Co-Scheduling on Fused CPU-GPU Architectures with Shared Last Level Caches

Marvin Damschen, *Member, IEEE*, Frank Mueller, *Fellow, IEEE*, Jörg Henkel, *Fellow, IEEE*

***Abstract*—Fused CPU-GPU architectures integrate a CPU and general-purpose GPU on a single die. Recent fused architectures even share the last level cache (LLC) between CPU and GPU. This enables hardware-supported byte-level coherency. Thus, CPU and GPU can execute computational kernels collaboratively, but novel methods to co-schedule work are required. This paper contributes three dynamic co-scheduling methods. Two of our methods implement workers that autonomously acquire work from a common set of independent work items (similar to bag-of-tasks scheduling). The third method, *host-side profiling*, uses a fraction of the total work of a kernel to determine a ratio of how to distribute work to CPU and GPU based on profiling. The resulting ratio is used for following executions of the same kernel.**

Our methods are realized using OpenCL 2.0, which introduces fine-grained Shared Virtual Memory (SVM) to allocate coherent memory between CPU and GPU. We port the Rodinia Benchmark Suite, a standard suite for heterogeneous computing, to fine-grained SVM and fused CPU-GPU architectures (*Rodinia-SVM*). We evaluate the overhead of fine-grained SVM and analyze the suitability of OpenCL 2.0's new features for co-scheduling. Our host-side profiling method performs competitively to the optimal choice of executing kernels *either* on CPU *or* GPU (hypothetical *xor-Oracle*). On average, it achieves 97% of xor-Oracle's performance and a $1.43\times$ speedup over using the GPU alone (standard in Rodinia). We show, however, that in most cases it is not beneficial to split the work of a kernel between CPU and GPU compared to exclusively running it on the most suitable single compute device. For a fixed amount of work per device, cache-related stalls can increase by up to $1.75\times$ when both devices are used in parallel instead of exclusively while cache misses remain the same. Thus, not the cost of cache conflicts, but inefficient cache coherence is a major performance bottleneck for current fused CPU-GPU Intel architectures with shared LLC.

***Index Terms*—Heterogeneous computing, integrated architecture, scheduling, performance tuning**

I. Introduction

With the release of AMD's Fusion and Intel's Ivy Bridge architecture in 2011, the trend of processor integration resulted in *fused CPU-GPU architectures* that integrate a CPU and general-purpose GPU on a single die. The main benefit of such an integration is that time-consuming memory transfers between main memory and dedicated GPU memory become unnecessary. Instead, CPU and GPU access the same physical memory such that *zero-copy* transfers can be employed. Zero-copy transfers ensure coherency and translate pointers to memory buffers for the common CPU and GPU address space, but do not actually transfer data. However, such an integration introduces a memory bottleneck, because CPU and GPU compete for memory bandwidth of the shared physical memory.

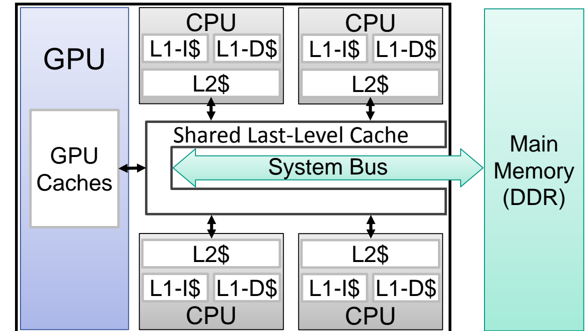

Fig. 1: High-level overview of a fused CPU-GPU architecture with shared last level cache

In more recent architectures, e.g., Intel Broadwell and beyond, CPU and GPU were further integrated so that they access the shared last level cache (LLC) as shown in Fig. 1. This enables hardware-supported byte-level cache coherency between CPU and GPU. Effectively, CPU and GPU can execute computational kernels on the same data in parallel and solve problems collaboratively. In this case, the shared LLC has the potential to alleviate the memory bottleneck present in earlier fused CPU-GPU architectures (without a shared LLC), because it can serve accesses to a common working set instead of requiring frequent main memory accesses [1].

The idea of heterogeneous compute devices performing computations on a common memory is also captured in the Open Compute Language (OpenCL) standard 2.0. Most prominently, OpenCL introduces *Shared Virtual Memory* (SVM), i.e., a shared virtual address space between heterogeneous compute devices in an OpenCL program. SVM is also supported by fused CPU-GPU architectures without a shared LLC, e.g., AMD's Accelerated Processing Units or System on Chips that feature ARM's Mali Bifrost GPU. However, because excessive coherency traffic is required across heterogeneous devices [2], SVM was proven inefficient on such architectures.

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Invasive Computing" (SFB/TR 89).

This work was supported in part by NSF grants 1329780, 1239246, and 1525609.

Marvin Damschen and Jörg Henkel are with the Chair for Embedded Systems, Karlsruhe Institute of Technology, Karlsruhe 76131, Germany (e-mail: damschen@kit.edu; henkel@kit.edu)

Frank Mueller is with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206, USA (e-mail: mueller@cs.ncsu.edu)

This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2018 and appears as part of the ESWEEK-TCAD special issue

In contrast, on fused CPU-GPU architectures with a shared LLC, OpenCL 2.0 promises efficient support for byte-level coherent (so-called *fine-grained*) SVM as well as cross-device atomics [3].

This work presents the first investigation of *collaborative execution* of computational kernels on a fused CPU-GPU architecture with a shared LLC using fine-grained SVM, i.e., CPU and GPU share cache-coherent memory so that the work of a computational kernel can be processed in parallel by both compute devices. We detail how OpenCL programs are ported to OpenCL 2.0's fine-grained SVM. This process is applied to the entire Rodinia Benchmark Suite [4] and overheads of fine-grained SVM are evaluated. Collaborative execution of computational kernels on fine-grained SVM requires novel co-scheduling approaches that determine how much work should be performed on CPU and GPU, respectively, for maximum performance. In previous studies on collaborative execution that used zero-copy transfers on fused CPU-GPU architectures with a shared LLC and OpenCL 1.2, a single static data-centric distribution of work was established for all kernels per program [5, 6]. Fine-grained SVM enables to decide the distribution of work dynamically, based on observed progress made by CPU and GPU while executing a kernel. Thus, a decision should be made per kernel instead of per program.

This work contributes three dynamic co-scheduling approaches that utilize different capabilities of OpenCL 2.0: one kernel-external method based on online profiling and two kernel-internal methods that utilize *cross-device atomics* (variables that can be modified atomically across multiple compute devices). Cross-device atomics are currently supported by OpenCL 2.0 only, apart from that our approaches could also be realized, e.g., in NVIDIA CUDA. One of the kernel-internal methods utilizes *device-side enqueuing*, another feature introduced with OpenCL 2.0 that enables enqueuing kernels to an OpenCL device from within an executing kernel. Device-side enqueuing is a similar technique to dynamic parallelism in NVIDIA CUDA. However, it is shown that device-side enqueuing introduces too much overhead to be suitable for implementing co-scheduling approaches. The other two co-scheduling approaches (one kernel-external and one kernel-internal) are further evaluated using the Rodinia Benchmark Suite, which we ported to OpenCL 2.0. Our kernel-external method performs competitively to the optimal choice of executing kernels within a program either on CPU or GPU (clairvoyant *xor-Oracle*, some kernels on CPU others on GPU within the same program). The method achieves 97% of the xor-Oracle's performance on average. We show, however, that for most benchmarks of the Rodinia Benchmark Suite it is not beneficial to split the work of a kernel between CPU and GPU compared to running a kernel either on CPU or GPU when fine-grained SVM is used. This observation is further analyzed and it is shown that it cannot be explained by cache conflicts, i.e., false or true sharing, but is the result of inefficient cache coherence. As of today, Intel platforms are the only architectures that support OpenCL 2.0's fine-grained SVM using a shared LLC. Therefore, we focus on this architecture in the remainder of the paper.

Our novel contributions are as follows:

- We evaluate the overhead of OpenCL 2.0's fine-grained Shared Virtual Memory, and analyze the suitability of cross-device atomics as well as device-side enqueuing for co-scheduling kernels on fused CPU-GPU architectures with a shared LLC in three different co-scheduling approaches.
- We develop a co-scheduling approach that is competitive to the optimal choice of executing kernels within a program either on CPU or GPU (on average 97% of the clairvoyant xor-Oracle's performance and $1.43\times$ speedup over only using the GPU), and via analysis show that inefficient cache coherence is the major performance bottleneck for collaborative execution of the same kernel on current fused CPU-GPU architectures with shared LLC.
- We port the Rodinia Benchmark Suite to OpenCL 2.0 with fine-grained SVM and make Rodinia-SVM as well as a variety of co-scheduling approaches available as open source[1].

II. Related Work

A. Co-Scheduling on Fused Architectures

In state-of-the-art related work on co-scheduling for fused CPU-GPU architectures, CPU and GPU do not share the last level cache [5, 6, 7, 8, 9, 10]. Thus, techniques like fine-grained SVM are not supported and communication between CPU and GPU has to rely on explicit data transfers. [7] presents an online profiling-based approach that is similar to our host-side profiling approach, but only treats the GPU as an OpenCL 1.2 device while CPU computations are performed in the host code. Therefore, barriers are required after every kernel run, whereas our approach treats CPU and GPU as OpenCL 2.0 devices and utilizes OpenCL events for lightweight synchronization (see Section V-B). Data transfer overheads between different devices are mentioned as a key issue, but not further analyzed. [9] uses the online profiling method of [7] and presents a power-aware co-scheduling method that aims to minimize the energy-delay product of heterogeneous applications running on a fused CPU-GPU architecture. The authors report an average of 12.3% percent improvement over the best performance-oriented schedules. [5] presents an offline, machine learning-based approach to co-scheduling that determines a single ratio that partitions the input data into separate parts processed by CPU and GPU, respectively. This saves additional transfers to maintain coherency between kernel executions, but does not allow for per-kernel decisions. [8] presents an OpenCL runtime system that automatically schedules kernels to multiple devices that were originally written for a single device. The runtime system takes care of buffer allocation and transfers to maintain coherency between all devices without programmer effort. [10] and [6] specifically target irregular workloads, in which some work items take considerably longer than others such that profiling information from a subset of work items is often not representative for the performance of the whole kernel. Both

[1] Source code available at: https://git.scc.kit.edu/CES/Rodinia-SVM

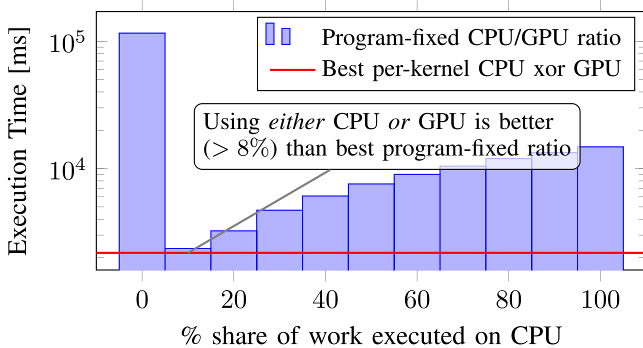

Fig. 2: Particle Filter benefits from a per-kernel scheduling decision compared to a fixed ratio for the whole benchmark when executed on OpenCL 2.0's fine-grained SVM

approaches identify application-specific features to model the computational kernels' performance for scheduling decisions.

Compared to our work, state-of-the-art co-scheduling approaches did not share cache-coherent memory between CPU and GPU, but were instead limited by explicit data transfers that were required to establish consistency.

B. Exploiting Shared Virtual Memory

In [1] the potential of fused CPU-GPU architectures with a shared LLC is explored simulatively. The authors present an approach where compiler-generated "pre-execution code" is run on the CPU, before executing a computational kernel on the GPU. The aim of this approach is to fill the shared LLC such that the amount of main memory accesses that need to be performed by the GPU is minimized. Using this approach, the authors report a performance improvement of up to 113%, and 21.4% on average. [11] presents an extension of the gem5-gpu simulator for fused CPU-GPU architectures [12] that supports the features of OpenCL 2.0. Compared to these works, our approach utilizes a commercial off-the-shelf architecture (Intel) instead of simulation.

In [2] a comprehensive performance evaluation of OpenCL 1.2, OpenCL 2.0 and Heterogeneous System Architecture (HSA) 1.0 is presented. In contrast to our work, the evaluated AMD Kaveri architecture does not feature a shared LLC between CPU and GPU. As a result, the authors observe that excessive coherency traffic is generated across devices that can affect performance significantly.

In summary, state-of-the-art related work on co-scheduling on fused CPU-GPU architectures either failed to leverage cache-coherent memory between CPU and GPU or only explored cache coherency between CPU and GPU in simulation.

III. Motivational Example

Before the introduction of OpenCL 2.0's fine-grained SVM, data needed to be explicitly transferred to compute devices. Furthermore, consistency guarantees for memory buffers that were accessed in parallel by different compute devices did not exist. Therefore, state-of-the-art co-scheduling approaches divided the input data into two separate parts that were processed by CPU and GPU, respectively [5, 6]. Effectively, a single

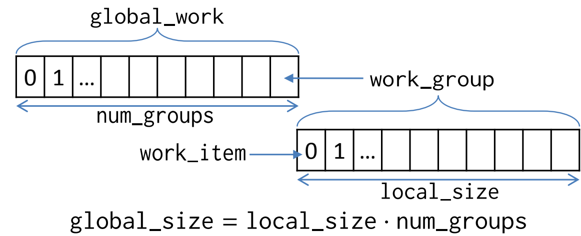

Fig. 3: Hierarchy of Work Items in an OpenCL Kernel

ratio that determines the share of work to be performed on each compute device was applied to all kernels of an OpenCL program. With fine-grained SVM pointers can be shared and accessed consistently by multiple devices in parallel.

Fig. 2 shows execution time results for the *Particle Filter* benchmark from the Rodinia Benchmark Suite (version 3.1 ported to fine-grained SVM) on an Intel Core i7-6700T (Skylake) fused CPU-GPU architecture with a shared LLC. The blue bars show the execution time for statically-fixed ratios of work performed on CPU and GPU, respectively, that are applied to all four kernels of the benchmark. The red line shows the execution time for deciding *per-kernel* whether to execute it *either* on CPU *or* on GPU. Only the single best overall decision (first two kernels on GPU, remaining two on CPU) is shown. In any case, the four kernels need to be executed in sequence. Two of four kernels contain loops that result in extremely poor performance when executed on the GPU only (thus, the execution time drops from $x = 0\%$ to $x = 10\%$), while the other two kernels benefit strongly from execution on the GPU compared to the CPU (thus, the execution time increases from $x = 10\%$ to $x = 100\%$). Therefore, deciding a single ratio of how to distribute work for all kernels results in a compromise that performs worse than executing each kernel exclusively on the most suitable device. Due to the fact that fine-grained SVM is shared consistently among different compute devices without any explicit data transfers in between kernel executions.

This example shows that per-kernel decisions of how to distribute work have a performance benefit over a single data-centric ratio that is applied to all kernels of a program. In this work, we explore co-scheduling methods that leverage OpenCL 2.0 features to perform per-kernel decisions at run-time beyond the binary decision of either using the CPU or GPU but by utilizing both compute devices in parallel.

IV. Background on Heterogeneous Execution

In this section we provide an overview of OpenCL in general and discuss features introduced in OpenCL 2.0 that we utilize for co-scheduling.

A. OpenCL

The Open Compute Language (OpenCL) is an open standard for parallel programming of heterogeneous systems [13]. It consists of a host-side API and a C-like programming language for writing computational *kernels*. The host-side API provides access to the *platform*, i.e., a view of the system that

```
1  int* ptr = (int*)malloc(...);
2  for (int i=0; i<n; i++)
3    ptr[i] = i;
4  ptr_device = clCreateBuffer(...);
5  clEnqueueWriteBuffer(ptr_device, ptr,...);
6  clSetKernelArg(...,ptr_device);
7  clEnqueueNDRange(...);
8  clEnqueueReadBuffer(ptr_device, ptr,...);
9  clFinish(...);
10 printf("Result: %d\n", ptr[0]);
```

```
1  int* ptr = (int*)clSVMAlloc(...);
2  for (int i=0; i<n; i++)
3    ptr[i] = i;
4
5
6  clSetKernelArgSVMPointer(...,ptr);
7  clEnqueueNDRange(...);
8
9  clFinish(...);
10 printf("Result: %d\n", ptr[0]);
```

Fig. 4: Simplified example of memory allocation in OpenCL 1.2 (left) and OpenCL 2.0 with fine-grained SVM (right)

the OpenCL program is executed on. The platform comprises one or more *devices* that are capable of executing OpenCL kernels. Within fused CPU-GPU architectures, CPU (including all cores) and GPU are separate devices[2] belonging to the same platform. For communication between host and devices, the host-side API provides functions to submit *commands* to *command queues*. Commands specify tasks that should be performed by a device, e.g., memory operations, synchronization or kernel execution. Each command queue is associated with exactly one device. *Events* can be used to formulate dependencies between commands (from the same or different command queues) as directed acyclic graphs. A command can emit an event upon successful execution. When submitting a command to a command queue, it can be specified that the command should only be executed after one or more events were emitted by finishing the execution of respective commands.

Generally, when implementing an OpenCL kernel, the goal is to represent parallelism at the finest possible granularity. Figure 3 shows how OpenCL divides work hierarchically as well as OpenCL keywords used by the host-side API[3]. The smallest unit of execution is a *work item*. Each work item executes an instance of the kernel body, e.g., for a kernel that implements vector addition a work item would compute a single element. When submitting a kernel to a command queue, usually thousands of work items are instantiated and execute concurrently (as many as given by `global_size`). Work items are divided into *work groups*. Work groups are equally-sized (by `local_size`) and each group has a unique `group_id`. Work items have a `local_id` (0, ..., `local_size` − 1) that is unique within a work group only, as well as a globally unique `global_id` (0, ..., `global_size` − 1) that is used for address calculations. The `global_id` specifies on which part of the input data a specific work item executes the kernel body on. Only within a work group can work items perform barrier operations and share *local memory*. This way, the OpenCL compiler can perform device-specific optimizations, e.g., on CPUs a work group is serialized to a single thread.

[2]Note that commonly used terms like 'compute unit' or 'processing element' are defined as specific parts of a device in OpenCL

[3]OpenCL supports up to three-dimensional index spaces. At this point, we explain the one-dimensional case for brevity

B. OpenCL 2.0

The OpenCL specification 2.0 introduced several features that provide opportunities for improved collaboration between different devices as well as the host [13, 14]. The most prominent feature is Shared Virtual Memory (SVM) that introduces a shared virtual address space between host and devices in an OpenCL program. SVM eliminates explicit data transfers between host and device memory, and enables direct sharing of pointer-based data structures. OpenCL 2.0 introduces *coarse-grained* and *fine-grained* SVM. Coarse-grained SVM allows host and devices to share virtual memory pointers, but still requires buffers that are explicitly mapped and unmapped from host and devices. A coarse-grained SVM buffer can only be mapped to a single device or the host at a time, concurrent accesses by multiple devices are not supported. Fine-grained SVM is an optional feature of OpenCL 2.0 that defines memory consistency guarantees for SVM allocations that are concurrently accessed by the host and one or more devices. With fine-grained SVM, host and devices can share memory at byte-level granularity and read from it concurrently. Concurrent writes are supported to non-overlapping bytes. Consistency is guaranteed before and after each command execution. When more fine-grained consistency is required, *atomics* can be used.

Atomics are another optional feature introduced by OpenCL 2.0. In combination with fine-grained SVM, atomics can be shared between different devices. This enables cross-device atomic operations and additionally provides a means of synchronization. This way, byte-level consistency can be guaranteed within a kernel.

Before OpenCL 2.0, the only way to execute commands on a device was to submit commands to a command queue using the host-side API. This means that the number of work items that should be executed when launching a kernel needed to be known before the kernel was executed. OpenCL 2.0 introduces *device-side enqueuing*, i.e., kernels get the ability to enqueue child kernels in a device-side command queue. Similarly to dynamic parallelism in NVIDIA CUDA, this enables implementation of kernels that perform calculations iteratively or use recursion. Like in host-side enqueuing, dependencies between child kernels can be specified using events, but generated events are only visible to the parent kernel. Child kernels run asynchronously to the parent kernel. However, the parent kernel is only registered as successfully

```
1 clEnqueueNDRangeKernelFused(commandsCPU,
     commandsGPU, kernel,...) {
2   // ... (calculate work item shares and IDs)
3   if(workItemsCPU>0) // work assigned to CPU?
4     clEnqueueNDRangeKernel(commandsCPU,
          kernel, ..., &eventGPUDone[curr-1],
          &eventCPUDone[curr]);
5   else
6     clSetUserEventStatus(eventCPUdone[curr],
          CL_COMPLETE);
7   if(workItemsGPU>0) // work assigned to GPU?
8     clEnqueueNDRangeKernel(commandsGPU,
          kernel, ..., &eventCPUDone[curr-1],
          &eventGPUDone[curr]);
9   else
10    clSetUserEventStatus(eventGPUdone[curr],
          CL_COMPLETE);}
```

Fig. 5: Launching a kernel on a fused CPU-GPU architecture without host-side synchronization

TABLE I: Rodinia Benchmark Suite – OpenCL Benchmarks

Name	Abbreviation	#Kernels
Back Propagation	bp	2
Breadth-First Search	bfs	2
B+Tree	b+	2
CFD Solver	cfd	4
GPUDWT	dwt	3
Gaussian Elimination	ge	2
Heart Wall	hw	1
HotSpot3D	hs3D	1
HotSpot	hs	1
Hybrid Sort	hys	7
K-Means	km	2
LavaMD	md	1
Leukocyte Tracking	lc	3
LU Decomposition	lud	4
Myocyte	mc	1
Nearest Neighbor	nn	1
Needleman-Wunsch	nw	2
Particle Filter	prtf	4
Path Finder	pthf	1
Streamcluster	sc	1

executed (and may emit an event), when all its child kernels finished execution.

V. Utilizing fine-grained SVM on fused CPU-GPU Architectures

A. Memory Allocation

Until OpenCL 2.0, communication between the host program and compute devices required explicit allocation of device-side buffers. As shown in the simplified example in Fig. 4 (l.4, left), memory that is allocated and initialized by the host program needs to be transferred to the device-side buffer first (l.5), before a kernel can be launched using `clEnqueueNDRange(...)`. After kernel execution finishes, the results are transferred back (l.8).

In Rodinia-SVM, we removed all device-side buffer allocations from the original Rodinia Benchmark Suite and utilize fine-grained SVM instead, as shown in Fig. 4 (right). This allows all devices and the host to access memory using shared pointers. As a result, all explicit transfers between host and devices are eliminated. Furthermore, while device-side buffers are owned by a single device at a time, fine-grained SVM can be accessed consistently by multiple devices and the host.

B. Kernel Launch and Synchronization

For launching kernels on a fused CPU-GPU architecture, one command queue is instantiated for each device (CPU and GPU). Then, the same kernel is enqueued with only a share of the total work items (`global_size`, see Fig. 3) plus offsets that are used for calculating global work item IDs. ID calculation depends on the specific co-scheduling method, and is therefore detailed in Section VI.

Earlier versions of OpenCL required explicit synchronization at the host-side, e.g., using `clFinish(...)` (see Fig. 4) or `clWaitForEvents(...)`, to achieve consistency [5]. Synchronization with the host induces a significant overhead, however, because the devices' command queues can no longer be processed in parallel. Because fine-grained SVM maintains consistency, host-side synchronization is not required anymore. However, we still need to ensure that CPU and GPU execute kernels in lock step, i.e., when launching a sequence of kernels like `clEnqueueNDRange(kernelA, ...); clEnqueueNDRange(kernelB,...);`, the CPU should not begin executing `kernelB` before the GPU finished executing `kernelA` and vice versa. Otherwise, results from `kernelA` that `kernelB` depends on might not be ready when one device races ahead. This can lead to erroneous results. As shown in Fig. 5, we utilize events to express these dependencies. For each device a ring buffer is allocated that stores one event for each enqueued kernel (`eventCPUDone` and `eventGPUDone`, respectively). If work items are assigned to the CPU, the kernel is enqueued on the CPU (l.4). The execution of the kernel depends on an event that is emitted when the previous kernel that was enqueued to the GPU completes execution (`eventGPUDone[curr-1]`). In case no work items were assigned to the CPU, the event that indicates completed execution of the current kernel launch on the CPU is emitted immediately so that no deadlocks occur (`eventCPUDone[curr]`, l.6). Kernel launches on the GPU are performed analogously. In Rodinia-SVM, we replaced all calls to `clEnqueueNDRange(...)` with our `clEnqueueNDRangeFused(...)` implementation. Furthermore, the co-scheduling methods that we detail in Section VI are also applied by `clEnqueueNDRangeFused(...)`.

C. Overheads of Fine-Grained SVM

Figure 6 shows execution time results for all benchmarks of the Rodinia Benchmark Suite (version 3.1, listed in Table I) in two variants: the original OpenCL 1.2 version as well as our OpenCL 2.0 port where all device-side buffers were replaced by fine-grained SVM allocations as explained in Section V-A. In both variants, kernels are executed on the GPU only (the fused kernel launch of Section V-B is not used) on a Intel Core i7-6700T (Skylake). Kernel compilation times

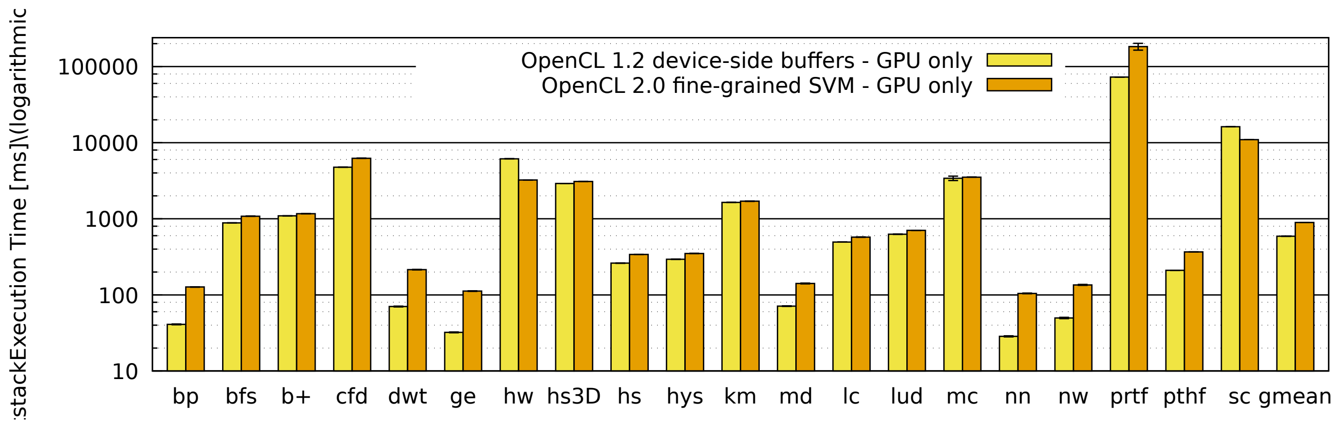

Fig. 6: Compared to the original OpenCL 1.2 implementation of the Rodinia Benchmarks Suite that executes on the GPU only and uses device-side buffers, the use of OpenCL 2.0 incl. fine-grained SVM introduces overheads but maintains consistency

are omitted[4]. The results show that the convenience of being able to pass host-side pointers directly into kernels comes at a cost. In particular, short-running benchmarks (100ms and less) are significantly slowed down, e.g., `ge` takes almost 3.5× longer (112ms instead of 32ms) when executed on fine-grained SVM instead of device-side buffers. Benchmarks that run 100ms or more in the OpenCL 1.2 version only take 1.14× longer on average (geometric mean). Longer-running benchmarks that alternate between kernel execution and host-side computations like `hw` and `sc` even benefit from fine-grained SVM (1.9× and 1.48× speedup, respectively), because with OpenCL 1.2 they explicitly need to synchronize with the host and invoke transfers after every kernel execution. However, the geometric mean execution time increase over all benchmarks for the OpenCL 2.0 versions compared to the OpenCL 1.2 version is 1.51×. The overheads stem from the fact that the OpenCL 1.2 device-side buffers used in the Rodinia benchmarks are already allocated as *zero copy* buffers on fused CPU-GPU architectures[5], i.e., instead of allocating separate host-side and device-side memory, both buffers are mapped to the same shared physical memory. Consequently, data transfers between host-side and device-side buffers do not actually transfer data, but only translate pointers and initiate the OpenCL 1.2 runtime system to establish consistency between CPU and GPU. OpenCL 2.0's fine-grained SVM adds overhead compared to zero copy buffers, because consistency is not only established explicitly using transfers (e.g., at the beginning and end of a computation often consisting of multiple enqueued kernels), but continuously when kernels are executed.

Ultimately, these overheads have to be considered when implementing OpenCL 2.0 programs to decide whether to use fine-grained SVM or not. However, fine-grained SVM does not only provide the convenience of shared pointers, but also enables new features like cross-device atomics. In the following we will present co-scheduling methods that exploit these new features and evaluate them using Rodinia-SVM.

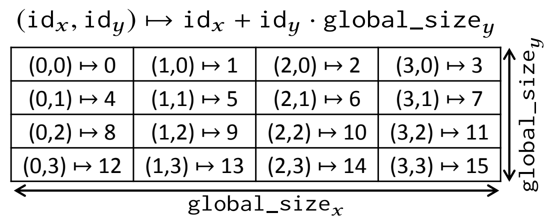

Fig. 7: For co-scheduling, multi-dimensional IDs are mapped to one-dimensional IDs

```
1 typedef struct global_work_state_struct {
2   atomic_uint workDone;
3   size_t globalWork;
4 } global_work_state;
```

Fig. 8: A `global_work_state` is shared between work items using fine-grained SVM to realize device-side scheduling

VI. Our Co-Scheduling Methods

Let us define two types of co-scheduling methods, namely (1) *device-side co-scheduling*, where work group scheduling is performed during execution of the respective kernel by the executing devices themselves, and (2) *host-side co-scheduling*, where work groups are assigned to CPU and GPU using the host-side OpenCL API only (outside of the kernels). In the

```
1 __kernel void kernel(...) {
2   PREAMBLE
3   ... // --- original kernel code ---
4   POSTAMBLE }
```

Fig. 9: The device-side methods add a preamble and postamble to each kernel that implement the co-scheduling methods

[4]Kernel compilation can be avoided using `clCreateProgramWithBinary(...)`

[5]using `CL_MEM_USE_HOST_PTR` or `CL_MEM_ALLOC_HOST_PTR` flags

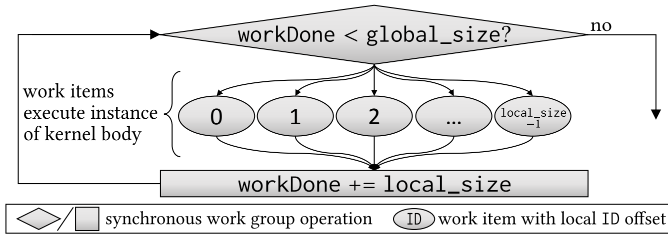

Fig. 10: A single work group executes in lock step (*atomic counting*). Multiple work groups execute in parallel

following we will present two device-side and one host-side co-scheduling methods. All methods leverage OpenCL 2.0's fine-grained SVM to achieve consistency while executing kernels on CPU and GPU in parallel.

When enqueuing an OpenCL kernel using the OpenCL host API function `clEnqueueNDRangeKernel(...)`, the `global_size` parameter specifies how many work items should be launched by the OpenCL runtime system on a specific device (see Fig. 3). `global_size` can be given as an up to three-dimensional array. In this case, work items are assigned a global ID for each dimension by the OpenCL runtime system. For co-scheduling, we project multi-dimensional kernel IDs onto one-dimensional IDs. An example for the two-dimensional case is given in Fig. 7.

In our co-scheduling methods, we launch a subset of the total work items on CPU and GPU and then proceed to schedule the remaining work items based on the observed performance. The main idea behind the device-side methods is to treat the work of a kernel as a bag-of-tasks that contains independent work groups. Initially, only a few work groups are launched (enough to fully utilize CPU and GPU). The work items of these work groups act as workers that autonomously acquire and process work from the bag-of-tasks. To implement this scheme, the device-side methods utilize a `global_work_state` struct that is stored in SVM and shared between CPU and GPU (see Fig. 8). `globalWork` is the total amount of times the body of an enqueued kernel needs to be executed (equal to the `global_size` parameter passed to `clEnqueueNDRangeKernelFused(...)`). In all methods, the kernel is executed `globalWork` times in total. `workDone` is an atomic counter that keeps track of how many work items were executed. It is used to calculate work item IDs and to decide whether another work group needs to be scheduled, i.e., while `workDone` $<$ `globalWork`. Furthermore, all device-side methods add a preamble or postamble macro to each kernel as shown in Fig. 9. The specific preamble and postamble implementations are presented below. Please note that, e.g., modifying atomic variables, calculating work item IDs or handling corner cases, results in lengthy code that we simplified in our presentation below for comprehensibility[6].

A. Atomic Counting

In the *atomic counting* method, each work item acts as a worker that loops over the original kernel code. Initially, multiple same-sized work groups are launched (`clEnqueueNDRangeKernelFused(...)`) and execute in parallel (e.g., one work group per CPU core and multiple ones on GPU). No further work groups are launched during kernel execution. Each work item sequentially executes the kernel body repeatedly for different global IDs. Work items that belong to the same work group execute the kernel body in lock step as shown in Fig. 10. This way, they can share an atomic counter to derive their global IDs and iterate through all IDs that constitute the global work at `local_size` granularity. As detailed in Fig. 11, the atomic counter `workDone` (initially zero) is used to assign group IDs to work items of the same group:

Before each execution of the original kernel code (l.7), each work group (the last work item of a work group) fetches the value of `workDone` and increments the counter by the work group size (l.4 and l.9). The while loop beginning in line 6 is executed until the total amount of work required by the respective kernel launch is done. `workDoneCpy` (defined in l.2) is a variable that stores the fetched value of `workDone` and is allocated once for all work items that belong to the same work group (once for each work group). Independent of how many work groups execute in parallel, `workDoneCpy` will take the values of 0, `get_local_size()`, 2·`get_local_size()`, ..., `globalWork−get_local_size()`, each exactly once for a single work group that enters the while loop (`globalWork` is an integer multiple of `local_size`, see Fig. 3). Accessing the atomic counter only once per iteration of a work group (instead of, e.g., once per work item) reduces contention during the atomic operations, but work items of the same work group need to synchronize after each iteration (thus, execute in lock step). Synchronization is achieved using a barrier. It ensures that every work item of the same work group sees the same value of `workDoneCpy` at all times. This way `workDoneCpy` can be used to derive work item IDs, i.e., `get_global_id()` is redefined as `workDoneCpy` + `get_local_id()`. Ultimately, the original kernel body is executed exactly once for each work item ID 0, 1, 2, ..., `globalWork`−1.

B. Device-Side Enqueuing

The *device-side enqueuing* method does not define a preamble, but only a postamble as detailed in Fig. 12. Similarly to the atomic counting method, it uses the atomic counter `workDone` to keep track globally of how many times the kernel body was executed. Again, only as many work groups are launched initially as needed to fully utilize CPU and GPU (using `clEnqueueNDRangeKernelFused(...)`). The main difference to the atomic counting method is how work is processed by the work items. Instead of looping, a single work item executes the kernel body only once. After executing the

[6]The full implementation of all approaches is available at: https://git.scc.kit.edu/CES/Rodinia-SVM

```
1  __kernel void kernel(...) {
2    local unsigned int workDoneCpy;
3    if (get_local_id() == get_local_size()-1)
4      workDoneCpy = atomic_fetch_add(workDone,
           get_local_size());
5    barrier(CLK_LOCAL_MEM_FENCE);
6    while (workDoneCpy < globalWork) {
7      ... // --- original kernel code ---
8      if (get_local_id() == get_local_size()-1)
9       workDoneCpy = atomic_fetch_add(workDone,
            get_local_size());
10     barrier(CLK_LOCAL_MEM_FENCE); }}
```

Fig. 11: In *atomic counting*, work groups loop over the original kernel code until the total amount of work is done

```
1  __kernel void kernel(...) {
2    ... // --- original kernel code ---
3    if (get_local_id() == get_local_size()-1) {
4      int workDoneCpy = atomic_fetch_add(
           workDone, get_local_size());
5      if (workDoneCpy < globalWork) {
6        ndrange_t child_ndrange =
             ndrange_1D(workDoneCpy,
             get_local_size(), get_local_size());
7        enqueue_kernel(get_default_queue(),
             CLK_ENQUEUE_FLAGS_NO_WAIT,
             child_ndrange, ^{kernel(...);}); }}}
```

Fig. 12: The *device-side enqueuing* method enqueues additional work groups using device-side queues

kernel body, additional work groups may be launched by the work items itself using OpenCL 2.0's device-side enqueuing. As shown in Fig. 12, the work item with the highest ID inside a work group (l.3) launches another work group by enqueuing the current kernel into the device-side queue (l.7).

In OpenCL 2.0, Kernels are enqueued to the device-side command queue using the Clang [15] *block syntax*, a non-standard C extension by Apple Inc. (also known as *closure* in other programming languages) that allows to define functions that can access variables outside their scope (belonging to a captured environment). In our case (l.7) the block `^{kernel(...);}` defines a function that only calls the current kernel with the (captured) arguments that were passed to the initial kernel call from the host-side API. This may seem overly complex for our use case, however, potential alternatives like function pointers are not supported in OpenCL 2.0 and function calls are always inlined [14].

Line 6 defines the parameters of the enqueued kernel, i.e., the global ID offset (`workDone`), the total amount of work items to be launched (`get_local_size()`) and the work group size (`get_local_size()`), respectively. Effectively, work item ID calculation does not have to be redefined as in atomic counting, but `get_global_id()` will return the correct IDs 0, 1, 2, ..., `globalWork`−1 for exactly one work item each. We also evaluated variants of this method, e.g., enqueuing larger amounts of work items than single work groups per `enqueue_kernel(...)` call. However, OpenCL 2.0's device-side enqueuing in general introduces too much overhead (caused by runtime evaluation of the block syntax) to be suitable for co-scheduling as we show in Section VII-A.

C. Host-Side Profiling

In contrast to the device-side co-scheduling methods, the *host-side profiling* method does not apply any modifications to the executed kernels and work items behave exactly the same as in standard OpenCL. Host-side profiling utilizes the OpenCL host-side API, only. Similar to the Inspector-Executor paradigm, the performance behavior of a specific kernel is characterized in an initial phase. Afterwards, this characterization is used to schedule all following executions of the same

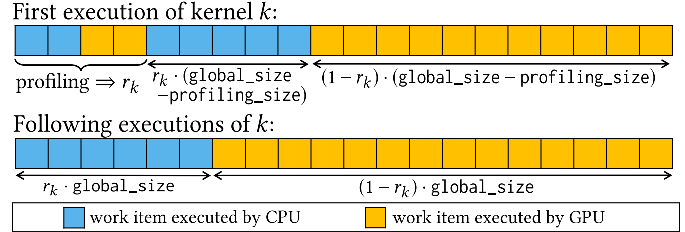

Fig. 13: At the first execution of a kernel k, host-side profiling determines a ratio r_k to distribute work items

kernel. Upon the first execution of a kernel k, only a fraction of the total work items (`profiling_size`) is executed for profiling as shown in Fig. 13. The `profiling_size` is split with half of it executing on the CPU and the other half on the GPU. The execution time of the profiling depends on the specific kernel. OpenCL events are used (1) to synchronize both devices with the host program once profiling finishes and (2) to obtain the execution times of the work items executed on CPU (timeCPU) and GPU (timeGPU), respectively (using the OpenCL API call `clGetEventProfilingInfo(...)`). A ratio $r'_k \in [0, 1]$ of work items to distribute to the CPU is then determined using these measured execution times as follows:

$$r'_k = 1 - (\text{timeCPU}/(\text{timeCPU} + \text{timeGPU}))$$

This ratio is slightly adjusted to obtain the final ratio r_k. Low percentages of work items on GPU showed to be detrimental to the performance compared not using it at all, while following executions on the GPU performed slightly better than the initial profiling in our experiments:

$$r_k = \begin{cases} 1, & r'_k > 0.8 \quad \text{(all CPU)} \\ \min(0, r'_k - 0.05), & \text{else} \quad \text{(mixed CPU/GPU)} \end{cases}$$

Finally, $r_k \cdot$ `global_size` and $(1 - r_k) \cdot$ `global_size` determine the amount of work items executed on CPU and GPU, respectively, for following executions of k (see Fig. 13, the values are rounded to multiples of the work group size). r_k is also used to distribute the remaining work items after profiling. The amount of work items to use for profiling is parameterized. In our experiments we achieved the best compromise between accuracy of the determined ratio and

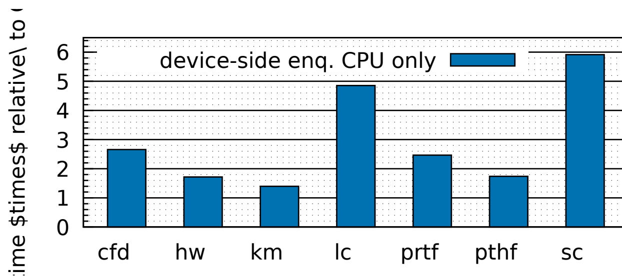

Fig. 14: Device-side enqueuing adds significant overhead, even when no kernel is enqueued. The overheads stem from the kernel call in the block syntax

overhead of the profiling run when 50% of `global_size` was used for profiling when a kernel k was executed for the first time.

VII. Experimental Evaluation

The following results were obtained using a Intel Core i7-6700T (Skylake) fused CPU-GPU architecture with 32 GB of main memory. The Intel Core i7-6700T features a quad-core CPU and the HD Graphics 530 GPU. CPU and GPU share 8 MiB of last level cache (maximum for Skylake). All benchmarks were compiled using GCC version 7.2.1 and the Intel SDK for OpenCL Applications version 2017 R1. They were executed on CentOS Linux release 7.4.1708 with the Intel OpenCL 2.0 CPU/GPU driver package SRB5.0 (Linux kernel 4.7.0.intel.r5.0). To minimize execution time variance, hyper-threading was disabled and CPU frequency scaling set to 'performance' (which sets the highest frequency to all cores and effectively disables turbo boost). The Rodinia-SVM benchmarks were executed using the default inputs from the Rodinia Benchmark Suite for reproducible and comparable results. Results report the average of 10 executions of the respective benchmark with a standard deviation $< 2\%$ of the average, and do not include kernel compilation times[7].

In the following, we first show that device-side enqueuing causes too much overhead to be suitable for co-scheduling. Then, we evaluate our co-scheduling approaches, and finally show that cache coherency is a major performance bottleneck.

A. Device-Side Enqueuing

In this section we evaluate device-side enqueuing on a subset of the Rodinia-SVM benchmarks that result in the highest overheads when device-side enqueuing was applied. We execute the benchmarks in two versions: First we execute the kernels on the CPU only without applying any co-scheduling method. Then, we execute the benchmarks again with the device-side co-scheduling method of Section VI-B applied (still CPU only). However, we immediately launch all work items (the total `global_size`) when the kernels are launched from the host-side API. Effectively, co-scheduling is never actually performed, i.e., the if statement in line 5 of Fig. 12 always evaluates to 'false', i.e., the postamble of the device-side enqueuing method is never executed.

Figure 14 shows the execution time increase of the device-side enqueuing method relative to execution without any co-scheduling method applied. Note that even though the co-scheduling code is not executed, the execution times increase significantly, up to almost 6× for `sc`. The overheads disappear, as soon as we remove the kernel call from the block syntax in line 7 of Fig. 12 (e.g., by replacing `kernel(...)` with a `printf`). This means that runtime processing of the block syntax (capturing the environment) is performed even when that part of the code is not executed, and that it introduces high overheads, which render device-side enqueuing unsuitable for implementing co-scheduling methods. These results may surprise, but are in line with results published by Intel, where a naive port of an iterative implementation of *Sierpiński Carpet* to a recursive implementation using device-side enqueuing resulted in a 186× execution time increase (2050ms instead of 11ms) [16]. Due to this cost, we exclude device-side enqueueing from further experiments.

B. Co-Scheduling Results of Rodinia-SVM

Figure 15 shows evaluation results for the co-scheduling approaches atomic counting and host-side profiling, and execution on CPU-only as well as GPU-only. The results are shown as speedups over the optimal per-kernel choice of whether to execute the kernel either on CPU or GPU (clairvoyant xor-Oracle, see Section III for a discussion compared to a program-fixed ratio as determined by state-of-the-art approaches designed for fused CPU-GPU architectures without shared LLC). All speedups are relative to xor-Oracle (100%) and given in percent (of the relative performance achieved). The geometric mean (gmean) shows that on average execution on GPU-only performs worst (67.5%), mainly because two of the benchmarks (`mc` and `prtf`) perform very badly when their kernels are executed only on the GPU (they contain long-running loops). With 77.6% performance of xor-Oracle on average, execution on CPU-only performs better than GPU-only or with atomic counting. In other words, however, xor-Oracle on average achieves a 1.48× and 1.29× speedup over CPU-only and GPU-only, respectively, by using the most suitable compute device for each kernel.

When using both compute devices in parallel using the co-scheduling methods, one would expect to achieve a considerable speedup over the xor-Oracle that only uses one compute device at a time. As our results show, however, this is rarely the case (which we will analyze further in the following section). At best, atomic counting achieves 110.4% of xor-Oracle's performance (`hw`). On average it achieves 74.8% and thus performs better than GPU-only, but worse than CPU-only. One problem of atomic counting is that some kernels perform very badly on a particular device. Even when only a few work groups are launched initially, their execution times dominate the kernel's overall execution time (e.g., in `mc` and `prtf`). Additionally, atomic counting adds logic, and thus overhead, to the kernels itself.

Host-side profiling, on average, achieves 96.8% of xor-Oracle's performance and a speedup of 1.43× and 1.25×

[7] Kernel compilation can be avoided using `clCreateProgramWithBinary(...)`

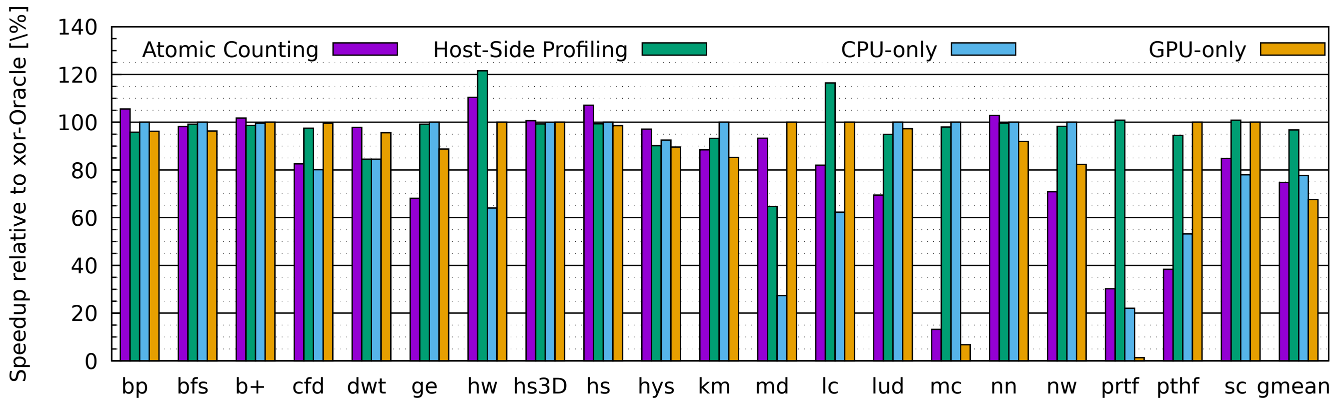

Fig. 15: Speedup of the co-scheduling methods applied to Rodinia-SVM, on a fused CPU-GPU architecture with shared LLC. Results are relative to performing the optimal choice for each kernel of *either* executing on CPU *or* GPU (*xor-Oracle* is 100%)

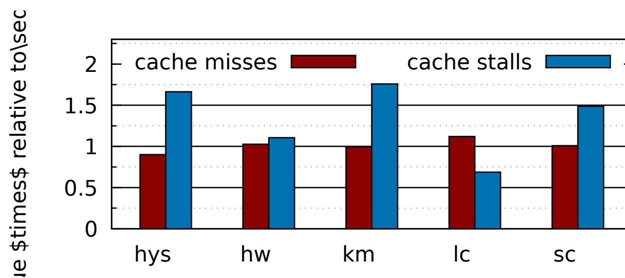

Fig. 16: Cache performance metrics (all levels, measured on CPU) when executing kernels in parallel on CPU and GPU relative to executing the same work item distribution sequentially (first on CPU, then on GPU; = 1 on y-axis)

over GPU-only and CPU-only, respectively. It also performs considerably better on average than atomic counting (1.29× speedup), mainly because it only adds overheads to the very first kernel execution (when profiling) and does not add any code to the kernels. The overhead of profiling is especially evident in `md` that only executes a single kernel once, where host-side profiling performs worst over all benchmarks (64.9% of xor-Oracle). At maximum, host-side profiling achieves a 122.5% of xor-Oracle's performance in `hw`, but only in one other benchmark (`lc`) is another considerable performance benefit over xor-Oracle achieved (116.4%). Note that a host-side profiling implementation that tries to select the best device instead of distributing the work would incur similar overheads without any resulting speedups over xor-Oracle.

In summary, host-side profiling performs best over all methods and is on average competitive to the clairvoyant and thus hypothetical xor-Oracle. However, in most benchmarks it does not benefit from executing kernels on CPU and GPU in parallel compared to executing on the most-suitable single compute device, only.

C. Cache Performance Bottleneck

To analyze why executing kernels on both compute devices in parallel on fine-grained SVM does on average not provide a considerable performance benefit over executing the kernels on the most-suitable device only, we measured cache metrics using CPU-internal hardware performance counters. A subset of the Rodinia-SVM benchmarks was selected, for which host-side profiling was utilized to distribute work items to CPU and GPU for all kernels ($\forall k : 0 > r_k < 1$). These benchmarks potentially benefit most from utilizing both devices in parallel. Furthermore, the selected benchmarks synchronize with the host after each kernel execution (the same as in their original versions) which allows us to measure the performance counters for the kernel executions, only. We use the ratios r_k from the previous section for all kernels k, without performing the profiling step of the host-side profiling method.

First, all benchmarks are executed while using the devices sequentially, i.e., for each kernel we execute the work items assigned to the CPU first, synchronize with the host, and then execute the work items assigned to the GPU. For this *device-sequential* execution, the total cache misses and cache stalls (all levels) that are encountered by the CPU are measured[8]. Then, all benchmarks are executed while using both devices in parallel (as in the previous section) and the same measurements are performed. In both measurements the CPU (and GPU) performs the same amount of work, but in the device-sequential case the CPU has more idle time.

Fig. 16 shows the measured cache metrics from the device-parallel execution relative to the device-sequential execution (= 1 on y-axis). For `hys`, `km` and `sc`, the cache misses do not increase (`hys` even benefits from device-parallel execution). This means that there are no cache conflicts like false or true sharing that impair the performance. However, the cache-related stalls increase considerably by up to 1.75× and 1.64× on average. A similar effect has previously been observed under simulation for cache-coherent fused architectures without a shared LLC [17]. The authors demonstrated that the amount of data probes sent by the highly-parallel GPU to the shared cache directory occupied the directory bandwidth which considerably slowed down the memory bandwidth that

[8]There are no publicly documented interfaces to access Intel GPU performance counters when not using OpenGL

can be sustained by the cache hierarchy. Our results demonstrate the existence of a similar cache coherency bottleneck when fine-grained SVM is used on the Intel fused CPU-GPU architecture, even when CPU and GPU share an inclusive LLC. Further research is required to analyze and resolve this bottleneck (in software or hardware) to fully benefit from co-processing on fused CPU-GPU architectures.

For `hw` and `lc`, a similar increase in cache-related stalls cannot be observed. These results are in line with the speedup results shown in Fig. 15: `hw` and `lc` (group 1) benefit considerably from co-scheduling over the xor-Oracle, while `hys`, `km` and `sc` (group 2) do not. The main difference between these two groups of benchmarks is that the kernels of group 1 are considerably longer (>100 lines of code on average) than the kernels of group 2 (<30 lines of code on average). Therefore, the kernels of group 1 perform considerably more operations per work item than the kernels of group 2.

VIII. Conclusion and Future Work

This work presented the first investigation of collaborative execution of computational kernels on a fused CPU-GPU architecture with a shared LLC using fine-grained SVM. We contributed two novel device-side co-scheduling methods that perform scheduling within the kernel code. It was shown that device-side enqueuing introduces considerable overhead stemming from the evaluation of the block syntax that is used in device-side enqueuing of kernels (up to $6\times$ execution time increase), too much to be suitable for implementing co-scheduling methods.

Our host-side co-scheduling method achieved 96.8% of the clairvoyant and thus hypothetical xor-Oracle's performance on average (optimal per-kernel choice of exclusive CPU or GPU usage) and a speedup of $1.43\times$ and $1.25\times$ over execution on GPU only and CPU only, respectively. It also provided a $1.29\times$ speedup over 'atomic counting', the best device-side co-scheduling method, because it does not add overhead to kernel execution once profiling is done. This makes our host-side co-scheduling method the most competitive practical scheme to date. We further showed that cache coherency is the major performance bottleneck in current fused CPU-GPU architectures with a shared LLC. It was shown that when CPU and GPU execute kernels in parallel on an Intel architecture, cache-related stalls observed on the CPU can increase by up to $1.75\times$ while cache misses remain the same compared to executing the same work on the CPU and only then on the GPU (while the CPU is idle).

However, some benchmarks benefited considerably from collaborative execution on CPU and GPU (up to $1.23\times$ speedup) compared to using the most suitable device. It depends on the memory access patterns of the kernels whether cache coherency becomes a performance bottleneck or not. In future work, we will categorize memory access patterns and design optimizations to alleviate this performance bottleneck for even more effective co-scheduling of kernels on fused CPU-GPU architectures.

References

[1] Y. Yang, P. Xiang, M. Mantor, and H. Zhou, "Cpu-assisted gpgpu on fused cpu-gpu architectures," in *Int. Symp. on High Perf. Comp. Arch.* IEEE, 2012, pp. 1–12.

[2] S. Mukherjee, Y. Sun, P. Blinzer, A. K. Ziabari, and D. Kaeli, "A comprehensive performance analysis of hsa and opencl 2.0," in *IEEE Int. Symp. on Perf. Analy. of Syst. and Soft.* IEEE, 2016, pp. 183–193.

[3] S. Junkins, "The compute architecture of intel® processor graphics gen9," *Intel whitepaper v1*, 2015.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE Int. Symp. on Workl. Charact.* Ieee, 2009, pp. 44–54.

[5] F. Zhang, J. Zhai, B. He, S. Zhang, and W. Chen, "Understanding co-running behaviors on integrated cpu/gpu architectures," *IEEE Trans. on Par. and Dist. Syst.*, vol. 28, no. 3, pp. 905–918, 2017.

[6] F. Zhang, B. Wu, J. Zhai, B. He, and W. Chen, "Finepar: Irregularity-aware fine-grained workload partitioning on integrated architectures," in *IEEE/ACM Int. Symp. on Code Gen. and Opt.* IEEE, 2017, pp. 27–38.

[7] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali, "Adaptive heterogeneous scheduling for integrated gpus," in *Proc. of the Int. Conf. on Par. Arch. and Comp.* ACM, 2014, pp. 151–162.

[8] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices," in *IEEE/ACM Int. Symp. on Code Gen. and Opt.* ACM, 2014, p. 273.

[9] R. Barik, N. Farooqui, B. T. Lewis, C. Hu, and T. Shpeisman, "A black-box approach to energy-aware scheduling on integrated cpu-gpu systems," in *IEEE/ACM Int. Symp. on Code Gen. and Opt.* IEEE, 2016, pp. 70–81.

[10] J. Liu, N. Hegde, and M. Kulkarni, "Hybrid cpu-gpu scheduling and execution of tree traversals," in *Proc. of the Int. Conf. on Supercomp.* ACM, 2016, p. 2.

[11] L. Wang, R.-W. Tsai, S.-C. Wang, K.-C. Chen, P.-H. Wang, H.-Y. Cheng, Y.-C. Lee, S.-J. Shu, C.-C. Yang, M.-Y. Hsu *et al.*, "Analyzing opencl 2.0 workloads using a heterogeneous cpu-gpu simulator," in *IEEE Int. Symp. on Perf. Analy. of Syst. and Soft.* IEEE, 2017, pp. 127–128.

[12] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous cpu-gpu simulator," *IEEE Comp. Arch. Letters*, vol. 14, no. 1, pp. 34–36, 2015.

[13] D. R. Kaeli, P. Mistry, D. Schaa, and D. P. Zhang, *Heterogeneous Computing with OpenCL 2.0.* Morgan Kaufmann, 2015.

[14] K. O. W. Group *et al.*, "The OpenCL specification version 2.0," *https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf*, 2015.

[15] C. Lattner, "LLVM and Clang: Advancing Compiler Technology," *Proc. of FOSDEM*, 2011.

[16] R. Ioffe, S. Sharma, and M. Stoner, "Achieving performance with OpenCL 2.0 on Intel®processor graphics," in *Proc. of Int. Works. on OpenCL*. ACM, 2015, p. 3.

[17] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous system coherence for integrated CPU-GPU systems," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 457–467.

Marvin Damschen (M'15) received a B.Sc. degree –with distinction– and M.Sc. degree –with distinction– in Computer Science with a minor in Mathematics from the University of Paderborn, Germany, in 2012 and 2014, respectively. Currently, he is pursuing his Ph.D. at the Chair for Embedded Systems (CES) at the Karlsruhe Institute of Technology (KIT), Germany, under the supervision of Prof. Dr. Jörg Henkel. His current research interests include analysis and architectures for predictable embedded systems with special focus on heterogeneous and reconfigurable architectures. Mr. Damschen has served as an external reviewer for major conferences in embedded systems and computer architecture. He is a student member of the IEEE.

Frank Mueller (M'94-SM'06-F'16) is a Professor in Computer Science and a member of multiple research centers at North Carolina State University. Previously, he held positions at Lawrence Livermore National Laboratory and Humboldt University Berlin, Germany. He received his Ph.D. from Florida State University in 1994. He has published papers in the areas of parallel and distributed systems, embedded and real-time systems and compilers. He is a member of ACM SIGPLAN, ACM SIGBED and a senior member of the ACM and an IEEE Fellow. He is a recipient of an NSF Career Award, an IBM Faculty Award, a Google Research Award and two Fellowships from the Humboldt Foundation.

Jörg Henkel (M'95-SM'01-F'15) is the Chair Professor for Embedded Systems at Karlsruhe Institute of Technology. Before that he was a research staff member at NEC Laboratories in Princeton, NJ. He received his diploma and Ph.D. (Summa cum laude) from the Technical University of Braunschweig. His research work is focused on co-design for embedded hardware/software systems with respect to power, thermal and reliability aspects. He has received six best paper awards throughout his career from, among others, ICCAD, ESWeek and DATE. For two consecutive terms he served as the Editor-in-Chief for the ACM Transactions on Embedded Computing Systems. He is currently the Editor-in-Chief of the IEEE Design & Test Magazine and is/has been an Associate Editor for major ACM and IEEE Journals. He has led several conferences as a General Chair incl. ICCAD, ESWeek and serves as a Steering Committee chair/member for leading conferences and journals for embedded and cyber-physical systems. Prof. Henkel coordinates the DFG program SPP 1500 "Dependable Embedded Systems" and is a site coordinator of the DFG TR89 collaborative research center on "Invasive Computing". He is the chairman of the IEEE Computer Society, Germany Chapter, and a Fellow of the IEEE.