workload imbalance is caused by a varying number of non-zero elements per row. For graph algorithms, one work-item typically processes one vertex. Vertices with a varying number of edges cause a workload imbalance. Such applications are implemented with nested loop structures. The outer loop iterates over the rows of a matrix or the vertices, and the inner loop processes all non-zero elements of the row or edges of the vertex. The iterations of the outer loop are mapped to work-items which then contain loops with a varying number of iterations in dependence of the input data.

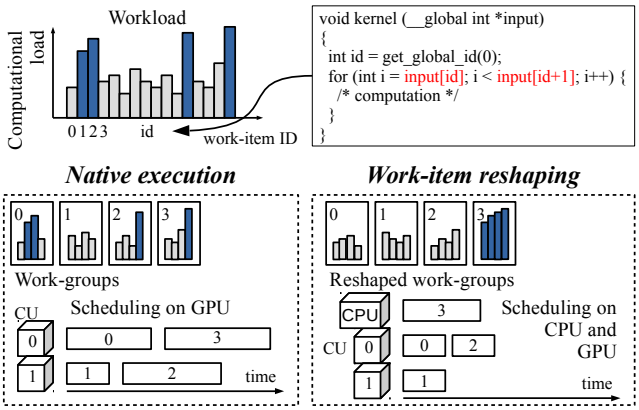

Figure 3: The effectiveness of work-item reshaping.

Figure 3 shows how such workloads are executed on GPUs and what the potential benefits from reshaping the workload are. Each work-item accesses its data using its identifier obtained by calling`get_global_id()`. The computation occurs in a loop inside the kernel. By default, work-items are linearly grouped into work-groups. The number of work-items per work-group is configurable; in the figure, the 16 work-items are split into four work-groups. Work-groups are allocated to a CPU core or a compute unit (CU) in the GPU. On a CPU core, the work-items of a work-group are executed serially [13, 15] or using user-level threads [8]. On the GPU, one CU, comprising a set of processing elements (PE), executes one work-group. The PEs operate in SIMT mode to execute the work-items. *Native execution* illustrates how a workload is executed on a GPU with conventional work-group scheduling based on a round-robin algorithm. With the GPU's SIMT execution model, the execution time of a CU is determined by the execution time of the work-item that takes the longest to complete. In this case, the CUs in the GPU suffer from a longer execution time for work-groups 0, 2, and 3 caused by the darkblue work-items exhibiting a higher computational load. On the other hand, with *Work-item reshaping*, we collectively group the four longer work-items shown in darkblue and allocate them to a CPU core. If the CPU is fast enough, we can salvage the underutilization of the GPU cores and reduce the execution time of the irregular workload.

This suggests that fine-grained workload partitioning is beneficial for irregular workloads. To reshape work-items, the number of iterations in the loop of a work-item can be used as an indicator of its computational load. Many implementations for sparse matrix computations and graph algorithms (e.g. BFS, GraphColoring, TriangleCount, etc.) contain kernels consisting of a one-dimensional irregular loop that can benefit from this approach [20].

3 METHODOLOGY

This section discusses the design of the proposed technique and analyzes competing state-of-the-art approaches.

3.1 Workload Partitioning

Our approach to workload partitioning is based on a source-to-source transformation of OpenCL kernels and a runtime system that leverages this code generation. Figure 4 shows an OpenCL kernel implementation [9] of Brin and Page's PageRank algorithm [3]. The graph is stored using the compressed sparse row (CSR) format [24]. A work-item computing the new rank of a vertex contains one loop (lines 4 – 5) by accumulating the rank values of the connected vertices. The number of loop iterations (i.e., the number of connected vertices) is stored in an input data buffer named `rowstr`. To benefit from the proposed workload partitioning, irregular OpenCL kernels are annotated with the "irregular" pragma directive (line 3) that appertains to the `for` statement following in line 4. This annotation informs our source-to-source transformation engine and the runtime system that the work-items of the corresponding loop exhibit an irregular number of iterations. At the moment, the annotation has to be inserted by the programmer; automatic detection of irregular loops is part of future work. Once this annotation is given, the techniques presented in the paper are fully automatic.

The workload partitioning consists of the three steps *Profiling*, *Reshaping*, and *Partitioning*. In the first step, *Profiling* collects the number of loop iterations for each work-item. The profiling code, generated automatically by the source-to-source compiler, of the OpenCL PageRank algorithm is shown in Figure 5. The workload distribution of the individual work-items depends on the input data. Many iterative algorithms use the output of iteration i as the input to iteration $i + 1$, hence this profiling code of complexity $O(N)$ is executed at runtime and for each kernel launch. The *Reshaping* step determines the execution device (CPU or GPU) for each work-item. To avoid a costly rearrangement of the input data, an indirection in form of `remap` buffer of size $sizeof(int) \times \#work\_items$ is added by the source-to-source compiler that reassigns the IDs of the work-items. This `remap` buffer enables us to control what work-items are executed on which device; querying the buffer increases the runtime of the kernel only minimally (Figure 6, line 2). The last step, *Partitioning*, divides the workload for the CPU and the GPU and launches the kernel. The partitioning is executed in the OpenCL runtime by intercepting calls to the `clEnqueueNDRangeKernel()` function which contains all information about the work-item space and the work-group size of a kernel. As discussed in Section 2.1, workload partitioning can be implemented efficiently in integrated architectures.

```
__kernel void PageRank(__global int* rowstr,
                       __global float* ranks,
                       __global int* indices,
                       __global float* newRanks)
{
1:  int id = get_global_id(0);
2:  float newRank = 0.0;
3:  #pragma irregular
4:  for (int k = rowstr[id]; k < rowstr[id+1]; k++)
5:      newRank += ranks[indices[k]];
6:  newRank = (1.0 - DAM) / numPages + DAM * newRank;
7:  newRanks[id] = newRank / numPages;
}
```

Figure 4: Kernel code of PageRank using CSR.

```
__kernel void PageRank_Profiling(__global int* rowstr,
                                 __global float* ranks,
                                 __global int* indices,
                                 __global float* newRanks,
                                 __global int* loop_size)
{
1:  int id = get_global_id(0);
2:  float newRank = 0.0;
3:  loop_size[id] = rowstr[id+1] - rowstr[id];
4:  for (int k = rowstr[id]; k < rowstr[id+1]; k++)
5:      newRank += ranks[indices[k]];
6:  newRank = (1.0 - DAM) / numPages + DAM * newRank;
7:  newRanks[id] = newRank / numPages;
}
```

Figure 5: Generated code for workload profiling.

```
__kernel void PageRank_Reshaping(__global int* rowstr,
                                 __global float* ranks,
                                 __global int* indices,
                                 __global float* newRanks,
                                 __global int* remap)
{
1:  int id = get_global_id(0);
2:  id = remap[id];
3:  float newRank = 0.0;
4:  for (int k = rowstr[id]; k < rowstr[id+1]; k++)
5:      newRank += ranks[indices[k]];
6:  newRank = (1.0 - DAM) / numPages + DAM * newRank;
7:  newRanks[id] = newRank / numPages;
}
```

Figure 6: Generated code for workload reshaping.

3.2 Workload Reshaping

Once a workload has been profiled it has to be reshaped for execution on the CPU and GPU devices. We first look at state-of-the art, then discuss the performance and overhead of the different techniques.

3.2.1 State-of-the-art. We consider two existing methods performing workload reshaping for execution on CPUs and GPUs for comparison. Shen et al. [25] target systems comprising CPU and GPU devices with discrete memory. The input data of work-items is sorted in order of their computational demand. Work-items with higher computational load are assigned to the CPU. The division of the work-item space (i.e., the ratio of work-items assigned to the CPU and the GPU) is based on a performance model that considers the performance and the data copy overhead of the CPU and the GPU. In contrast, our target platforms are integrated architectures that do not require costly sorting and data copying operations.

Algorithm 1 Sort based partitioning.

```
1: function SORT-H(#_items, loop_size, partition)
2:     array = sort_ascending(loop_size)
3:     array_gpu = array[0 : partition − 1]
4:     array_cpu = array[partition : #_items]
5:     return {array_gpu, array_cpu}
```

Algorithm 2 Threshold based shifting.

```
1: function SHIFT-H(#_items, loop_size, threshold)
2:     for id = 0 → #_items do
3:         if loop_size[id] ≤ threshold then
4:             array_gpu.append(id)
5:         else
6:             array_cpu.append(id)
7:     return {array_gpu, array_cpu}
```

FinePar [27], on the other hand, performs work-item reshaping using a threshold value for integrated CPU/GPU architectures. If the computational load of a work-item is higher than the threshold value, the work-item is assigned to the CPU and vice-versa. For a given input data set, FinePar determines the threshold for reshaping based on a performance model and reshapes the workload while reading the input data from disk. The linear regression model uses a carefully tuned number of features such as the average elements per row and their variation and distribution. A disadvantage of this method is that the model has to be trained offline for each application and each target platform with different input data sets to capture the performance behavior of irregular workloads.

3.2.2 Performance Analysis. Unlike the previous proposals, we focus on a dynamic approach that does not require any offline efforts from users and can be directly applied to different platforms. Furthermore, we perform workload reshaping at every kernel launch to be able to cope with input data generated at runtime. The overhead of profiling and reshaping, therefore, must be low. To compare the benefits and the overhead of state-of-the-art reshaping methods, we evaluate the following four reshaping strategies.

- **Sort-H** sorts all work-items according to the number of loop iterations and assigns work-items with high computational load to the CPU (Algorithm 1) [25].
- **Sort-L** sorts work-items as `Sort-H` but assigns work-items with lower computational load to the CPU.
- **Shift-H** assigns work-items with a loop iteration count above a given threshold to the CPU [27]. The work-items are not sorted. Algorithm 2 details this approach.
- **Shift-L** is similar to `Shift-H` but assigns work-items of with a load below a given threshold to the CPU.

Oracle partitioning (percentage of work-items assigned to the CPU)

Method	Fullchip	Scale20	In2004	Wats.	LDBC	Knowl.
Sort-L	100%	65%	100%	100%	65%	100%
Sort-H	2%	1%	4%	1%	18%	61%
Shift-L	100%	100%	44%	100%	100%	17%
Shift-H	2%	2%	2%	1%	30%	1%

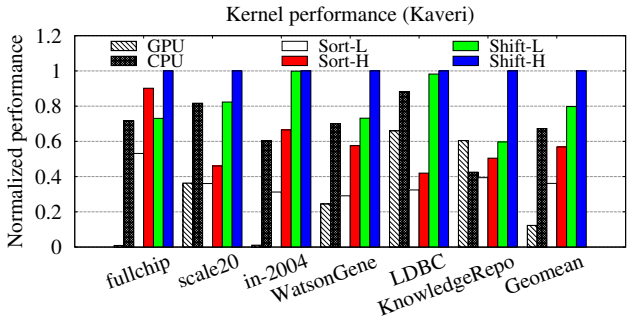

Figure 7: PageRank kernel performance for varying input data with optimal partitioning under different reshaping methods. Performance is normalized to `Shift-H`; the reshaping overhead is excluded. The four cores of the `CPU` are assigned an equal number of work-items.

Figure 7 shows the performance of the four reshaping methods for the PageRank kernel described in Figure 4 using six different input data sets. The results only show the pure execution time, the overhead for the analysis and sorting/shifting is excluded. The experiments are performed on an AMD Kaveri APU [1]. The performance of workload partitioning depends on the ratio of assigned work-items for the CPU and the GPU and the threshold value. To see the potential benefit of each method, we plot the optimal performance under each reshaping strategy with the best partitioning point found through an exhaustive search. The characteristics of the input data strongly affect the performance of the workload. Most rows of the `fullchip` input, for example, contain only few elements with a few outliers containing a very large number of elements. As a consequence, `fullchip` performs extremely poor on the GPU. On the other hand, the GPU performs well for regular workloads such as `LDBC`. The characteristics of each input data set and a more detailed performance analysis are given in Section 6.

The experimental results show that, for the most part, `Shift-H` achieves the best performance. It also significantly outperforms single-device execution on the GPU or the CPU, demonstrating the potential of workload reshaping and exploiting the different characteristics of the CPU and the GPU. We observe that both `Sort-L` and `Sort-H` perform poorly in comparison to `Shift-L` and `Shift-H`. Sorting the work-items often results in a significant performance degradation because the memory accesses of the sorted workloads cannot exploit data locality anymore. This is more critical for integrated architectures where the CPU and the GPU are limited

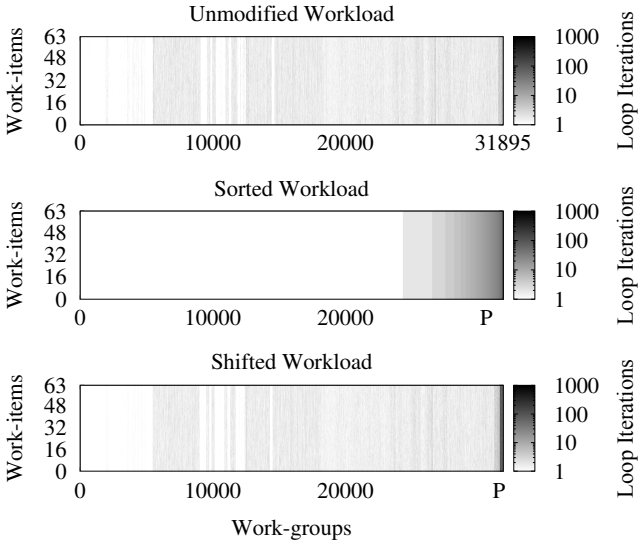

Figure 8: Workload analysis for WatsonGene. The X/Y-axis represents the work-groups and the work-items, respectively. Work-items with higher iteration counts are represented with darker color. *P* indicates the best partitioning point. Work-groups to the right of *P* are assigned to the CPU.

by a relatively small memory bandwidth compared to discrete memory systems [28]. `Shift-H` outperforms `Shift-L` but in most cases only assigns a small percentage of the workload to the CPU (Figure 7). This implies that CPUs are better suited to execute a small number of work-items with a large number of loop iterations.

Figure 8 visualizes the `WatsonGene` workload reshaped with `Sort-H` and `Shift-H`. Following a power-law distribution, the majority of work-items have a very low loop iteration count (visible by the white and lightgray colors). The best performance is achieved by `Shift-H` that assigns only the top 1% of work-items with the largest number of loop iterations to the CPU. The variance in the compuational load of the 99% of work-items assigned to the GPU is low and the work-groups benefit from the GPU's SIMT execution.

The overhead of profiling and reshaping a workload is an important concern for on-the-fly techniques. Figure 9 displays the overhead of workload profiling normalized to the kernel execution time of the `Shift-H` technique. The profiling code is executed with one and four threads on the CPU (`CPU-single` and `CPU-Parallel`) and the GPU (`GPU`). All configurations incure a significant overhead, even the best performing `CPU-Parallel`. The overhead of reshaping is shown in Figure 10 for the sorting and shifting methods. Sorting (using the C++ STL) is not a viable technique to reshape a workload at runtime due to the high computational complexity. While the reshaping overhead of shifting is considerably lower, the total overhead of profiling and reshaping is significant and a concern for online methods.

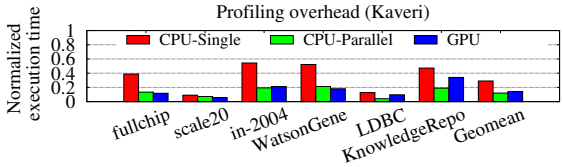

Figure 9: Profiling overhead analysis.

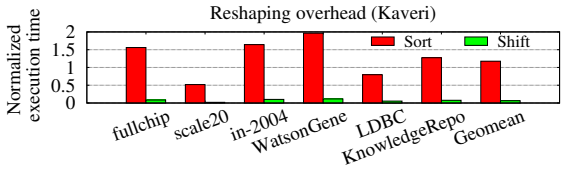

Figure 10: Reshaping overhead analysis.

3.2.3 Analysis. Based on the performance analysis performed in Section 3.2, the proposed on-the-fly workload partitioning technique employs workload reshaping based on shifting using a threshold value. Shifting provides better performance than sorting in terms of both the kernel execution time and the runtime overhead. However, it still remains a challenge to profile and reshape a workload at runtime. This is addressed in the following section.

4 ON-THE-FLY WORKLOAD PARTITIONING

To address the challenges explained above, an on-the-fly workload partitioning technique is presented herein. The technique reduces the workload partitioning overhead by dynamically launching kernels and selects a proper threshold value for workload division based on runtime feedback.

4.1 Pipelined Job Execution

When a GPU kernel is launched, we divide the total work-item space into a number of jobs that are independently profiled, reshaped, and executed. The key advantage of this approach is that it allows us to hide the overhead of profiling and reshaping of one job by overlapping it with the execution of another job. Figure 11 gives an overview of the technique.

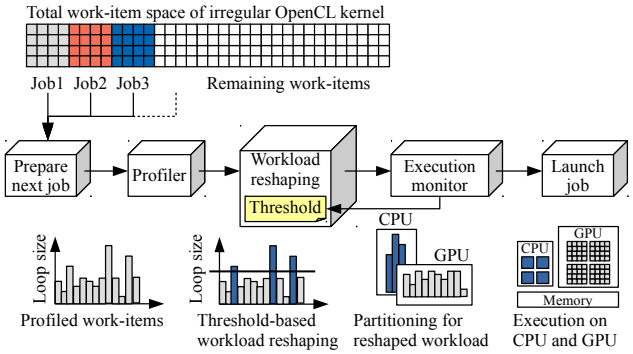

Figure 11: On-the-fly workload partitioning.

Algorithm 3 Pipelined job execution.

```
1:  procedure PipelineJobs(#total_items)
2:      offset ← 0; adjust ← 1; job_idx ← 0
3:      #items ← GetNextJob(job_idx)
4:      loop_size ← Profiler(offset, #items)
5:      threshold ← GetThreshold(loop_size, adjust)
6:      { array_gpu, array_cpu } ← Shift-H(offset, #items,
                                  loop_size, threshold) // Alg. 2
7:      offset ← #items
8:      repeat
9:          GPU_Launch(array_gpu, job_idx)        // run in
10:         CPU_Launch(array_cpu, job_idx)        // parallel
11:         #items ← GetNextJob( )
12:         loop_size ← Profiler(offset, #items)
13:         threshold←GetThreshold(loop_size, adjust)
14:         { array_gpu, array_cpu } ← Shift-H(offset,
                      #items, loop_size, threshold) // Alg. 2
15:         adjust ← ExecutionMonitor(job_idx++)
16:         offset ← offset + #items
17:     until offset ≤ #total_items
18:     Wait until all jobs have finished
```

After profiling a job, its work-items are reshaped based on a dynamically determined threshold value. Work-items with a loop iteration count higher than the threshold are assigned to the CPU, all others to the GPU. To hide the runtime overhead, the profiling and reshaping code for the next job runs on the CPU while the current job is being executed on both CPU and GPU. Thanks to the simplicity of the profiling and reshaping code, the time taken by profiling and reshaping is, in general, shorter than the execution time of a job with the same number of work-items, that is, the profiling and reshaping overhead can be completely hidden.

Algorithm 3 details how the runtime system profiles, reshapes, and executes the jobs of an OpenCL kernel. First, the initial job is fetched using the function `GetNextJob()` that also returns the number of work-items in this job (line 3). Next, the job is profiled on the CPU cores, and the loop iteration counts of the work-items are saved into the *loop_size* array (line 4). After determining the threshold for partitioning (see Section 4.2), the work-items are reshaped by assigning those with a smaller iteration count than *threshold* to the GPU and all others to the CPU (lines 5–6). The arrays *array_gpu/cpu* hold the work-item redirection IDs as detailed in Section 3.1. Once the CPU threads and the GPU kernel have been launched (lines 9–10), the following job is retrieved to be profiled and reshaped (lines 11–14). Prior to launching this job, the function `ExecutionMonitor()` is called to query the execution speed of the CPU and the GPU engines. The new value of the *adjust* variable is determined based on the their progress. The *job_idx* variable holds the index of the current job and is used to track the execution speed of the GPU and the CPU (line 15).

4.2 Determining Threshold and Job Size

Since the presented framework is not tailored towards a specific architecture, the relative performance of the CPU cores with respect to the GPU is unknown. To achieve optimal performance, the execution times of the work assigned to the CPU and the GPU should be as similar as possible. In the absence of any knowledge about the relative performance of the two devices, the framework uses an adaptive approach that dynamically adjusts the value of the threshold and thus the amount of work assigned to the CPU and the GPU.

The threshold value is computed based on the profiled loop iteration counts and a variable *adjust* as follows

$$threshold = adjust \times avg(loop\_size[offset : offset + \#items])$$

Initially, *adjust* is set to 1 for each kernel execution, that is, the threshold value of the initial job is set to the average of all loop iteration counts in the *loop_size* array. *Adjust* is updated by `ExecutionMonitor()` that checks which device has completed more jobs. This information is obtained through shared memory (CPU) and `clGetEventInfo()` on the GPU.

$$adjust = adjust \times \begin{cases} 0.8 & \text{CPU completed more jobs} \\ 1.0 & \text{otherwise} \\ 1.5 & \text{GPU completed more jobs} \end{cases}$$

A smaller value of *adjust* results in a lower threshold and thus more work being assigned to the CPU and vice-versa. The multipliers 0.8 and 1.5 have been determined empirically.

Each kernel is divided into 20 jobs by default with a minimum and maximum bound of $512 \times wg\_size$ and $8190 \times wg\_size$, respectively, for the number of work-items per job. Here, *wg_size* represents the number of work-items in a work-group. These values have also been determined empirically and found to do not affect overall performance significantly. The overhead of splitting a single OpenCL kernel into a number of independent invocations increases the total runtime by up to 10%. The performance gains obtained by on-the-fly workload partitioning, however, are larger, resulting in a net performance improvement (see Section 6).

4.3 Advantages and Limitations

The proposed technique has a number of advantages over competing approaches. First, the technique is hardware-agnostic and all required information is determined dynamically at runtime. The method automatically adjusts the workload partitioning to the speed of the CPU and the GPU in order to obtain the best-possible benefit. Second, splitting an OpenCL kernel into several jobs and execution these jobs in a pipelined fashion allows us to hide the overhead of profiling and reshaping; two steps that are required by any workload partitioning scheme. Existing methods perform these steps offline [25] or during the data initialization phase [27].

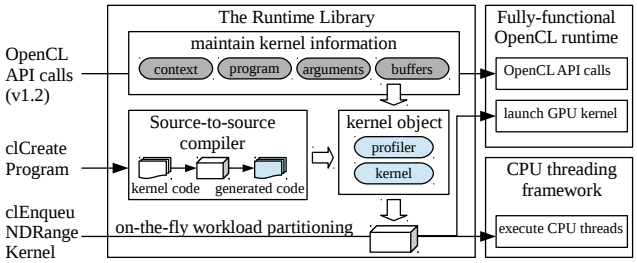

Figure 12: The software framework.

At the moment, the presented technique suffers from a number of limitations. The technique assumes that profiling and reshaping take less time than the computation to completely hide the overhead of workload partitioning. This is true for almost all analyzed benchmarks but may not hold for extremely simple OpenCL kernels. Another weakness is that inter-work-group communication is not supported. The bulk synchronous parallel model allows a barrier in work-items to enable communication between work-items of a work-group, however, we assume that there is no communication inside a work-group. Finally, the source-to-source compiler currently assumes that OpenCL kernels have a one-dimensional work-item space and contain one irregular loop.

5 IMPLEMENTATION

The structure of the framework's implementation is shown in Figure 12. The main components are a source-to-source compiler that generates the profiling code and the kernel to operate with the reshaped workload and a runtime library that manages execution of irregular OpenCL kernels.

5.1 Runtime Library

The runtime library is in charge of workload profiling, reshaping, partitioning, and controlling the execution of an OpenCL kernel. The library is activated when an OpenCL application executes. Library interpositioning is used to intercept all relevant OpenCL API calls at runtime; as a consequence, the library is compatible with any OpenCL runtime. Intercepted are calls to create a context (`clCreateContext`), a program (`clCreateProgram`), and all data buffers (`clCreateBuffer`). Calls to `clSetKernelArg` are intercepted to forward the corresponding OpenCL buffer objects or arguments values to the profiling code and the reshaped kernel. The actual partitioning is performed when the application calls `clEnqueueNDRangeKernel` for an OpenCL kernel annotated as irregular as detailed in Sections 3 and 4.

To execute OpenCL work-items on the CPU, a separate CPU threading framework has been implemented. The threading framework coalesces a group of reshaped work-items into one function call to reduce the overhead of processing each work-item separately [15].

5.2 Compiler Framework

OpenCL kernels are compiled from source by calling an OpenCL API function such as `clCreateProgramWithSource`. Our runtime library intercepts this call and invokes the internal source-to-source compiler. The compiler analyzes the OpenCL kernel code to generate the profiling code and the transformed kernel code as shown in Section 3.1. This prepares a kernel for execution with our on-the-fly workload partitioning scheme.

The compiler engine preprocesses and parses the OpenCL C/C++ code and then runs a semantic analysis on the resulting abstract syntax tree. For an annotated kernel containing an irregular loop, the code generator generates the profiling code and an OpenCL kernel that supports reshaped execution (see Figures 5 and 6). The generated profiling code is compiled with a standard x86 compiler. The binaries for GPU/CPU execution of the reshaped OpenCL kernel are generated by the OpenCL compiler of the OpenCL framework such as the AMD/Intel OpenCL runtime and an x86 compiler.

6 EVALUATION

6.1 Target Architecture

The presented technique is evaluated on an AMD A10-7850K APU (Kaveri) architecture [1] and a quad-core Intel Skylake i5-6600 processor system [10]. The AMD system integrates a Streamroller-based quad-core CPU and a Graphics Core Next (GCN)-based GPU. The GCN GPU comprises eight CUs, and each CU is composed of 64 PEs (512 PEs total). The Intel system contains an Intel HD Graphics 530 GPU which comprises 24 CUs each consisting of 32 PEs (768 PEs total).

6.2 Target Workloads

A total of four OpenCL applications are evaluated with six different input data sets for a total of 24 data points for each method and architecture. The applications include the `PageRank` and `SpMV` OpenCL workloads (CSR input data format [24]) and two graph applications, `GraphColoring` and `BFS` from the GraphBIG [20] benchmark suite that were ported from CUDA to OpenCL. The GraphBIG suite provides topology-driven and data-driven implementations [21]; we evaluate the topology-driven implementation because data-driven implementation rely on atomic operations [7] between the CPU and the GPU that are not supported on the Intel platform. The work-group size is set to 64 work-items for all applications as recommended by the `clinfo` command from the AMD/Intel OpenCL runtimes.

For the input data sets, we use two real-world graph data sets, `WatsonGene` and `KnowledgeRepo`, and one synthetic data set, `LDBC` from the GraphBIG [20] benchmark suite. We further include two sparse matrices used in previous studies [18, 27], `fullchip` and `in-2004` from the University

Table 1: Input data.

Input data	VCNT	ECNT	AVG	MAX	α
Fullchip	6.0M	29.6M	5.0	2312.5K	2.6
Scale20	1.7M	32.4M	19.2	66.5K	1.7
In-2004	2.4M	18.3M	7.7	21.9K	1.9
WatsonGene	2.0M	12.2M	6.0	172.6K	2.6
LDBC	1.0M	28.8M	28.8	0.1K	86.0
KnowledgeRepo	0.14M	1.4M	10.1	9.0K	2.6

of Florida Sparse Matrix Collection [5], and `scale20` from Graph500 [19]. Table 1 lists several properties of the data sets: the number of vertices (VCNT), the number of edges (ECNT), the average edge count per vertex (i.e., the average of the inner loop iterations) (AVG), and the maximum edge count (MAX). `AVG` represents the average number of computation steps over all work-items, and `MAX` is an indicator for the irregularity of a workload. The table also contains the power-law parameter α, obtained using `plfit` [4], showing that the computational load of most input data follow a power-law distribution. The only exception is the synthetic input, `LDBC`, that exhibits a more regular workload pattern (`AVG` and `MAX` values have a smaller difference while α is higher) than the other input data sets.

6.3 Comparisons

For comparison, we use the following configurations.

- `GPU` executes the original workload on the GPU. No data copying to the GPU is necessary.
- `CPU` executes the original workload with as many threads as there are CPU cores using our threading framework (Section 5.1). Each thread is assigned the same number of work-items.
- `WGP` (WorkGroup Partitioning) assigns work-groups to the CPU and the GPU (no workload reshaping). The best results of all CPU:GPU workload partitionings from 95:5, 90:10, 85:15, …, 5:95 percent is shown.
- `Oracle` represents the oracle kernel performance that is obtained with the `Shift-H` method (Section 3.2). The best threshold value is found offline using exhaustive search. We test 99 configurations by assigning 1%, 2%, 3%, …, 99% of the work-items with the highest load to the CPU. On the CPU, we use as many threads as there are CPU cores and equally distribute the work.
- `OnTheFly`: Our on-the-fly workload partitioning technique, including all overhead.

Regarding the performance of `Oracle`, even though the best threshold value is given, profiling and reshaping must happen at runtime. The results for `Oracle` show the pure runtime (red bar) plus the overhead of profiling (white) and reshaping (grey). The state-of-the-art method, FinePar [27], uses a machine learning technique to determine the threshold value. If we assume that the model has been trained and works perfectly, the performance of FinePar is identical to that of `Oracle`. All experiments are executed three times and the average value is reported.

Oracle partitioning percentage of work-items assigned to the CPU

Platform	Fullchip	Scale20	In2004	Wats.	LDBC	Knowl.
Kaveri	2%	2%	2%	1%	30%	1%
Skylake	1%	21%	3%	10%	2%	3%

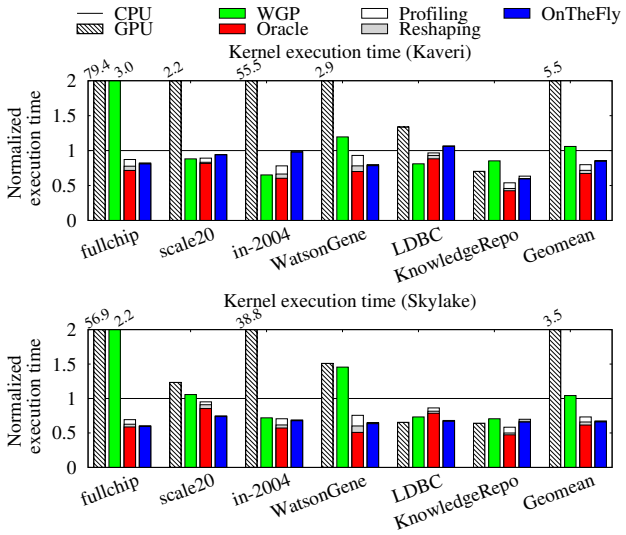

Figure 13: PageRank kernel performance.

6.4 Experimental Results

6.4.1 PageRank. In `PageRank`, a work-item contains an irregular loop that accumulates the rank values of the connected vertices to compute a new rank value.

```
1:  for (k = rowstr[id]; k < rowstr[id+1]; k++)
2:    newRank += ranks[indices[k]]
```

Figure 13 shows the execution time of the `PageRank` workload for the different workload partitioning schemes. The execution time is normalized to the `CPU` configuration. The results show that threshold-based workload reshaping, `Oracle` and our `OnTheFly` technique, can achieve a dramatic speedup compared to the original GPU execution and also outperform the CPU. For the geometric mean of the six input data sets, our `OnTheFly` technique achieves a 6.4x and 5.1x speedup on the AMD and the Intel system, respectively. Looking at the normalized performance, `OnTheFly` achieves a reduction of the execution time of 14.3% and 32.5% compared to the CPU on the AMD and the Intel system, respectively. While these workloads run faster on the CPU than the GPU, big-data workloads are typically executed on the GPU because it is often not known a priori whether an input is GPU-unfriendly.

Oracle partitioning percentage of work-items assigned to the CPU

Platform	Fullchip	Scale20	In2004	Wats.	LDBC	Knowl.
Kaveri	2%	1%	7%	7%	38%	94%
Skylake	1%	26%	10%	61%	1%	2%

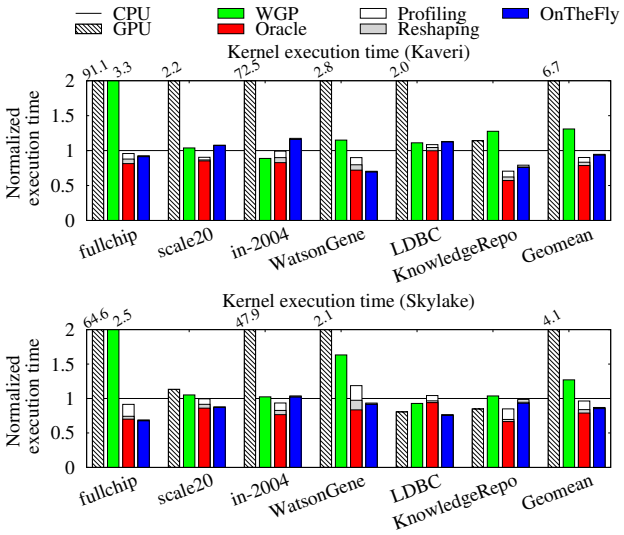

Figure 14: SpMV kernel performance.

Oracle partitioning percentage of work-items assigned to the CPU

Platform	Fullchip	Scale20	In2004	Wats.	LDBC	Knowl.
Kaveri	35%	3%	2%	7%	15%	59%
Skylake	5%	10%	1%	2%	1%	1%

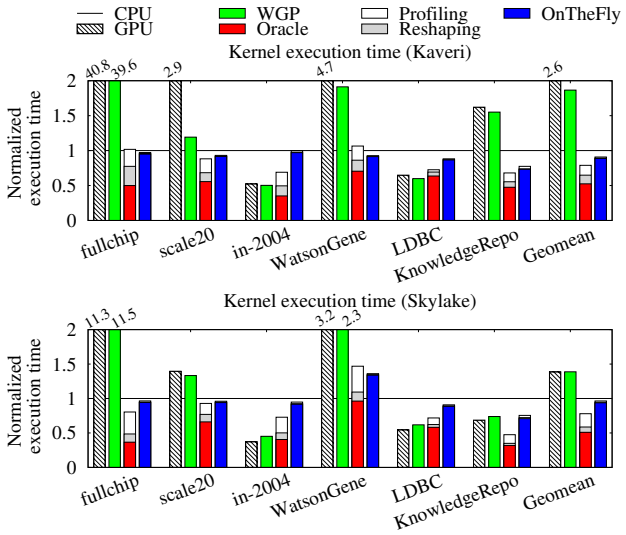

Figure 15: GraphColoring kernel performance.

The results show that the presented approach achieves a considerable speedup over the baseline, the GPU configuration, and can exploit the different characteristics of the CPU and the GPU. On the other hand, the conventional work-group partitioning approach, WGP, does not achieve consistent performance improvements by partitioning. OnTheFly achieves similar performance as Oracle when including the overhead of the (required) profiling and reshaping phases. Unlike a single kernel launch with Oracle, the runtime overhead can be hidden by our on-the-fly workload partitioning technique thanks to pipelined job execution.

We observe that performance is highly input-dependent. For example, the performance of GPU is very low for fullchip and in-2004 because these sparse matrices have a small average number of elements per row with a few rows containing a large number of elements (Table 1). As displayed in the table of Figure 13, the percentage values of the work-items assigned to the CPU show that assigning only a small number of work-items with a high loop iteration count to the CPU is beneficial for these workloads. GPU, on the other hand, shows better performance for LDBC and KnowledgeRepo. The LDBC data is relatively regular, and the work-items exhibit a similar computational load. On the Intel processor, GPU achieves the best performance for LDBC among all configurations. The execution querying the remap buffer (Figure 6) and the runtime overhead in our runtime library is responsible for this performance degradation.

6.4.2 SpMV. SpMV is another important workload used in many numerical applications. The implementation of SpMV with the CSR format is similar to that of PageRank, however, the irregular loop contains an additional memory access and multiplies the values of the sparse matrix as follows.

```
1: for (k = rowstr[id]; k < rowstr[id+1]; k++)
2:   sum += a[k] * p[colidx[k]];
```

where a[] contains the values of the sparse matrix, and p[] represents the dense vector.

Figure 14 shows that workload reshaping also achieves a large speedup for SpMV. Using the geometric mean, a speedup of 7.1x and 4.8x is achieved by OnTheFly on the AMD and the Intel machine, respectively. Compared to Oracle, OnTheFly reports a performance gap of about 10% on the AMD system and achieves higher performance on the Skylake system.

Similar to PageRank, SpMV runs poorly for the GPU-only configuration. In addition, we observe that the benefits from workload reshaping are smaller compared to PageRank. OnTheFly achieves a reduction of 5.5% and 13.0% in execution time compared to CPU. The irregular loop of SpMV accessing three independent data buffers generates a lot of memory operations. We suspect that executing a memory-heavy kernel on both the CPU and the GPU triggers congestion in the memory system that slows down that latency-sensitive CPU.

6.4.3 GraphColoring. In the GraphColoring algorithm, each vertex has a random initial value, and one kernel is executed repeatedly. In each iteration, each vertex accesses all connected vertices. If the value of the connected vertex is larger

than its own, the current iteration is marked (i.e., colored). The kernel is repeated until there is no newly colored vertex.

```
1: for (k = vlist[id]; k < vlist[id+1]; k++) {
2:   dest = elist[id]; dest_rand = randlist[id];
3:   if (vplist[dest] < color) continue;
4:   if ((dest_rand > my_rand) ||
5:     (dest_rand == my_rand && dest < id))
6:     { found_larger = 1; break; }
7: }
```

Figure 15 shows the performance of this kernel for varying input data. Our on-the-fly workload partitioning technique achieves a 2.9x and 1.4x speedup for the AMD and the Intel system, respectively. We observe that the speedup obtained from workload reshaping is smaller than that for PageRank or SpMV. The main reason is that the irregular loop has a relatively small computational load to benefit from workload partitioning. For the same reason, Oracle suffers from a large profiling and reshaping overhead because the kernel execution time is not long enough. Moreover, the irregular loop has several conditional branches in the loop. Our approach considers the number of iterations of a loop but currently does not recognize the irregularity of this workload caused by the conditional branches; we plan to consider conditional branches in future work. Nevertheless, the presented on-the-fly workload partitioning is still a viable optimization method as OnTheFly reports a remarkable speedup over the native execution GPU and provides performance improvement compared to CPU.

6.4.4 BFS. The BFS application executes a GPU kernel that marks the connected vertices in iterations until there is no additionally marked vertex. The irregular loop in the kernel is given as follows.

```
1: for (k = vlist[id]; k < vlist[id+1]; k++) {
2:   dest = elist[id];
3:    if (vplist[dest] == INITIAL)
4:      { vplist[dest]=curr+1; *changed=1; }
5: }
```

Despite the conditional branch, the speedup of Oracle and OnTheFly is significant. OnTheFly reports a 5.0x and 3.7x speedup on the AMD and the Intel system, respectively. The irregular loop contains a smaller number of operations than other evaluated algorithms; it is thus hard to achieve peak performance through workload partitioning performed at runtime. The results in Figure 16 show that both Oracle and OnTheFly suffer from a small performance degradation compared to CPU. The huge performance degradation compared to CPU for KnowledgeRepo is because of the small number of work-items and loop iterations (Table 1) for this input, and the execution time on the CPU is too short to optimize dynamically.

Oracle partitioning percentage of work-items assigned to the CPU

Platform	Fullchip	Scale20	In2004	Wats.	LDBC	Knowl.
Kaveri	2%	14%	25%	1%	54%	5%
Skylake	9%	1%	1%	2%	95%	3%

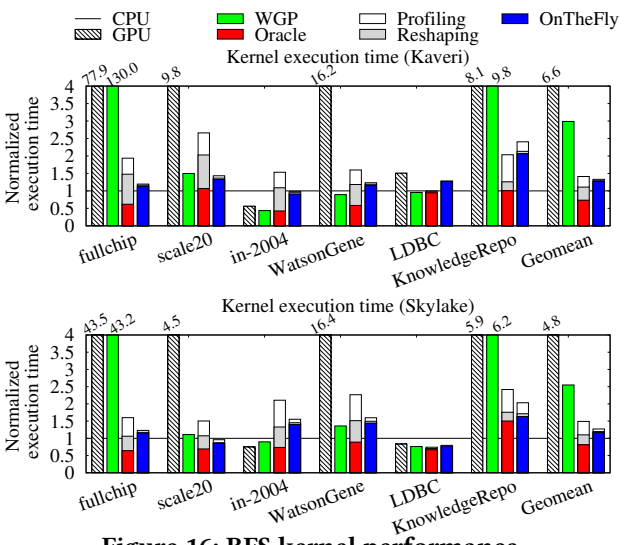

Figure 16: BFS kernel performance.

7 RELATED WORK

Workload distribution between CPUs and GPUs has recently attracted a considerable amount of research. For multiple CPU and GPU devices with dedicated memory, Lee et al. [16] perform static workload partitioning based on a performance model. Pandit et al. [23], on the other hand, distribute the workload dynamically. For integrated architectures, Kaleem et al. [11] propose dynamic technique that considers the heterogeneity of the different cores. While these techniques perform coarse-grained workload partitioning at the level of a work-group, in this work, we focus on reshaping work-items that vary in the computational load to achieve an efficient execution on integrated architectures.

There are two works that tackle fine-grained workload distribution to CPUs and GPUs. Shen et al. [25] present an optimization method for irregular applications that sorts all work-items by their computational load to avoid under-utilization of the GPU. Workloads are partitioned based on a performance model. The approach is of limited practical use as the work-items have to be reshaped before the application is started. Similar to our work, FinePar [27] performs work-item reshaping at runtime on integrated architectures. The reshaping scheme relies on extensive offline training that needs to be conducted for each application and a set of input data. In addition, workload reshaping is performed for a kernel when initializing the input data, which renders the approach impractical for iterative algorithms. In contrast, the proposed on-the-fly workload partitioning technique is hardware agnostic and gathers all required information at runtime without relying on any offline information.

8 CONCLUSIONS

New processor architectures that integrate CPUs as well as GPUs with shared memory provide interesting opportunities for optimization. This optimization of irregular workloads, however, is challenging for the runtime system because the workload pattern is highly dependent on the actual input data. We presented an optimization technique for improving the performance of irregular workloads for such integrated architectures. The on-the-fly workload partitioning technique properly partitions work-items with irregular computation demands into appropriate units for execution on the CPU and the GPU. Workload partitioning and reshaping leverages automatically generated profiling code and seamlessly adapts to different hardware architectures. An evaluation with various irregular workloads and GPU kernels shows that, on average, the on-the-fly workload partitioning provides significant speedups from 2.9x to 7.1x on an AMD APU and 1.4x to 5.1x on an Intel i5-6600 processor system.

The presented framework and the benchmarks used this study can be accessed at https://csap.snu.ac.kr/software/.

A ARTIFACT APPENDIX

A.1 Abstract

An artifact is provided to show the effectiveness and the benefits of our on-the-fly workload partitioning technique.

The artifact consists of three components.

- **Source-to-source compiler:** a source-to-source translator that performs the code generation explained in Section 3.1 for the on-the-fly workload partitioning.
- **The OpenCL library:** a dynamically loadable library that performs the on-the-fly workload partitioning explained in Section 4 by levearging the code generation.
- **Benchmark suite:** it includes the benchmark applications and some scripts that reproduce the experimental results in Section 6.

A.2 Artifact check-list (meta-information)

- **Program:** our software framework is based on C++.
- **Compilation:** gcc/g++, makefile
- **Run-time environment:** Linux with OpenCL
- **Hardware:** any integrated CPU/GPU architectures that support unified memory in OpenCL.
- **Output:** execution times in CSV and plots in EPS files.
- **Experiments:** reproduce the experiments in Section 6.
- **Workflow frameworks used?:** no
- **Publicly available?:** yes, but some parts of the framework are provided as binaries.

A.3 Description

A.3.1 How delivered. The artifact is delivered as a downloadable software package at https://csap.snu.ac.kr/software/. We also provide access to our hardware platforms (Section 6.1) used for the evaluation in the paper. To get access to the target platforms, please email bernhard@csap.snu.ac.kr.

A.3.2 Hardware dependencies. The artifact assumes integrated CPU/GPU architectures where the CPU and the GPU share the same off-chip DRAM.

A.3.3 Software dependencies. Our OpenCL library assumes a Linux system where a fully-functional OpenCL (version 1.2 or above) framework is installed and support unified memory for OpenCL programming, as explained in Section 2.1.

A.3.4 Data sets. Users download the data sets in Table 1 from our webserver through a script in the artifact.

A.4 Installation

After downloading the artifact, install our OpenCL library and the benchmark applications using the build scripts in the artifact. The detailed installation steps are provided in the README.md file in the artifact.

A.5 Experiment workflow

The steps to reproduce the results in Section 6 are provided in the README.md file.

A.6 Evaluation and expected result

The experiments from Section A.5 cover all experimental scenarios in Section 6.

A.7 Experiment customization

Experiment customization (e.g. using different data sets and different OpenCL applications) is possible. The README.md file contains information to use our OpenCL framework for arbitrary OpenCL applications.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for the helpful feedback. This work was supported, in part, by the National Research Foundation of Korea (NRF) funded by the Korea government (MSIT) (grant NRF-2015K1A3A1A14021288), BK21 Plus for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU) funded by the NRF (grant 21A20151113068), the Swiss National Science Foundation (SNF) grant IZKSZ2_162084 as part of the Korean-Swiss Science and Technology Programme (KSSTP) and by SNF R'Equip grant 206021_133835, and by the Promising-Pioneering Researcher Program through Seoul National University in 2015. ICT at Seoul National University provided research facilities for this study.

REFERENCES

[1] Dan Bouvier and Ben Sander. 2014. Applying AMD's Kaveri APU for heterogeneous computing. In *2014 IEEE Hot Chips 26 Symposium (HCS), Cupertino, CA, USA, August 10-12, 2014*. IEEE, 1–42. https://doi.org/10.1109/HOTCHIPS.2014.7478810

[2] Alexander Branover, Denis Foley, and Maurice Steinman. 2012. AMD Fusion APU: Llano. *IEEE Micro* 32, 2 (March 2012), 28–37. https://doi.org/10.1109/MM.2012.2

[3] Sergey Brin and Lawrence Page. 2012. Reprint of: The anatomy of a large-scale hypertextual web search engine. *Computer Networks* 56, 18 (2012), 3825 – 3833. https://doi.org/10.1016/j.comnet.2012.10.007

[4] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. 2009. Power-Law Distributions in Empirical Data. *SIAM Rev.* 51, 4 (2009), 661–703. https://doi.org/10.1137/070710111 arXiv:https://doi.org/10.1137/070710111

[5] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[6] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. 2017. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *IEEE Micro* 37, 2 (Mar.-Apr. 2017), 52–62. https://doi.org/10.1109/MM.2017.38

[7] Juan Gómez-Luna, Izzat El Hajj, Li-Wen Chang, Victor García-Floreszx, Simon Garcia de Gonzalo, Thomas B. Jablin, Antonio J. Peña, and Wen-mei Hwu. 2017. Chai: Collaborative heterogeneous applications for integrated-architectures. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 43–54. https://doi.org/10.1109/ISPASS.2017.7975269

[8] Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R. Gaster, and Bixia Zheng. 2010. Twin Peaks: A Software Platform for Heterogeneous Computing on General-purpose and Graphics Processors. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*. ACM, New York, NY, USA, 205–216. https://doi.org/10.1145/1854273.1854302

[9] Thilina Gunarathne, Bimalee Salpitikorala, Arun Chauhan, and Geoffrey Fox. 2011. Optimizing OpenCL Kernels for Iterative Statistical Applications on GPUs. In *Proceedings of the 2nd International Workshop on GPUs and Scientific Applications (GPUScA '11)*. University of Vienna, 33–44. http://www.par.univie.ac.at/publications/download/TR-11-1.pdf

[10] Intel. 2015. Intel Core i5-6600 Processor. http://ark.intel.com/products/88188/Intel-Core-i5-6600-Processor-6M-Cache-up-to-3_90-GHz. (2015). [online; accessed August 2018].

[11] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshav Pingali. 2014. Adaptive Heterogeneous Scheduling for Integrated GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/2628071.2628088

[12] Khronos Group. 2009. The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/. (2009). [online; accessed August 2018].

[13] Hee-Seok Kim, Izzat El Hajj, John Stratton, Steven Lumetta, and Wen-Mei Hwu. 2015. Locality-centric thread scheduling for bulk-synchronous programming models on CPU architectures. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 257–268. https://doi.org/10.1109/CGO.2015.7054205

[14] Junghyun Kim, Thanh Tuan Dao, Jaehoon Jung, Jinyoung Joo, and Jaejin Lee. 2015. Bridging OpenCL and CUDA: a comparative analysis and translation. In *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2807591.2807621

[15] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. 2012. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. ACM, New York, NY, USA, 341–352. https://doi.org/10.1145/2304576.2304623

[16] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. 2015. SKMD: Single Kernel on Multiple Devices for Transparent CPU-GPU Collaboration. *ACM Trans. Comput. Syst.* 33, 3, Article 9 (Aug. 2015), 27 pages. https://doi.org/10.1145/2798725

[17] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28, 2 (March 2008), 39–55. https://doi.org/10.1109/MM.2008.31

[18] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*. ACM, New York, NY, USA, 339–350. https://doi.org/10.1145/2751205.2751209

[19] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* (2010).

[20] Lifeng Nai, Yinglong Xia, llie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: understanding graph computing in the context of industrial solutions. In *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2807591.2807626

[21] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Data-Driven Versus Topology-driven Irregular Computations on GPUs. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 463–474. https://doi.org/10.1109/IPDPS.2013.28

[22] CUDA Nvidia. 2007. Compute unified device architecture programming guide. (2007).

[23] Prasanna Pandit and R. Govindarajan. 2014. Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 273, 11 pages. https://doi.org/10.1145/2544137.2544163

[24] Youcef Saad. 1990. SPARSKIT: A basic tool kit for sparse matrix computations. https://ntrs.nasa.gov/search.jsp?R=19910023551. (1990). [online; accessed August 2018].

[25] Jie Shen, Ana Lucia Varbanescu, Peng Zou, Yutong Lu, and Henk Sips. 2014. Improving Performance by Matching Imbalanced Workloads with Heterogeneous Platforms. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS '14)*. ACM, New York, NY, USA, 241–250. https://doi.org/10.1145/2597652.2597675

[26] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter Mccardwell, Alejandro Villegas, and David Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10. https://doi.org/10.1109/IISWC.2016.7581262

[27] Feng Zhang, Bo Wu, Jidong Zhai, Bingsheng He, and Wenguang Chen. 2017. FinePar: Irregularity-aware Fine-grained Workload Partitioning on Integrated Architectures. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, Piscataway, NJ, USA, 27–38. http://dl.acm.org/citation.cfm?id=3049832.3049836

[28] Feng Zhang, Jidong Zhai, Bingsheng He, Shuhao Zhang, and Wenguang Chen. 2017. Understanding Co-Running Behaviors on Integrated CPU/GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems* 28, 3 (March 2017), 905–918. https://doi.org/10.1109/TPDS.2016.2586074