Abstract Representation of Shared Data for Heterogeneous Computing

Tushar Kumar[1], Aravind Natarajan[1], Wenjia Ruan[1], Mario Badr[2], Dario Suarez Gracia[3], and Calin Cascaval[4]

[1] Qualcomm Research**: {tushark,naravind,wenjiar}@qti.qualcomm.com
[2] University of Toronto: mario.badr@mail.utoronto.ca
[3] Universidad de Zaragoza: dario@unizar.es
[4] Barefoot Networks: cascaval@acm.org

Abstract. Data management across address spaces in heterogeneous platforms represents a significant performance bottleneck and energy cost for applications, particularly on mobile System-on-Chip (SoC). We propose a light-weight middleware layer to regulate concurrent access to shared data in a Heterogeneous SoC. Our approach uses acquire-release semantics to provide the following benefits: *i)* enable high-level heterogeneous programming frameworks to easily maintain consistent non-device, non-platform-specific data abstractions for programmers, and *ii)* provide an abstract memory interface with strong analyzable properties about the correctness and performance of the synchronization operations across memory-types. These benefits are achieved while retaining the ability to plug-in arbitrary types of heterogeneous memory frameworks and to enable platform-specific and inter-framework synchronization optimizations. We experimentally demonstrate that our approach avoids paying the "abstraction cost" and compares favorably with manually optimized OpenCL, while providing a simpler and understandable API.

Keywords: heterogeneous System-on-Chip, memory synchronization, memory concurrency, data sharing

1 Introduction

Heterogeneous computing systems allow programmers to match parts of an application to the strengths of the different devices available [14]. The ultimate goal of heterogeneous computing is to obtain higher performance at lower power by judiciously balancing the computation. Prior work has focused on partitioning applications across heterogeneous devices. For example, the Fast Multipole Method has been shown to work better on a CPU-GPU architecture [10]. However, heterogeneous systems are not limited to CPUs and GPUs. As we scale to a more diverse set of accelerators, a major impediment to programmers becomes moving data across devices. Mobile Systems-on-Chip (SoCs) typically share data across devices using a contiguously-allocated block of memory (i.e., a buffer) that

** Qualcomm Research is a division of Qualcomm Technologies, Inc.

is modifiable by one device at a time. Without advanced hardware support [2] synchronization is *explicit*, placing a significant burden on programmers. We propose to simplify data movement across devices via intuitive abstractions.

Not all devices share a common view of system memory or even have access to it. For example, a device may not be cache coherent, and some devices may address 32-bit memory while others address 64-bit. Our approach leverages the familiar notion of a *buffer* augmented with acquire-release semantics to deal with this non-uniformity. We abstract the diverse mechanisms for data access into a uniform set of synchronization primitives that is implemented on top of hardware support. Put simply, in acquire-release semantics either the entire buffer is made available to a kernel or none of it is. This provides programmers with a familiar technique for sharing data in a heterogeneous system.

Currently, many frameworks exist to enable offloading data to a device. For example, OpenCL, CUDA, or OpenGL are used to offload computation to the GPU and to share data between the CPU and GPU [7, 18, 22]. Android ION is another industry standard that allocates memory accessible by any ION-compliant devices on an SoC. It is often used to share data between compute devices and custom-components of the SoC, such as image-processing accelerators [11]. However, to efficiently use ION from OpenCL kernels (i.e., to avoid unnecessary copying of data from ION into GPU-accessible memory regions), programmers must use specialized extensions of OpenCL API calls (typically vendor specific). This results in multiple versions of the application code to support the range of platforms. Prior work focused on establishing a cache-coherent shared memory model over multiple devices does not capture this level of intricacies [6].

The challenge of managing multiple frameworks for offloading data gets more complex as more devices are added to heterogeneous systems. For example, FPGAs and ML accelerators [1, 17] will have their own mechanisms. Currently, a programmer looking to take advantage of all the compute devices available must:

1. Synchronize data across any combination of devices correctly and efficiently
2. Be aware of supported memory-optimizations for each target platform
3. Write application code to accommodate multiple device combinations

To alleviate the programmer's burden and enable heterogeneous applications, we propose an abstraction layer with a novel representation for a shared data buffer (Figure 1). The synchronization state of a shared data buffer is maintained in a device- and platform-agnostic manner (*buffer state* – the middle of Figure 1). An existing memory framework; e.g., OpenCL or ION, is plugged-in underneath the buffer-state via an abstract representation of memory called an *arena* (the bottom of Figure 1). An arena provides generic operations to the buffer-state to (1) allocate storage and (2) manage access to the storage, while hiding the particulars of the underlying memory framework. Each arena maintains an *arena state* capturing information sufficient for the buffer-state to correctly manage storage and synchronize data across arenas. The arena state includes the *map state*; i.e., which devices may currently access the storage (a generalization over OpenCL operations that burden the user to track state). Arenas serve as a wrapper around an existing memory framework, placing the onus on the platform developer rather

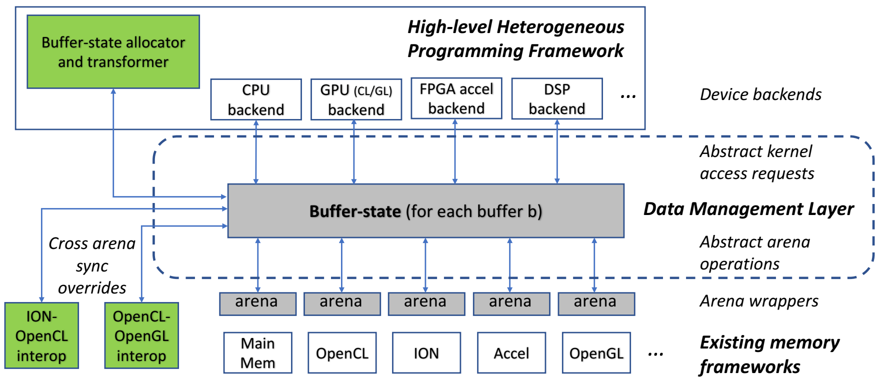
Buffer-state allocator and transformer
High-level Heterogeneous Programming Framework
CPU backend
GPU (CL/GL) backend
FPGA accel backend
DSP backend
...
Device backends
Abstract kernel access requests
Buffer-state (for each buffer b)
Data Management Layer
Abstract arena operations
Cross arena sync overrides
ION-OpenCL interop
OpenCL-OpenGL interop
arena
arena
arena
arena
arena
Arena wrappers
Main Mem
OpenCL
ION
Accel
OpenGL
...
Existing memory frameworks

Fig. 1. The abstract representation of shared data decouples existing memory frameworks from their use by a high-level programming framework.

than the application developer. For example, an OpenCL-arena would translate the arena operations into appropriate OpenCL API calls and correspondingly update the arena state. The arena and buffer-state transactions together ensure (1) correct storage allocation and synchronization of shared data across the memory frameworks, and (2) correct throttling of concurrent access by device kernels to the shared data based on the arena states. To ensure synchronization between any two arenas, the buffer-state requires the arenas to support a *default synchronization mechanism*, for example, require all arenas to allow the CPU to access their data, so the CPU may perform memory-copies between arenas. An ION-arena may simply provide a pointer to its ION-allocated storage, while an OpenCL-arena may use the OpenCL APIs to “read” or “map” data from the GPU memory into main memory (memory frameworks typically provide CPU-access mechanisms as current heterogeneous devices typically offload the CPU). The buffer-state synchronizes data across arenas using two interface functions: $can\_copy(src, dst)$ and $copy(src, dst)$. Under the default mechanism described above, $can\_copy(src, dst)$ would prevent synchronization between the src and dst arenas if granting access to the CPU would interfere with kernels already accessing the data in src, e.g., an OpenCL-arena in unmapped state may not be mapped for access by the CPU while GPU OpenCL kernels are accessing it. When permitted by $can\_copy(src, dst)$, $copy(src, dst)$ performs the data synchronization, including allocating storage inside dst if not already allocated.

The platform developer may override the default synchronization mechanism between two arenas with a *cross-arena synchronization mechanism* (bottom-left of Figure 1) that is faster, requires less storage-allocation than the default, and/or provides alternative paths to synchronization. While the default $can\_copy()$ may disallow synchronization when the CPU cannot be granted access, the optimized $can\_copy()$ may evaluate use of platform-specific or memory-type-specific mechanisms to perform the synchronization without disrupting executing kernels. The corresponding $copy()$ may also *bind-allocate* storage between arenas, e.g., the MainMem-arena and OpenCL-arena may directly use the storage allocated in the ION-arena instead of allocating their own. Crucially, the arena abstraction greatly alleviates the burden on the platform developer when implementing cross-arena optimizations – the developer may perform arbitrary transforms on the arena-state to carry out the synchronization, while easily verifying against the arena-state whether a transform impacts access to currently executing kernels.

Finally, the buffer-state itself provides an abstract representation that can be transformed by a high-level heterogeneous runtime in a fairly arbitrary manner to implement sophisticated program-specific and platform-specific optimizations, with an automatic guarantee of correctness (top-left of Figure 1).

In this paper we make the following contributions:

1. We define an abstract representation of memory regions and synchronization primitives to maintain consistency across memory regions in the presence of multiple heterogeneous devices with different capabilities (Section 3).
2. We propose an overload mechanism, the cross-arena, that allows arbitrary memory synchronization optimizations to be plugged-in while retaining the ability to easily verify correctness of implementation (Section 3.2).
3. We implement these abstractions into a middleware, which we refer to as the *data management layer* (DML), that enables the creation of high-level programming frameworks with abstract representations of shared data, and provides portability across a range of platforms supporting any combinations of low-level memory-frameworks (Section 4).
4. We evaluate the proposed ideas by incorporating the DML into a C++ high-level programming framework, the Qualcomm® Symphony System Manager SDK[24, 23] (*"the SDK"*). The SDK exploits its knowledge about program structure and data sharing patterns by performing transformations on the buffer-state abstraction, and plugs-in cross-arena optimizations to take advantage of platforms-specific synchronization mechanisms (Section 5).

In summary, our abstraction decouples the low level, platform-specific details of sharing and synchronizing memory regions from the high-level programming data abstraction and allows application programmers to write efficient, portable code. Next, we motivate our work with examples of the programmer burden.

2 Illustration of Programmer's Burden

Consider the application phases in Figure 2. In the first phase, buffers are allocated for use with a CPU, GPU and DSP. On the given platform, ION is used to share data with the DSP, whereas an additional pool of memory is available to the GPU OpenCL driver for sharing data between the CPU and GPU. Option α allocates a buffer from GPU driver memory. Option β allocates from the ION pool and then uses the OpenCL-ION extensions available on some platforms to access ION memory efficiently from the GPU. Option ϵ allocates ION memory, but does not use the OpenCL-ION extensions, leading to the GPU OpenCL driver potentially copying data back and forth between the ION- and GPU-allocated storage. Option δ allocates ION storage for data that will only be shared between

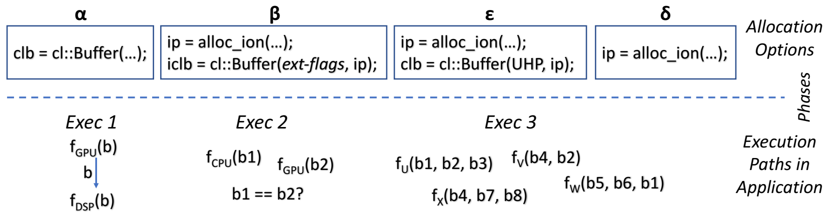

Fig. 2. Data management burden: the heterogeneous application developer is forced to track platform-specific functionality and the dynamic execution order of components.

the CPU and DSP. For each buffer, the application programmer has to keep track of which buffer allocation option – α, β, ϵ or δ – was used. For data to be shared between the CPU, GPU and DSP, the programmer has to select at compile-time or run-time between options β and ϵ, to support platforms with and without support for the OpenCL-ION extensions.

Next, consider the data management challenges in the execution phase of the application. Exec 1 shows a producer-consumer relationship between a GPU and a DSP component using data buffer b. If b has format β, typically an OpenCL 'map' would be the correct synchronization operation. For format ϵ, however, an OpenCL 'read' may be more performant on some platforms, requiring the programmer to select synchronization operations based on platform. Exec 2 shows a CPU and a GPU component executing concurrently. When $b1$ and $b2$ are distinct data buffers, the two components may execute without any synchronization. However, when invoked on the same buffer, the programmer has to detect $b1 = b2$ during execution and serialize the components; e.g., by holding locks on the buffers. If f_{CPU} is serialized to execute before f_{GPU} and the buffer was allocated using option α or ϵ, the programmer must perform OpenCL 'unmap' or 'write' operations in between (whichever is more performant on the given platform), and 'map' or 'read' for the opposite execution order. With option β, 'unmap' or 'map' would correspondingly be used for the two serialization orders.

Of course, if the components only read the buffers, it would generally be feasible to execute them concurrently even when $b1 = b2$. Even so, option α allocation may force a serialization if f_{GPU} executes first and the buffer has never been 'mapped' before — the f_{CPU} component is unable to access the data because the OpenCL buffer may not be 'mapped' while in use by the GPU.

Finally, Exec 3 shows a scenario of arbitrary concurrent components running on multiple heterogeneous devices (with more than one component possible on a device). Each component accesses multiple buffers. Depending on whether a given buffer, say $b4$, is read-only accessed by f_V and f_X or not, control serialization must be imposed between f_V and f_X. Similarly, every other buffer creates potential serialization between any of f_U, f_V, f_W and f_X. Correspondingly, there are a large combinations of data synchronization operations that must be judiciously selected between any serialized components to ensure correct and performant data management. This quickly becomes overwhelming for the programmer to manage, even at the level of relatively platform-abstract OpenCL and ION APIs.

3 Abstract Representation

Clearly a high-level programming runtime should alleviate the burden on the programmer by taking care of the data management, and consider a number of optimizations based on the memory access patterns: read-only accesses, concurrent access, allocation usage, etc. Toward this goal, our proposed data management layer (DML) introduces an abstract representation of the following concepts: *arena* to abstract operations over a device-specific memory, *buffer state* to abstract the distribution of a data buffer over multiple device-specific memories and to track the device kernels accessing the data, and *cross arena* to compartmentalize knowledge about platform-specific synchronization optimizations.

3.1 arena

The arena representation captures the following *abstract state* information for a single type of memory or memory framework:

– **allocation state**: the storage of the underlying memory framework may be *unallocated*, *internally-allocated*, *externally-allocated*, or *bind-allocated.*
– **bind-arena**: identifies which other arena under the buffer state is this arena bind-allocated with (if at all).
– **native device**: a fixed attribute identifying which device accesses the memory wrapped by this arena to execute its kernels.
– **remote devices**: a fixed attribute identifying additional devices that are directly able to access the storage on this memory, not necessarily efficiently. By 'directly' we mean the remote device is able to access the data without involving another device (to perform a mem-copy, for example).
– **map handles**: for each remote device a handle that the device may use to access the data storage on this memory. For example, an OpenCL 'map' operation would provide the CPU with a 'map pointer' handle to access the OpenCL-allocated memory in an OpenCL-arena. The map handle for a custom accelerator's memory arena may be a set of memory IDs that a remote CPU may use to program a DMA engine. Therefore, the map handle may have a platform- or memory-dependent format, but the handle is abstract in the sense that it is never interpreted by the buffer-state or arena abstractions.
– **map state**: either *unmapped*, *mapped* or *bimapped.* Unmapped implies that the data storage is currently accessible to the native device. Mapped implies that the data storage is available to an identified subset of remote devices using their corresponding map handles. Bimapped implies simultaneous access by the native device and an identified subset of remote devices.
– **ref-count**: tracks how many kernels executing on the native device have currently been granted access to the data storage on this memory.

The allocation state captures whether storage has been allocated for the data buffer on the memory wrapped by the arena. An internal-allocation implies that the arena itself has allocated the storage, which will be de-allocated when the buffer-state containing the arena is de-allocated. An external-allocation implies that the application program or some other entity in the high-level programming framework has already allocated storage on the wrapped memory, and wants the buffer-state to use the available storage and the data on it. Bind-allocated is crucial for optimizing the storage allocation and synchronization across different types of memory holding the shared data buffer. It implies that the storage has been allocated by another arena, and there is some specialized cross-arena mechanism available to synchronize data between this arena and the another arena. For example, an OpenCL-arena and a MainMem-arena may both bind-allocate to an ION-arena – this means that storage was initially allocated inside ION memory, and the OpenCL-arena and MainMem-arena are able to use the ION-allocated storage pointer instead of allocating from their own memory pool.

An *arena wrapper* over a memory framework consists of the following aspects:

1. **Abstract arena state**: as described above.
2. **Abstract arena operations API**: a device-, platform- and memory-agnostic API, used by the buffer state to manipulate the abstract arena state.
3. **Arena implementation**: concrete implementation that is highly specific to the memory-type. Ties the abstract state and the abstract operations to the specific memory-type or memory-framework wrapped by this arena; e.g., translate to OpenCL API calls in an OpenCL-arena.

For example, an OpenCL-arena would have native device as GPU, remote device as CPU, internal-allocation would allocate an OpenCL buffer, external-allocation would track a user-allocated OpenCL buffer, bind-allocate would create an OpenCL buffer using memory from a MainMem-arena or ION-allocated memory from an ION-arena, the map handle would be a CPU-accessible pointer, the map state would correspond to the map/unmap state of the contained OpenCL buffer, and ref-count would track how many GPU kernels currently access the contained OpenCL buffer. A MainMem-arena would have both the native and remote device as CPU, internal-allocation would malloc storage, external-allocation would save a user-pointer, bind-allocation would use the pointer from an ION-arena or the pointer provided by the OpenCL 'map' operation in an OpenCL-arena, the map-state would always be bimapped, and the ref-count would track how many CPU kernels currently access the MainMem-arena.

The abstract arena API provide the following operations to the buffer state:

- **allocate storage** – internal, external or bind-allocate.
- **request/revoke access** to one or more remote devices, or to the native device. Depending on the specific arena implementation, granting access may implicitly revoke access for another device, which the map state will reflect.
- **up-ref** tracks that one more kernel executing on the native device now has access to the arena's storage, while **down-ref** indicates a kernel completion.

The up-ref call increments the arena ref-count, while down-ref decrements it. A ref-count $= 0$ indicates no kernel on the native device is accessing the storage allocated in this arena, and hence any map-state changes are now permitted. A ref-count > 0 would disallow any map-state change that would disrupt access to executing kernels, possibly preventing data synchronization with another arena.

3.2 cross-arena

The cross-arena representation allows the following synchronization operations to be overridden by platform-specific implementations:

- bool **can_copy**(src-arena, dst-arena)
- **copy**(src-arena, dst-arena)

The buffer state provides a default implementation of these functions. In one possible default implementation, the CPU serves as a remote device for every arena, and the src and dst arenas can be mapped so the CPU may perform a mem-copy. In general, can_copy(src, dst) returns success if the data in the src arena can be used to update the storage in the dst arena. It may fail for the following reasons – *i)* the src and dst do not share a common device in either their native or remote devices ("the sync device"), or *ii)* using the sync device to access the data would require a change to src's map state, but ref-count > 0 in the src's arena state currently prevents that change. Note that when the buffer state invokes can_copy() or copy(), the dst arena does not contain valid data and has ref-count $= 0$, making any map-state change in dst feasible.

The cross arena interface in the DML provides a mechanism for plugging in platform-specific and memory-type specific overrides for any combination of src and dst arena types. The overridden copy() will also determine the best manner to allocate storage in the dst arena (whether to internal-allocate or bind-allocate), which frequently enables zero-copy optimizations. can_copy() can also return a cost to help buffer-state more precisely determine the cheapest copy mechanism.

The buffer-state abstract machine may evaluate multiple combinations of can_copy() calls to determine the cheapest or most optimal "sync sequence" currently available to synchronize data into a desired dst arena. On finding the best available sequence, the abstract machine will call one or more copy(src, dst) to enact data transfer on the chosen sequence. The copy() calls will allocate storage in their corresponding dst arenas if the storage was not already allocated. In this manner, more optimal bind-allocations that *minimize synchronization costs and the total amount of storage allocated* may be performed.

Overlapping remote and native devices across arenas create a path for the movement of data across corresponding memory types. Therefore, the buffer state essentially needs to find the 'shortest sync path' from any src arena currently holding *valid data* to the dst arena needing to get the shared data of the buffer. The path is over a graph whose nodes represent arenas, and an edge connects two arenas if they have a sync device in common. The edge cost is the cost returned by can_copy(). Overridden copy() and can_copy() may provide new mechanisms to synchronize data without changing the map-state of the src, and may also introduce alternative remote devices. *Therefore, platform-specific cross-arena overrides often enhance the degree of concurrency possible for executing kernels.*

As an example, consider that memory-type $m1$ (wrapped in arena $a1$) currently holds valid data. Suppose device $d2$, the native device of memory-type $m2$ (wrapped in arena $a2$) needs to execute a kernel that reads the data buffer. Suppose can_copy($a1$, $a2$) fails because $a1$'s map-state cannot be currently changed. The buffer-state's abstract machine can try can_copy($a1$, $a3$) and can_copy($a3$, $a2$) to accomplish can_copy($a1$, $a2$) if another arena $a3$ has sync devices in common with $a1$ and $a2$. This may also create bind-allocations between $a1$, $a2$ and $a3$.

Next, consider custom hardware accelerators $accel1$ and $accel2$ on an FPGA, with their corresponding private memory banks $m1$ and $m2$ wrapped by arenas $a1$ and $a2$, respectively. $a1$ has the CPU as remote device because $m1$ is connected to the system bus, but $a2$ only has $accel1$ as remote device because $accel1$ can also access $m2$. Such a situation may occur in custom FPGA designs, and our proposed abstractions will find a path to transmit data from the CPU to $accel2$.

3.3 buffer-state

The buffer-state representation contains the following information:

– **arenas-set** – one arena for each type of memory.
– **valid-set** – which arenas currently have valid data.
– **requestor-set** – which kernels (over multiple devices) currently access the shared data buffer, what are their corresponding *access types* (read-only, write-invalidate, read-write), and which arena is accessed by each kernel.

The high-level programming runtime makes a *kernel-access request* to the buffer-state whenever a kernel k on device d needs to access data buffer b. Each data buffer maintains a distinct buffer-state. To grant a kernel-access request, the buffer-state's abstract machine must first ensure that the *access-type of the request does not conflict* with the accesses already granted; e.g., multiple read-only access requests may be granted, but only one write-invalidate or read-write request may exist in the requestor-set. Secondly, the buffer-state must ensure that an arena with native device d has been *i)* storage allocated and *ii)* the arena either already has valid data, or a sequence of can_copy() calls have been found that will make the latest data buffer contents available on the arena. If so, the buffer-state will issue the corresponding sequence of copy() calls, and return a "grant-success" to the high-level runtime. A "grant-failure" would prevent k from executing, whereby the DML would force a serialization in the runtime until one or more of the already executing kernels complete and are removed from the requestor-set. Completion of another kernel $k2$ may perhaps resolve an access-type conflict with k. Or, the arena $a2$ associated with $k2$ may now allow map state changes suitable for copying valid data out from $a2$ into the arena needed for executing k.

A kernel k may be represented in the requestor-set using any encoding scheme suitable for the high-level runtime. For example, a kernel could be represented simply by a pointer if the high-level runtime has a unique pointer to each kernel, or the kernel may be represented by a more complex data structure in the requestor-set. The only requirement is for the kernel representation to support an equality test, so the presence of a kernel k can be checked for in the requestor-set.

Our proposed data management layer relies on acquire-release semantics to provide kernels access to a shared buffer. Either the entire buffer is made available to a kernel or none of it is. For this reason, it is sufficient to maintain a valid-set, where each arena can be marked as holding valid data or as invalid. All arenas that are marked valid are considered to hold identical data contents.

3.4 Correctness Under Concurrent Kernel Access Requests

The following steps are taken by the buffer state on a kernel access request. Invariant properties at each step provide the requisite correctness guarantees under arbitrary concurrent requests from the high-level programming runtime.

Terms Given kernel k requesting access to shared buffer b:

1. $RS(b)$: Requestor-set of b, the kernels that currently access b.
2. Kernel's device d: the device k will execute on. Can be more specific than just the hardware component; e.g., d = cpu, gpucl, gpugl; when the GPU supports both OpenCL and OpenGL kernels.
3. Kernel's arena to access b: arena a under b's buffer-state that k will use to access the data buffer contents. The native device for a would be d.

***Step 1*: Acquire(k, b)**: the runtime makes a kernel access request to the DML

1. A lock is held on b to stall another concurrent request on b until the current request is granted or fails.
2. The access-type of k is checked against $RS(b)$ – request fails if a conflict is found with another kernel in $RS(b)$, e.g., if k requests read-write access to b.
3. If a does not have valid data, buffer-state determines a can_copy sequence to make a valid, or returns failure if no such sequence currently exists.
4. k and its access-type is added to $RS(b)$.
5. buffer-state executes the corresponding copy() sequence, to make a valid.
6. change a to unmapped state or bimapped state to allow access by k (unmapped vs bimapped depends on the arena-wrapper implementation).
7. a.upref()

Invariant: Either k fails to acquire b, or k's arena a is identified, has valid data, and has been put in a map-state suitable for access by k.

***Step 2*: Execute(k)**: runtime executes k after acquiring all its buffers.

1. No lock is held on b, allowing concurrent kernel access requests involving b to be made while k executes.
2. k executes and accesses the storage inside a. For example, an OpenCL k would be launched with the OpenCL buffer extracted from inside b's OpenCL-arena.

Invariant: The map-state of a will remain unmapped or perhaps become bimapped during the execution of k. No Acquire($k2$, b) or Release($k2$, b) for a concurrent kernel $k2$ may make a inaccessible to k. a will continue to hold the latest valid data of the shared buffer b until k completes execution.

***Step 3*: Release(k, b)**: runtime relinquishes access to b from k once k has completed execution.

1. A lock is held on b to update buffer-state of b safely.
2. k is removed from $RS(b)$.

```
void foo(float *a, float *b, int size, float& result_sum) {
  // Create buffers using storage and data from existing pointers
  auto buf_a = symphony::create_buffer<float>(a, size);
  auto buf_b = symphony::create_buffer<float>(b, size);

  // Pass 'hint' that buffer will be accessed by CPU, GPU and DSP devices. The hint enables optimal
  // allocation of storage by Symphony & DML on any platform, e.g., pre-allocate ION if available.
  auto buf_c = symphony::create_buffer<float>(size, {symphony::cpu, symphony::gpu, symphony::dsp});

  // Execute GPU task first
  auto t1 = symphony::launch(gpu_kernel, symphony::range<1>(size), buf_a, buf_b, buf_c);
  t1->wait_for();

  // Execute DSP and CPU tasks concurrently: both read buf_c, producing result_sum and buf_a.
  auto t2 = symphony::launch(dsp_kernel, buf_c, &result_sum);
  auto t3 = symphony::launch(cpu_kernel, buf_c, buf_a);

  t2->wait_for();
  t3->wait_for();
}
```

Fig. 3. High-level compute APIs in the SDK enabled by the data management layer.

A b-specific lock is held only for the duration of the Acquire and Release calls involving b. Kernel execution is allowed to happen without the lock held. The steps above have the annotated invariants that guarantee correctness. *High concurrency in kernel execution* is facilitated by *i)* having individual locks for buffers, and *ii)* only briefly holding those locks for associated Acquire and Release.

4 Use in a Heterogeneous Programming Runtime

We incorporate our proposed DML inside one high-level framework for C++ heterogeneous programming, the Qualcomm Symphony System Manager SDK[24, 23] for mobile SoCs. Section 3 described how the DML ensures correct concurrent access to a single buffer. However, a kernel k often needs to access multiple buffers $b1$, $b2$, etc. during its execution. In accordance with our use of the acquire-release model, k must acquire all its buffers upfront before it can begin execution. If any buffer acquire fails, all previously acquired buffers must be released, and the whole operation to acquire buffers for k must resume from scratch.

Since the *Acquire* and *Release* operations hold a buffer's lock only for the duration of that operation, there is never a situation where a thread in the SDK would acquire a lock for $b2$ while holding a lock for $b1$. Thus, deadlock is avoided. Unfortunately, kernels $k1$ and $k2$ may acquire overlapping buffers in different orders, creating the possibility of livelock. The SDK avoids livelock by having each kernel acquire and release its buffers in a canonical order — the sorted order of the pointers to the DML's buffer-state objects representing the buffers.

The SDK supports a number of existing heterogeneous compute and memory frameworks. The SDK has *execution back-ends* that wrap the compute APIs of OpenCL, OpenGL and proprietary custom devices to launch kernels, while delegating the data allocation and synchronization aspects to the DML. Due to the DML, the SDK exposes very high-level programming APIs with an abstract representation for program data. As shown in the sample SDK program in

Figure 3, there are no explicit calls to allocate or synchronize data. In fact, the creation of buf_c does not imply any storage allocation for it. buf_a and buf_b illustrate the use of an externally-allocated MainMem-arena for each of them. However, buf_c only needs to allocate storage when accessed by a kernel. The execution of t1 requires OpenCL-arenas to be created under the buffer-states for buf_a, buf_b and buf_c, and OpenCL storage to be allocated inside them. Due to cross-arena optimizations, buf_a and buf_b bind-allocate their OpenCL arenas to the corresponding MainMem-arenas. The SDK accepted "program structure" hints for buf_c, indicating future access on the GPU and DSP. Therefore, the SDK relies on the program structure information to first create an ION-arena inside buf_c's buffer-state, so that the subsequent launch of t1 would cause the OpenCL-arena to bind-allocate to the ION-arena. Without the SDK's initial transformation of buf_c's buffer-state to first allocate ION-storage, the DML would allocate arena storage in the order of use by kernels — allocating independent storage in the OpenCL-, ION-, and MainMem-arenas, and produce two unnecessary memory-copies at the launch of t2 and t3. The SDK also provides "programming pattern" constructs, such as heterogeneous pipelines and parallel-fors, which provide a much more precise order of device access than the hints, allowing the SDK to perform additional buffer-state transformations.

5 Experimental Evaluation

We run experiments on a Qualcomm Snapdragon$^{\text{TM}}$ mobile SoC [21] with multi-core CPUs and an integrated GPU and DSP. We port the following applications to the SDK, and benchmark against hand-optimized OpenCL versions.

1. **Vector-Add**: Adds two 1-million element vectors on the GPU.
2. **Matrix-Multiply**: Multiplies two 512 x 512-element matrices on the GPU.
3. **BFS** from the Rodinia benchmark suite [5]. Performs a breadth first search on a graph of 1 million nodes. Executes one kernel once on the GPU and a second kernel in a loop on the GPU.
4. **SLAMBench** [15] solves the Simultaneous Localization and Mapping (SLAM) problem using the KinectFusion algorithm [16]. The algorithm consists of 9 kernels executed within multiple levels of loops. We use SLAMBench's reference data-set *living_room_traj2_loop.raw* consisting of 880 image frames.

Vector-Add and Matrix-Multiple execute a single kernel on the GPU once. The CPU creates the inputs and reads back the result to verify. BFS and SLAMBench are real-world applications that invoke multiple kernels repeatedly. There is repeated synchronization required between CPU steps and GPU kernels in each loop iteration. We also extend SLAMBench to execute the "mm2meters" kernel on the DSP. We count the number of OpenCL API calls in the SDK and hand-optimized versions — clEnqueue{Map, Unmap, ReadBuffer, WriteBuffer, NDRangeKernel}. We also compare execution times, averaged over 5 runs.

The hand-optimized implementations of the applications optimize performance and minimize use of the OpenCL calls based on the programmer's knowledge

Table 1. Comparison of operation counts and execution time.

Call	**Vector-Add**			**Matrix-Multiply**			**BFS**		
	hand	*noopt*	*opt*	*hand*	*noopt*	*opt*	*hand*	*noopt*	*opt*
CL map	1	3	1	0	3	0	0	20	0
CL unmap	0	2	0	0	2	0	0	18	0
CL read	0	0	0	1	0	1	13	0	13
CL write	2	0	2	2	0	2	18	0	18
CL kernel	1	1	1	1	1	1	24	24	24
alloc	-	6	3	-	6	3	-	14	7
bind	-	-	3	-	-	3	-	-	7
memcpy	-	3	0	-	3	0	-	31	0
Time (ms)	31.74	41.20	32.44	1011	1023	1015	149.57	156.31	154.54

Table 2. Comparison of operation counts and execution time for SLAMBench.

Call	**GPU**			**GPU-DSP**		**GPU-CPU**	
	hand	*noopt*	*opt*	*noopt*	*opt*	*noopt*	*opt*
CL map	0	41480	12066	41480	12945	12314	5280
CL unmap	0	41479	12065	41479	12944	12308	5274
CL read	12066	0	0	0	0	0	1752
CL write	29414	0	28534	0	28534	0	4402
CL kernel	34252	34252	34252	33372	33372	10120	10120
alloc	-	58844	29429	57965	29429	10587	5297
bind	-	-	29415	-	28536	-	5290
memcpy	-	41480	0	41480	0	12314	0
Time (ms)	100447	109357	105356	119538	114187	112818	108484

of the program structure and data access patterns. The SDK hides the low-level details from the programmer, while being conservative to ensure correct data synchronization at device kernel execution boundaries. We evaluate the SDK in two modes: (1) *non-optimized*, where all storage allocation and data synchronization decisions are defered to our proposed DML, which chooses the appropriate actions in the dynamic order of the kernel access requests received, and (2) *optimized* where the SDK plugs cross-arena optimizations into the DML and manipulates the buffer-state representation to force optimal storage allocation decisions based on advance knowledge of the program structure (Section 4).

Table 1 shows the count of CL operations and execution time for the hand-optimized OpenCL implementations (*hand*) against implementations using the SDK– both non-optimized (*noopt*) and optimized (*opt*). We also compare the number of storage allocations (*alloc*), the number of bind-allocations across arenas in a buffer-state (*bind*), and the number of memory-copies for data synchronization (*memcpy*). Vector-Add and Matrix-Multiply create two input buffers and one output buffer. Therefore, *hand* incurs two CL write operations to synchronize the inputs to the GPU, and one CL map or CL read to read back the GPU result. *noopt* allocates separate main-memory and GPU-memory storage for each buffer, resulting in an explicit mem-copy between the two allocated storages of the buffer, and a CL map or unmap. In contrast, in *opt* the SDK uses cross-arena optimizations to bind-allocate, creating a single storage per buffer and avoiding mem-copies. Therefore, *opt* matches both the CL operation count and the execution time of *hand*. The results for BFS show a similar trend.

Table 2 compares multiple implementations of the SLAMBench application: (1) hand-written OpenCL (*GPU-hand*), where all the kernels execute on the GPU, (2) SDK*GPU*, where all kernels execute on the GPU, (3) SDK*GPU-DSP*, where all kernels execute on the GPU except the "mm2meters" kernel which executes on the DSP, and (4) SDK*GPU-CPU*, which executes the "track" and "reduce" kernels on the CPU and the rest on the GPU. Each SDK implementation is evaluated in non-optimized (*noopt*) and optimized (*opt*) modes. Table 2 shows that *GPU-hand* executes the fastest, with *GPU-opt* within 5%. SDK *opt* versions show a substantially reduced number of OpenCL calls, storage allocations and mem-copies compared to *noopt*, resulting in execution time gains that range between 3.8 and 4.7%.

Overall, these results demonstrate that (1) the SDK does not substantially increase execution time over the hand-optimized implementations, successfully bounding the overheads of the DML, and (2) the DML abstractions allows the SDK to correctly and easily use platform-specific and program-structure knowledge to achieve significant performance improvements. Using the data management techniques presented in this paper and the SDK's high level abstractions we make it easy for programmers to get performance within 5-10% of hand optimized with much less effort.

6 Related Work

Distributed Shared Memory(DSM) is an abstraction that provides a shared address space over a multitude of, potentially heterogeneous, computing nodes. DSM enables the free sharing of pointers across the different nodes, ensuring a common logical memory space for accessing data. Several ideas have been proposed and implemented to improve the performance of DSM, such as reducing communication overhead by sending only the updates to a page as opposed to the entire page, and lazy release consistency—delaying the propagation of changes until the page is acquired by a node [12]. These systems address issues in which nodes are loosely connected and the latency of communication is high. InterWeave [6] assumes a distributed collection of servers and clients. Servers maintain persistent copies of shared data and coordinate sharing among clients by mapping cached copies of needed data into client local memory. The unit of sharing is a segment, with addresses represented as URLs. While such an approach works for large clusters of computers, it is too heavy for mobile SoCs.

Asymmetric DSM(ADSM) [9] is a programming model that implements a specialized DSM over a heterogeneous computing system, with CPUs and other accelerators. By allowing unidirectional CPU access to accelerator memory (the accelerator does not have access to the CPU memory), the overheads associated with ensuring coherency are significantly reduced.

Liu *et al.* describe a system that allows sharing of virtual memory between CPUs and accelerators [13]. Data transfer between the CPU and accelerator memories is achieved through an explicit communication buffer maintained by the runtime. The synchronization of data across the different memories is performed by the runtime, and programmers do not have finer control. Our system differs from this approach, in that we do not enforce a shared virtual address space.

A number of heterogeneous tasking models have proposed various mechanisms to address memory management: OmpSs [8] extends OpenMP [19] to enable concurrent execution of tasks on heterogeneous devices. Data transfers are inserted by the compiler, which performs array slice analysis to determine the necessary data movements. The programmer is responsible to define the read/write sets for the heterogeneous tasks. StarPu [4] is a task based heterogeneous runtime that allows the user to specify where the memory needs to be allocated via an API call. Fluidicl [20] is an OpenCL runtime that enables a kernel execution to be partitioned across the CPU and GPU. A data merge step on the GPU combines the partial results. OpenACC [3] supports heterogeneous systems where CPU and accelerator either share memory, or have separate memories. A "data" construct takes care of synchronizing data between the host and device memories.

Heterogeneous System Architecture (HSA) [2] provides a unified address space between CPUs and GPUs by performing memory management in hardware. In terms of synchronization, it offers cross-device atomic operations. This model is similar to shared memory programming in multi-core CPUs, with some additional restrictions on the size of coherence units and memory regions available for sharing, but it is too low level and requires expert knowledge of the underlying hardware.

7 Conclusions

We have proposed a data management layer based on a novel abstract representation over heterogeneous memory. The use of the data management layer hides the considerable complexity and platform-dependence that programmers of heterogeneous applications currently face. We advocate a software architecture where a high-level programming framework, whether general-purpose or domain-specific, may rely on our data management layer to easily avoid dealing with the complexities of multiple memory frameworks and platform variations. The high-level frameworks may focus on domain and application-specific optimizations and provide simpler high-level programming abstractions to the users, while relying on the data management layer to provide correct synchronizaton of data, optimized to the platform and program properties, and enhance the concurrent sharing of data over the available heterogeneous devices. We incorporated our data management layer into the Qualcomm Symphony System Manager SDK, and demonstrated using both micro-benchmarks and large multi-kernel workloads that the benefits of high-level user abstractions, platform-portability and strong correctness properties are achieved with only minimal overheads to performance compared to writing hand-optimized OpenCL applications.

References

1. IvyTown Xeon + FPGA: The HARP program. Intel Corp.

2. HSA programmer's reference manual: HSAIL virtual ISA and programming model, compiler writer, and object format (BRIG). Technical report, HSA Foundation, July 2015.
3. The OpenACC: Application Programming Interface. Technical report, OpenACC-Standard.org, October 2015.
4. Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.
5. Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.
6. DeQing Chen, Sandhya Dwarkadas, Srinivasan Parthasarathy, Eduardo Pinheiro, and Michael L. Scott. InterWeave: A middleware system for distributed shared state. In *Selected Papers from the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, LCR '00, pages 207–220, London, UK, UK, 2000. Springer-Verlag.
7. Compute unified device architecture (CUDA). `http://www.nvidia.com/object/cuda_home_new.html`.
8. Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.
9. Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 347–358, New York, NY, USA, 2010. ACM.
10. Qi Hu, Nail A. Gumerov, and Ramani Duraiswami. Scalable fast multipole methods on distributed heterogeneous architectures. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 36:1–36:12, New York, NY, USA, 2011. ACM.
11. The Android ION memory allocator. `https://lwn.net/Articles/480055/`.
12. Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association.
13. Wei Liu, Brian Lewis, Xiaocheng Zhou, Hu Chen, Ying Gao, Shoumeng Yan, Sai Luo, and Bratin Saha. A balanced programming model for emerging heterogeneous multicore systems. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.
14. Sparsh Mittal and Jeffrey S. Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Computing Surveys*, 47(4):1–35, 2015.
15. L. Nardi, B. Bodin, M.Z. Zia, J. Mawer, A. Nisbet, P.H.J. Kelly, A.J. Davison, M. Lujan, M.F.P. O'Boyle, G. Riley, N. Topham, and S. Furber. Introducing SLAMBench, a performance and accuracy benchmarking methodology for slam. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 5783–5790, May 2015.

16. R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. In *Mixed and Augmented Reality (ISMAR), 2011 10th IEEE International Symposium on*, pages 127–136, Oct 2011.
17. Norman P. Jouppi *et al.* In-datacenter performance analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.
18. OpenCL, The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl`.
19. The OpenMP API specification for parallel programming. `http://www.openmp.org/`.
20. Prasanna Pandit and R Govindarajan. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 273. ACM, 2014.
21. Qualcomm Snapdragon. Qualcomm Technologies Inc. `https://www.qualcomm.com/products/snapdragon`.
22. Dave Shreiner and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Professional, 7th edition, 2009.
23. Heterogeneous computing made simpler with the Symphony SDK. `https://developer.qualcomm.com/blog/heterogeneous-computing-made-simpler-symphony-sdk`.
24. Qualcomm Symphony System Manager SDK. `https://developer.qualcomm.com/software/symphony-system-manager-sdk`.