Data Coherence Analysis and Optimization for Heterogenous Computing

Rafael Sousa and Marcio Pereira
Institute of Computing
UNICAMP
Email: {rafael.sousa,mpereira}@ic.unicamp.br

Fernando Magno Quintão Pereira
Department of Computer Science
UFMG
Email: fernando@dcc.ufmg.br

Guido Araujo
Institute of Computing
UNICAMP
Email: guido@ic.unicamp.br

***Abstract*—Although heterogeneous computing has enabled impressive program speed-ups, knowledge about the architecture of the target device is still critical to reap full hardware benefits. Programming such architectures is complex and is usually done by means of specialized languages (e.g. CUDA, OpenCL). The cost of moving and keeping host/device data coherent may easily eliminate any performance gains achieved by acceleration. Although this problem has been extensively studied for multicore architectures and was recently tackled in discrete GPUs through CUDA8, no generic solution exists for integrated CPU/GPUs architectures like those found in mobile devices (e.g. ARM Mali). This paper proposes Data Coherence Analysis (DCA), a set of two data-flow analyses that determine how variables are used by host/device at each program point. It also introduces Data Coherence Optimization (DCO), a code optimization technique that uses DCA information to: (a) allocate OpenCL shared buffers between host and devices; and (b) insert appropriate OpenCL function calls into program points so as to minimize the number of data coherence operations. DCO was implemented in AClang LLVM (www.aclang.org) a compiler capable of translating OpenMP 4.X annotated loops to OpenCL kernels, thus hiding the complexity of directly programming in OpenCL. Experimental results using DCA and DCO in AClang to compile programs from the Parboil, Polybench and Rodinia benchmarks reveal performance speed-ups of up to *5.25*x on an Exynos 8890 Octacore CPU with ARM Mali-T880 MP12 GPU and up to *2.03*x on a 2.4 GHz dual-core Intel Core i5 processor equipped with an Intel Iris GPU unit.**

I. Introduction

With the advent of heterogeneous computing many parallel programming models have emerged seeking to leverage the performance of sequential code by *offloading* computation *kernels* from a *host* machine (e.g. CPU) to an acceleration device (e.g. GPU). Computation offloading is typically achieved by annotating program fragments (e.g. hot loops) so that their execution is mapped to dedicated hardware like GPUs, APUs, FPGAs, among others. Most of these models use source code annotation standards like OpenACC and OpenMP or specialized language and libraries as in CUDA and OpenCL respectively. While they differ in the way the kernel code is written, all such models require data to be offloaded to the device and the result of the computation brought back to the host.

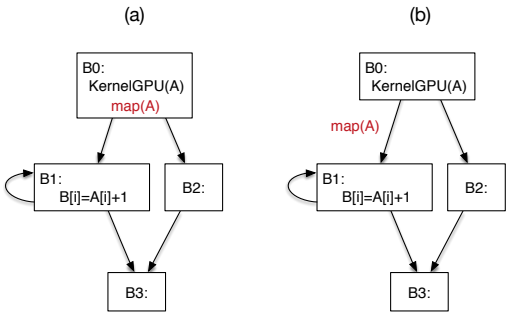
The task of offloading a kernel into an acceleration device can have a large impact on the overall program performance, particularly when the time required to move data in/out of the device approaches the time needed to perform the actual computation [6], [22]. This is a common problem in a *discrete GPUs*, where host and device do not share the same memory and data has to move through an interface card (e.g. PCI). On the other hand, even if host and device share the same memory, as in the case of *integrated GPUs*, coherence must be assured for shared data so as to avoid inconsistence between the computation in both sides.

Although there has been a number of efforts to address data coherence across host-device boundaries [7], [8] no universal hardware coherence protocol standard has yet been defined for heterogeneous systems. For the case of discrete GPUs, efforts are undergoing by NVIDIA in CUDA 8 that allow coherence between CPU and GPU through a page-fault mechanism [1]. For the case of integrated GPUs, coherence is performed in software by means of specific `map`/`unmap` operations or function calls that copy variables modified by the device/host back to the host/device thus squashing any old copies of the data that they might be holding. This papers addresses the problem of data-coherence optimization for integrated GPUs.

Coherence function calls are typically inserted by the programmer using functions from specialized libraries (e.g. OpenCL) or by a compiler that naively inserts such calls at the entry/exit of the kernels. It is important to highlight that a non-optimal insertion of `map`/`unmap` calls can result in unnecessary coherence operations thus impacting the overall program performance. Code optimization techniques are thus needed in order to minimize the insertion of coherence calls. To achieve that optimizing compilers targeting heterogeneous systems should identify variables that can be allocated in shared memory and perform code transformations that: (a) make these variables shared between CPU and GPU; (b) automatically avoid data movement of shared variables; and (c) keep the data used by host and device coherent.

Finding the best locations to insert `map`/`unmap` calls into source code can be cast as the *Data Coherence Optimization* (DCO) problem and involves: (1) identifying the blocks of code where shared variables are used by different devices (e.g. CPU or GPU) and (2) inserting `map`/`unmap` calls to minimize the need of data coherence operations among host and devices. Since coherence and data offloading impact program performance, these problems are inter-dependent

Fig. 1: Inserting `map` instruction to keep array `a` coherent between CPU and GPU.



and should be addressed together.

The *Control-Flow Graph* (CFG) of Figures 1a – 1b illustrates the need for DCO. For simplicity, this paper will consider an integrated CPU host and a GPU acceleration device. However, the ideas discussed herein can also be applied to discrete GPUs (e.g. NVIDIA). In Figure 1a basic block `B0` dispatches and executes kernel `KernelGPU` which modifies a shared array `A`. To keep `A` coherent with the CPU host, a non-expert programmer could insert a `map(A)` call at the end of `B0` as shown in Figure 1a. Furthermore, a naive compiler could insert a `map` call at the end `B0` to assure data coherence. This will make the GPU update its copy of `A` after `KernelGPU` finishes and the flow of execution goes through `B1`. On the other hand, if the execution takes the program through `B2` array `A` is not accessed and the cost of performing data coherence becomes an overhead. To avoid that, the programmer or a naive compiler should have inserted the `map` instruction on the edge that connects `B0` to `B1`, instead of inserting it at the end of `B0`. To address this problem, this paper makes the following contributions.

First, it proposes *Data Coherence Analysis*, a set of two data-flow analysis algorithms: (a) *Memory Usage Analysis* (MUA) - used to determine which kind of operation is done in a given variable; and (b) *Device Memory Analysis* (DMA) - used to track which device does this operation. Both algorithms propagate their informations through each program point.

Second, it introduces *Data Coherence Optimization* (DCO) that leverages DCA to insert OpenCL function calls `map` and `unmap` into program points so as to minimize the amount of data coherence operations required between host and device. DCO also replaces standard host memory allocation mechanisms (e.g. `malloc` and `calloc`) for specialized OpenCL shared buffer allocation calls.

The rest of the paper is organized as follows. Section II details the costs of data offloading and coherence operations in a typical heterogeneous platform. Section III introduces the two data-flow analyses of DCA and their mathematical formulation. Section IV discusses the implementation details of the corresponding LLVM optimization pass that implements a solution to DCO. Section V presents an overview of the *AClang* compiler, a LLVM based tool capable of automatically translating OpenMP 4.X annotated loops to OpenCL kernels, The experimental evaluation is described in Section VI. Section VII discusses related work and Section VIII concludes the work.

II. Background

Heterogeneous computing has shown that specialized acceleration devices (e.g. GPUs) can provide significant performance improvement for a range of applications [23]. However, knowledge about the architecture of the targeted device is critical to reap the full benefits of its specialized hardware. For instance, programming a CPU/GPU platform is made difficult by the subtleties required for a correct access to the shared memory between them. Fortunately specialized high-level languages (e.g. CUDA) and libraries (e.g. OpenCL) provide function calls to help with this task, though the programmer still needs to properly insert and use such calls.

OpenMP has been extensively used as a parallel programming standard for multicore architectures [24]. In order to extend OpenMP ability to program heterogeneous devices the OpenMP Accelerator Model [5] has been recently released; it adds to the standard a new set of clauses to deal with such architectures. By using these clauses one could synthesize OpenCL kernels from C/C++ OpenMP code. This approach enables programmers to leverage on OpenMP's easy of programmability and tap on OpenCL nice heterogeneous capabilities. The AClang compiler used in this work implements such translation process (Section V).

In order to reduce the offloading/coherence costs, modern integrated CPU/GPU architectures use shared global memory and try as much as possible to minimize the data movement between CPU and GPU. This is particularly critical in highly constrained architectures like those found in mobile architectures (e.g. ARM7/Mali) which need to minimize as much as possible the amount of energy consumed by the device. In such cases, useless data-movements between CPU and GPU represent an unacceptable overhead.

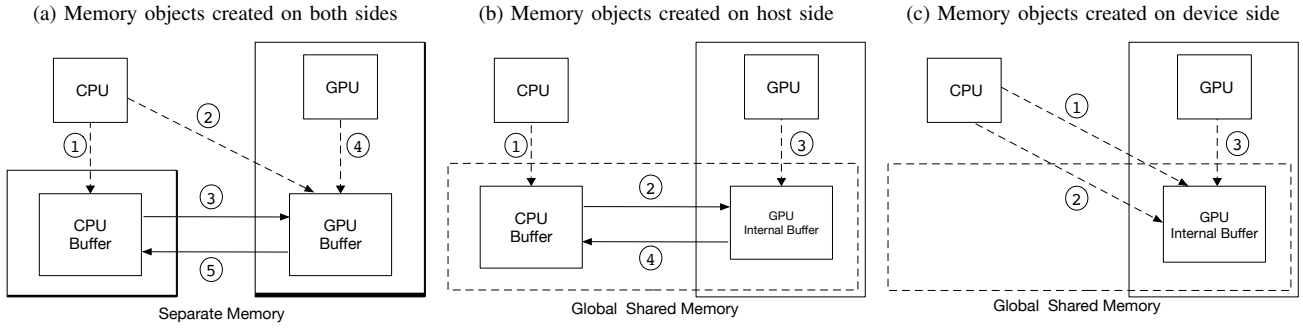
A. OpenCL Data Offloading/Coherence

This section discusses the main data-structures and functions calls required for offloading/coherence during the execution of an OpenCL kernel on a CPU/GPU platform.

Figures 2a – 2c show the typical ways in which host and devices use OpenCL buffers to communicate. In the Figures, dashed lines represent host/device actions on the buffers and full lines are data movement operations.

Separate Host/Device Memories: Figure 2a shows the kernel execution flow when host and device do not share the memory, a typical scenario when a GPU device has a dedicated memory and the data must be moved through an interface card to/from the host memory. First, a memory allocation routine (e.g. `malloc`) is called to create buffers in the host memory to store the host data variables ①. Before dispatching the kernel to the device, the host must call `clCreateBuffer` to create

Fig. 2: Cost of data offloading and coherence in a CPU/GPU platform using: (a) Memory Objects created on both sides; (b) Memory object created on host and (c) Memory object created on device.



the GPU buffer in the device memory ②. The host also needs to offload the data in the CPU buffer to the GPU buffer ③. After all the input data has been offloaded, the host dispatches the kernel to operate on the GPU buffer ④. The output of the kernel is then copied back to the CPU buffer ⑤.

Shared Host/Device Memory: Figures 2b – 2c show buffering approaches when the host and device share the same global memory (the device memory is mapped on the global shared memory space). In Figure 2b, prior to calling `clCreateBuffer` the host allocates the shared buffer and initializes its memory locations ①. This allocation typically uses host runtime calls like `malloc` which do not have the data layout expected by the device. The driver copies the newly created buffer from the host shared memory into the device internal memory layout in order to speed-up the kernel access to it ②. Thus, at the end of this call only the device has a valid pointer to the buffer and can operate on it ③. The host can request the data back through a `clEnqueueMapBuffer` (`map` [1]) call. In this case data is automatically transferred to the host and remains there until an `unmap` call occurs ④. In Figure 2c, memory objects are created at the device memory by the `clCreateBuffer` ①. If the host needs to access the data on the buffer, it calls the `map` function ②. The `map` function transfers data ownership to the host and the device cannot access it until the host calls `clEnqueueUnMapMemObject` (`unmap` [2]) and releases the device to access it ③. By using this approach, data offload becomes unnecessary, since only one pointer to a buffer is shared between CPU and GPU. This approach is used by DCO (Section IV) to exchange a CPU `malloc` call for an OpenCL buffer allocation call. This creates a single shared buffer between CPU and GPU and thus eliminates the need to perform expensive offloading and coherence operations between them.

[1] `map` hands the shared buffer pointer from the device to the host. It also flushes into the shared buffer all the data modified by the device that sits in its internal memory or cache.

[2] `unmap` hands the shared buffer pointer from the host to the device. It also flushes into the shared buffer all the data modified by the host.

III. Data Coherence Analysis (DCA)

To perform the optimizations that we have introduced in the previous section, we resort to a set of two inter-procedural data-flow analyses that we call *Data Coherence Analysis*. When combined, these optimizations give us the necessary information to insert `map` and `unmap` calls into programs.

The first of these data-flow algorithms is called *Memory Usage Analysis* (MUA), and the second *Device Memory Analysis* (DMA). Both algorithms propagate information in a backwards fashion, similar to the classic liveness. In the rest of this section we describe them in greater detail.

We shall use a minimalistic language to explain our analyses. Its syntax is shown in Figure 3 (Left). Our language gives us syntax to allocate, read and write arrays. All these operations can be carried out in the CPU, or in the GPU; hence, it contains six different instructions. We emphasize that this simplistic language is for presentation only: the techniques that we evaluate in this paper can handle the full fledged syntax of OpenCL 2.2-3.

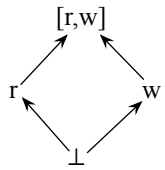
a) The Memory Usage Analysis (MUA): Figure 3 (Middle) shows the transfer functions for our Memory Usage Analysis. Our transfer functions work *per variable*. That is to say that they must run for each pointer-related array in the program. We use the expression *pointer related array* to denote a family of arrays which can alias each other, according to the results of any points-to analysis. The goal of MUA is to bind each array, at each program point, to an abstract state, which can be either $\perp$, r, w, or $[r, w]$. The former denotes arrays yet not visited by the data-flow resolution algorithm. If an array v is bound to the state r at a program point p, then there exists a path onto the program, from p till another point p', that does not cross any other usage of v, and that leads to an operation that reads that array. The state w assumes similar interpretation, except that we consider write operations. And the state $[r, w]$ indicates that the array can be read or written, depending on which path the program flow traverses. To keep track of abstract states, we associate with each array v, and each program point p, a pair of sets IN_u and OUT_u, which represent the

Fig. 3: (Left) The core language onto which we describe our data-flow analyses; (Middle) The transfer functions of the Memory Usage Analyses. (Right) The transfer functions of the Device Memory Analyses.

Core Language	**Memory Usage Analysis**	**Device Memory Analysis**
`c_bf(v)`; allocate array `v` in CPU	$IN_u[l] = OUT_u[l] \setminus \{(v, \_)\}$	$IN_d[l] = OUT_d[l] \setminus \{(v, \_)\}$
`g_bf(v)`; allocate array `v` in GPU	$IN_u[l] = OUT_u[l] \setminus \{(v, \_)\}$	$IN_d[l] = OUT_d[l] \setminus \{(v, \_)\}$
`c_st(v)`; write in array `v` on the CPU	$IN_u[l] = OUT_u[l] \setminus \{(v, \_)\} + \{(v, w)\}$	$IN_d[l] = OUT_d[l] \setminus \{(v, \_)\} + \{(v, c)\}$
`c_ld(v)`; read from array `v` on the CPU	$IN_u[l] = OUT_u[l] \setminus \{(v, \_)\} + \{(v, r)\}$	$IN_d[l] = OUT_d[l] \setminus \{(v, \_)\} + \{(v, c)\}$
`g_st(v)`; write in array `v` on the GPU	$IN_u[l] = OUT_u[l] \setminus \{(v, \_)\} + \{(v, w)\}$	$IN_d[l] = OUT_d[l] \setminus \{(v, \_)\} + \{(v, g)\}$
`g_ld(v)`; read from array `v` on the GPU	$IN_u[l] = OUT_u[l] \setminus \{(v, \_)\} + \{(v, r)\}$	$IN_d[l] = OUT_d[l] \setminus \{(v, \_)\} + \{(v, g)\}$

Fig. 4: (a) Join operation of MUA; (b) Join operation of DMA; (c) Initialization of each data-flow set in MUA and DMA.

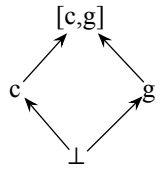
(a)



∨	⊥	r	w
⊥	⊥	r	w
r	r	r	r,w
w	w	r,w	w

$$OUT_u[\ell] = \bigvee IN_u[\ell_{succ}], \ell_{succ} \in \ell$$

(b)



∨	⊥	c	g
⊥	⊥	c	g
c	c	c	c,g
g	g	c,g	g

$$OUT_d[\ell] = \bigvee IN_d[\ell_{succ}], \ell_{succ} \in \ell$$

(c) **Initialization**:
$\forall \ell$, $IN_u[\ell] = \bot$, $OUT_u[\ell] = \bot$, $IN_d[\ell] = \bot$, $OUT_d[\ell] = \bot$

state of that array before and after p, respectively. MUA works on a height-two lattice, whose shape is outlined in Figure 4a. To solve MUA, we initialize each IN_u and OUT_u to $\bot$ (See Figure 4c), and iterate the resolution of a transfer function, until we reach a fixed point. Convergence is guaranteed, as Theorem III.1 states.

Theorem III.1 *MUA and DMA are guaranteed to terminate in $O(N)$, where N is the number of program points.*

Proof: transfer functions in Figure 3 are monotonic. Lattices in Figure 4 are finite, with height two. □

b) The Device Memory Analysis (DMA): This analysis is similar to MUA, except that it tracks the device in which each array is accessed. We assume two possibilities: CPU or GPU. To represent this information, we assign to each array one out of four possible abstract states: $\bot$, c, g, or $[c, g]$. As in MUA, $\bot$ denotes unknown information, and forms the bottom of DMA's lattice. The state c, when assigned to array `v` at point p, indicates the existence of a path from p to a point p' where `v` is accessed on the CPU. State g represents the same information, but for the GPU, and state $[c, g]$ indicates paths leading to accesses in the CPU or in the GPU, depending on the program flow. Like MUA, DMA is guaranteed to terminate in time proportional to the program size (see Theorem III.1). In other words, the abstract state of any array is allowed to change only twice; thus, after three visits, each array stabilizes. The lattice that determines how information evolves during this data-flow analysis is given in Figure 4b.

IV. Data Coherence Optimization (DCO)

The two analyses that we have seen in Section III gives us the necessary information to insert `map` or `unmap` directives in OpenCL programs. However, careless creation of such calls in the program might lead to redundancies, eventually downgrading performance. To avoid some redundancy, we shall describe a technique based on partial redundancy elimination [20] called *Latest Placement Analysis* (LPA). This analysis keeps track of program points where MAP or UNMAP directives are necessary, and propagates this information forwardly, until reaching a program point where such calls are no longer needed.

To ease the task of describing this analysis, we shall resort to three simplifications. First, we shall describe it only for UNMAP. To consider MAP, the reader must exchange the roles of CPU and GPU. Second, we will consider only one array `v`, to avoid having to mention variable names in our dataflow equations. Notice that in practice the analysis is applied on potentially many arrays at the same time. Third, we shall introduce new sets, IN_p and OUT_p, and shall adopt the following notational simplifications:

1) $x = r \vee w \vee [r, w]$;
2) $(\mathtt{v}, c) \in IN_d \wedge (\mathtt{v}, x) \in IN_u \Rightarrow (c, x) \in IN_p$ (resp. OUT_p);
3) $(\mathtt{v}, g) \in IN_d \wedge (\mathtt{v}, x) \in IN_u \Rightarrow (g, x) \in IN_p$ (resp. OUT_p).

Fig. 5: Dataflow equations for Latest Placement Analysis.

$$(g, x) \in IN_p \Rightarrow Latest \in OUT_p$$

$$\frac{Latest \in OUT_p \qquad (g, x) \in IN_p \qquad (c, x) \in IN_p}{Latest \in IN_p}$$

$$\frac{Latest \in IN_p \qquad (g, x) \in OUT_p \qquad (c, x) \in OUT_p}{Latest \in OUT_p}$$

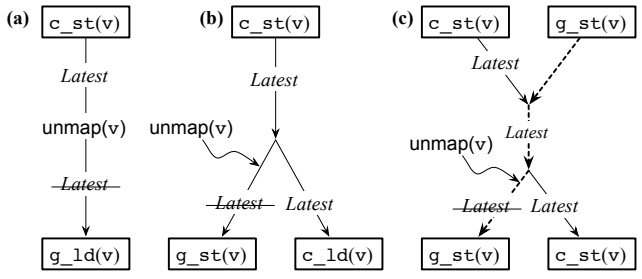
With these definitions, equations in Figure 5 define LPA.

The fixed point of the Equations in Figure 5 gives us a set of program points p with the following two properties: (i) p lays on a path from a point p_1, where the array is written by the CPU, to a point p_2, where it is accessed (read or written) by the GPU; and (ii) p also leads to a point p_3, where the array is accessed by the CPU itself. Thus, even though we need an `unmap` call at p, inserting it there could lead to potential redundancy, as this directive would also be in a path leading from CPU access to CPU access. To avoid such redundancy, we insert `unmap` only at *transition points*, i.e., points whose IN_p set contain *latest*, but whose OUT_p no longer does it:

$$\frac{p' \in succ(p) \qquad Latest \in OUT_p \qquad Latest \notin IN'_p}{\text{Insert } \texttt{unmap} \text{ between } p \text{ and } p'}$$

Figure 6 shows examples of programs requiring the `unmap` call. In Figures 6a – 6b, no redundant path exist: independent on the program flow, `unmap` is strictly necessary. Yet, it is still possible that LPA creates redundancy. Figure 6c illustrates this possibility. If the program flow goes from the `g_st(v)` towards the second `g_st(v)`, then it will traverse an unnecessary `unmap` call. We cannot remove this redundancy without replicating a whole path on the program – a task outside the scope of this paper. However, we emphasize that this redundancy does not compromise the semantics of the program.

Fig. 6: Examples of insertion of `unmap` calls. In (c), the dashed lines denote a path that still contains a redundancy.



V. The AClang Compiler

AClang [13] is an LLVM/Clang based compiler aimed at implementing the OpenMP Accelerator Model. It adds a new runtime library to LLVM/CLang that supports OpenMP offloading to devices like GPUs and FPGAs. Kernel functions are extracted from the OpenMP region and are dispatched as OpenCL code to be loaded and executed by OpenCL drivers.

The following example shows how *ACLang* works from a programmer perspective. Listing 1 presents two loops from `mvt` program of the Polybench [2] benchmark suite after they have been annotated with OpenMP 4 clauses. In the first loop the program computes the matrix vector multiplication followed by the transpose between `a` and `y1` storing the result into vector `x1`. The second loop does a similar task for `a`, `y2` and `x2`. As shown in Listing 1, the `target` clause defines the portion of the program that will be executed by the target device (i.e. GPU). The `map` clause details the mapping of the data between the host and the target device, i.e., the buffers marked in a `map` clause. For example, `a` and `y1` are two vectors that are offloaded from the CPU to the GPU memory before the two loops' kernels are sent to the device. This strategy offers maximal flexibility to the developer decide which part of the code is profitable to run on which architecture.

Listing 1: Fragment of Polybench mvt. benchmark

```
void mvt_gpu(float* a, float* x1, float* x2,
                       float* y1, float* y2) {
  #pragma omp target device (GPU)                    \
                    map(to: a[:N*N], y1[:N], y2[:N]) \
                    map(tofrom: x1[:N], x2[:N])
  {
    #pragma omp parallel for
    for (int i=0; i<N; i++)
      for (int j=0; j<N; j++)
        x1[i] = x1[i] + a[i*N + j] * y1[j];

    #pragma omp parallel for
    for (int i=0; i<N; i++)
      for (int j=0; j<N; j++)
        x2[i] = x2[i] + a[j*N + i] * y2[j];
  }
}
```

VI. Experimental Evaluation

AClang with DCO has been evaluated using two integrated CPU-GPU architectures: (a) a mobile Exynos 8890 Octa-core CPU (4x2.3 GHz Mongoose & 4x1.6 GHz Cortex-A53) integrated with an ARM Mali-T880 MP12 GPU (12x650 Mhz) running Android OS, v6.0 (Marshmallow); and (b) a laptop with 2.4 GHz dual-core Intel Core i5 processor integrated with an Intel Iris GPU with 40 execution units. The results presented in all experiments are averaged over ten executions. Variance is negligible; hence, we will not provide error intervals. The experiments use a set of programs from the Polybench [2], Parboil [3] and Rodinia [4] benchmarks with standard input sizes. The programs have been re-written in OpenMP 4. We chose only programs that could benefit from the proposed techniques. All experiments have used the tilling optimization available in AClang, and comparisons were performed by comparing the results of CPU and GPU executions.

A. DCO performance analysis

One of the claims of this paper is that DCO brings an improvement over the regular data offloading/coherence mechanism. To evaluate that, the AClang runtime library was instrumented to measure the percentage of the total program execution time corresponding to each one of the following tasks represented as bars in Figures 7a – 7d: (a) kernel computation (`Kernel` bar); (b) OpenCL driver tasks like context creation, queue management, kernel objects creation and GPU dispatch (`OpenCL` bar); and (c) kernel data offloading/coherence (`Offloading` bar). As shown in Figures 7a – 7b the `Offloading` bar is a major component of the total kernel execution time before DCO is applied; for example approximately 40% of the total execution time of *3dconv* on the Intel/Iris architecture is spent on offloading data and maintaining coherence.

Figures 7c – 7d show the results after applying DCO to the *Polybench*, *Rodinia*, and *Parboil* programs. As shown in the figure, after DCO inserts OpenCL `map`/`unmap` calls into the proper program points almost all data offloading and coherence overhead (`Offloading` bar) is removed from the programs.

Figures 7a – 7b reveal that the `OpenCL` driver takes an astonishing share of the total execution time in the majority of the tested programs (`OpenCL` bar). This effect is more pronounced in the ARM/Mali architecture meaning that the OpenCL driver used in this architecture needs some performance improvement. Also, Figures 7a – 7c and Figures 7b – 7d show that DCO can remove most of the program offloading overhead. Nevertheless, in some cases as in *bfs*, *hotspot* and *spmv*, the map/unmap calls need to be inserted within loops what account for the purple slices in the bars of the (Figures 7c – 7d).

Figures 8a – 8b show the speed-up of DCO optimization with respect to the original AClang (both compiled with -O3) to the Polybench, Parboil, and Rodinia programs, for Intel/Iris and ARM/Mali architectures. The benefit of DCO becomes clear as the complexity of the algorithms and the sizes of the data sets increase. For example, when"Gram-Schmidt decomposition" (*gramschmidt*) is compiled with the DCO optimization on the final code results in a speed-up of *5.3x* on ARM/Mali. In the non-optimized execution the kernel functions are extracted from the inner loops and the offloading is executed repeatedly, what does not occur in the DCO optimized version. We conclude that programs that create buffers more than once, as occur during the execution of *gramschmidt*, generate unnecessary data movement overhead. DCO can detect this situation and take advantage of it, since the buffers are created only once and used throughout the program.

A new experiment using Polybench applications on ARM/Mali was performed to evaluate how DCO impacts programs resulting from automatic annotation. DawnCC [14] [15] is a tool that automatically inserts OpenMP's 4.X annotations to parallelize code. Figure 9 shows the speed-up achieved when using AClang on the program annotated with DawnCC for two scenarios: with and without DCO. As shown in the figure, AClang using DCO improves performance for the majority of programs when compared with the optimization off. After a careful analysis we noticed that DCO was capable of eliminating additional data offloading operations introduced by automatic DawnCC annotation.

VII. Related Work

Previous work have shown that sharing host/device buffers in a shared memory *integrated CPU-GPU* can considerably improve program performance when comparing to a separate CPU-GPU architectures. Nilakant *et al.* [16] showed that using shared buffers in an integrated CPU-GPU outperforms by 15% to 50% the same application running on a separate CPU-GPU architecture. Backes *et al.* [17] showed a 30% improvement in the overall execution time when running the real-time image processing application on an integrated CPU-GPU architecture of a mobile device. Shen *et al.* [19] also reached good speed-ups by reducing more than 80% of the transfer time through an adequate usage of OpenCL memory flags. These works require the programmer to directly deal with the problem of sharing and making the data between CPU-GPU coherent. In AClang this task is automatically handled by the compiler.

Mendoa *et al.* [14] [15] developed a solution (DawnCC) to automatically insert OpenMP and OpenACC annotations into loops that expose parallelism. Their solution is capable of detecting the size of a given array, annotate code fragments for parallelization, and map them to be executed on a GPU. Even though DawnCC does a good job in coalescing offloaded data, AClang with DCO outperforms DawnCC when executing on a integrated GPU, since DCO uses a shared buffer approach, where memory movements are unnecessary.

Jablin *et al.* [11] proposed a solution called Dynamically Managed Data (DyManD), to automatically handle the data communication using a run-time library, without the need of any static analysis. Their solution creates an illusion of a buffer being shared between CPU and GPU; but, their library performs data offloading, since their memory allocator keeps equivalent allocation in both CPU and GPU. AClang with DCO avoids the automatic data transfer between CPU and GPU to ensure coherence, because only one buffer is created and shared among them.

Various approaches have tried before to raise the level of abstraction of OpenCL programming. Thouti *et al.* [21] proposed a methodology that takes function calls from C language and convert them to an equivalent OpenCL Kernel to be executed on GPU's devices. Unfortunately it only works on function calls. Thouti considered the usage of shared buffer between CPU and GPU, but they chose to use offloading.

To make GPU programming easy, Lee *et al.* [12] developed a source-to-source solution from OpenMP directives to CUDA. Their solution extracts marked regions

Fig. 7: The breakdown of total execution time: (a) & (b) before DCO optimization (c) & (d) after DCO optimization.

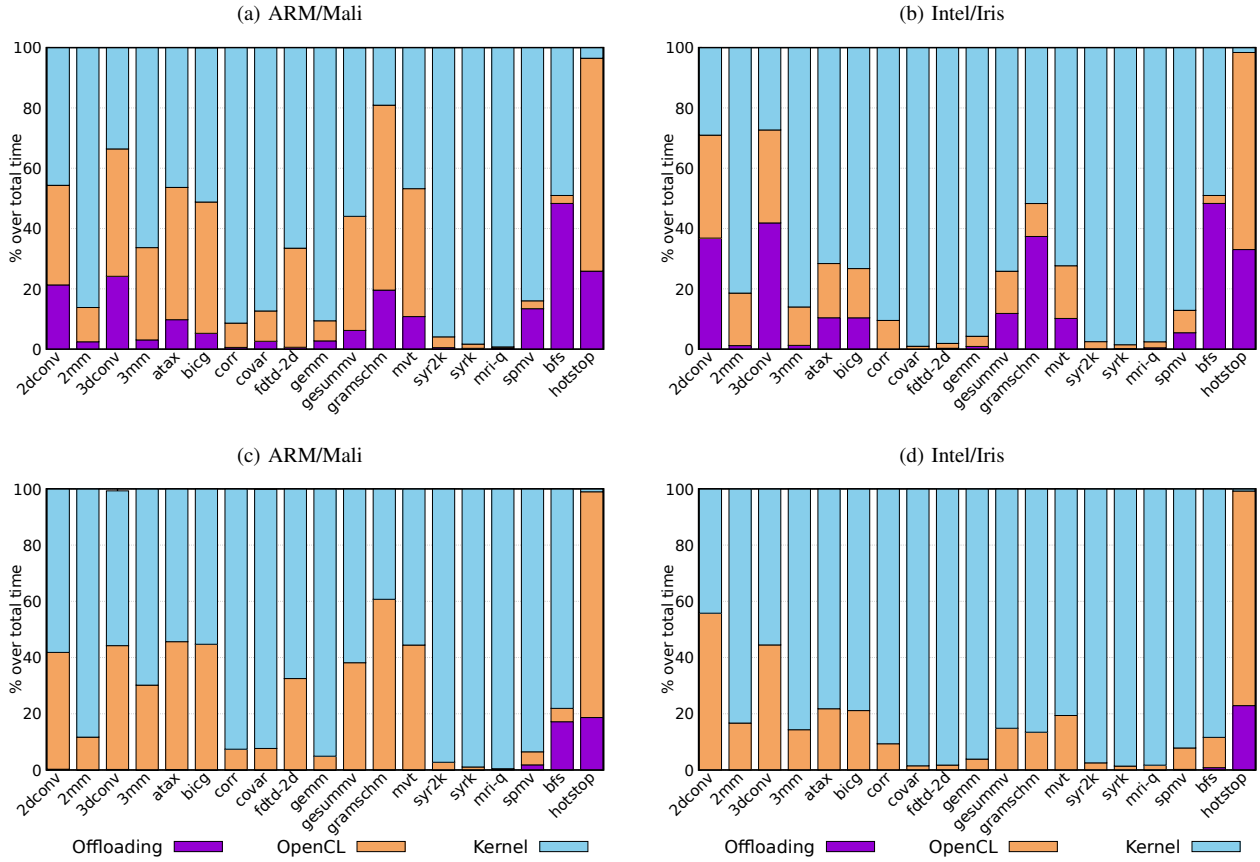
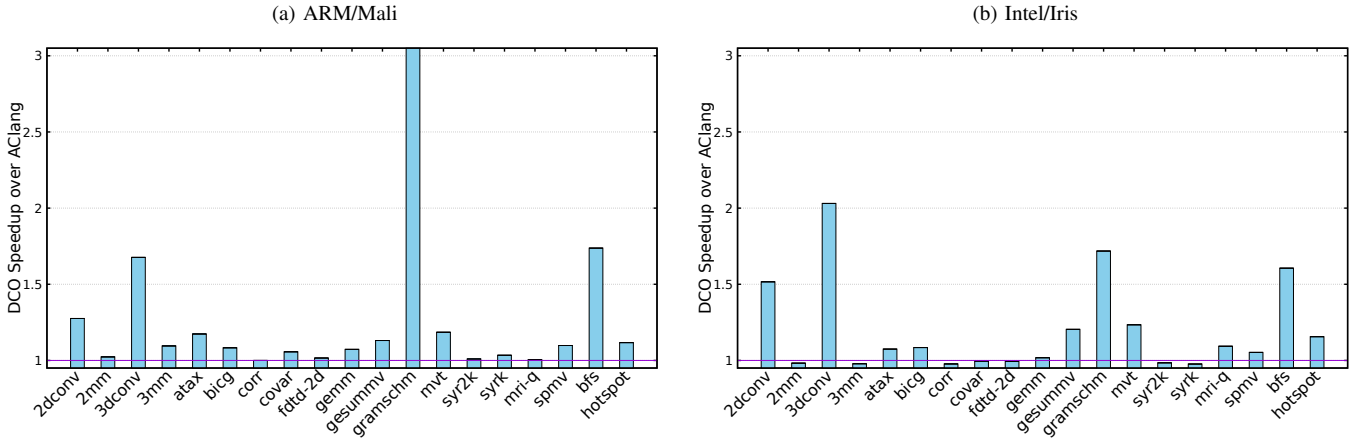


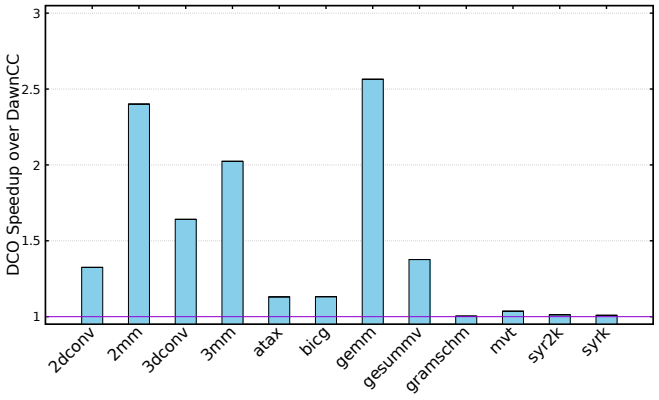
Fig. 8: DCO speed-ups with respect to *AClang*.



to be executed on GPUs. They developed an algorithm to reduce CPU-GPU memory transfer that only copies back the data modified inside the kernel region and used by the CPU afterwards. They also developed similar features as in [10]. Different from their solution, our work shares data buffers between CPU and GPU, without the need of memory transfers.

Said *et al.* [18] designed hiCL, an abstraction layer that was developed on top of OpenCL in order to reduce the programming burden, thus simplifying the memory management and the kernel execution. They developed functions to map buffers physically shared between CPU and GPU; however, their solution only abstracts the OpenCL complexity, leaving to the programmer the job of making the

Fig. 9: DCO speed-up upon automatic annotation.



coherence annotations in the code.

Some previous approaches proposed new architectural models for sharing and making data coherent between CPU and GPU. Gelado *et al.* developed a solution called Asymmetric Distributed Shared Memory (ADSM) [9]. Their architecture assures coherence between CPU-GPU by means of duplicated memory spaces in the host and the device. Thus memory copies are necessary to update both buffers. Furthermore, in their solution the programmer needs to manually assign coherence annotations.

VIII. Conclusion and Future Works

This paper described DCA, a set of dataflow analysis that has its root in the observation that making variables used by both CPU and GPU shared, one can avoid unnecessary data offloading. Moreover, it proposes DCO, an optimization that allows variable buffer sharing between CPU and GPU. Preliminary results show that DCO indeed improves the speedup of applications with large datasets, complex algorithms or medium-to-large kernel duration when running in integrated GPUs. For the future, we plan to assess the benefit of this technique on power reduction.

Acknowledgments

The authors would like to thank the anonymous reviewers for the insightful comments. This work is supported by Samsung (grant 4716.08) and FAPESP Center for Computational Engineering and Sciences (grant 13/08293-7).

References

[1] CUDA 8 Features Revealed https://devblogs.nvidia.com/parallelforall/cuda-8-features-revealed/

[2] PolyBench/GPU: Implementation of PolyBench codes for GPU processing http://web.cse.ohio-state.edu/~pouchet/software/polybench/GPU/

[3] Parboil Benchmarks http://impact.crhc.illinois.edu/parboil/parboil.aspx

[4] Rodinia:Accelerating Compute-Intensive Applications with Accelerators https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia:Accelerating_Compute-Intensive_Applications_with_Accelerators

[5] Liao, C., Yan, Y., Supinski, R., Quinlan, Daniel J., Chapman, Barbara. Early Experiences with the OpenMP Accelerator Model, *9th International Workshop on OpenMP*, Canberra, ACT, Australia, September 16-18, 2013

[6] Fujii, Yusuke and Azumi, Takuya and Nishio, Nobuhiko and Kato, Shinpei and Edahiro, Masato. Data Transfer Matters for GPU Computing. *Proceedings of the 2013 International Conference on Parallel and Distributed Systems*. pages 275–282. IEEE, 2013.

[7] Sinclair, Matthew D. and Alsop, Johnathan and Adve, Sarita V. Efficient GPU Synchronization Without Scopes: Saying No to Complex Consistency Models. *Proceedings of the 48th International Symposium on Microarchitecture – MICRO-48*, pages 647–659. ACM, 2015.

[8] Neha Agarwal, and David Nellans, and Eiman Ebrahimi, and Thomas F. Wenisch, and John Danskin, and Stephen W. Keckler. Selective GPU Caches to Eliminate CPU-GPU HW Cache Coherence. *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 494-506. IEEE, 2016

[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 *ACM SIGARCH Computer Architecture News*, volume 38, pages 347–358. ACM, 2010.

[10] Tianyi David Han and Tarek S Abdelrahman. hicuda: a High-level Directive-based Language for GPU Programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61. ACM, 2009.

[11] Thomas B Jablin, James A Jablin, Prakash Prabhu, Feng Liu, and David I August. Dynamically Managed Data for CPU-GPU Architectures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 165–174. ACM, 2012.

[12] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to gpgpu: A Compiler Framework for Automatic Translation and Optimization. *ACM Sigplan Notices*, 44(4):101–110, 2009.

[13] Marcio M Pereira, Rafael C F Sousa and Guido Araujo. Compiling and Optimizing OpenMP 4.X Programs to OpenCL and SPIR. *13th International Workshop on OpenMP (IWOMP 2017)*, 2017.

[14] Mendonça, Gleison and Guimarães, Breno and Alves, Péricles and Pereira, Márcio and Araújo, Guido and Pereira, Fernando Magno Quintão. DawnCC: Automatic Annotation for Data Parallelism and Offloading. *ACM Transactions on Architecture and Code Optimization (TACO)*, ACM, 2017.

[15] Mendonça, Gleison Souza Diniz and Guimaraes, Breno Campos Ferreira and Alves, Péricles Rafael Oliveira and Pereira, Fernando Magno Quintão and Pereira, Márcio Machado and Araújo, Guido. Automatic Insertion of Copy Annotation in Data-Parallel Programs *Computer Architecture and High Performance Computing (SBAC-PAD)*, 2016.

[16] Karthik Nilakant and Eiko Yoneki. On the Efficacy of APUs for Heterogeneous Graph Computation. In *Proc. 4th Workshop on Systems for Future Multicore Architectures (SFMA), Amsterdam, Netherlands*, pages 2–7, 2014.

[17] Luna Backes, Alejandro Rico, and Bjorn Franke. Experiences in Speeding up Computer Vision Applications on Mobile Computing Platforms. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)* pages 1–8, 2015.

[18] Issam Said, Pierre Fortin, Jean-Luc Lamotte, and Henri Calandra. hicl: An OpenCL Abstraction Layer for Scientific Computing, Application to Depth Imaging on GPU and APU. In *Proceedings of the 4th International Workshop on OpenCL*, page 14. ACM, 2016.

[19] Jie Shen, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. Performance Gaps Between OpenMP and OpenCL for Multi-core CPUs. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 116–125. IEEE, 2012.

[20] Knoop, Jens and Rüthing, Oliver and Steffen, Bernhard. Lazy Code Motion. In *PLDI*, pages 224–234. ACM, 1992.

[21] Krishnahari Thouti and SR Sathe. A Methodology for Translating C-programs to OpenCL. *International Journal of Computer Applications*, 82(3), 2013.

[22] Che, Shuai and Sheaffer, Jeremy W and Skadron, Kevin. Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2011.

[23] Ryoo, S., and Rodrigues, C. I., and Baghsorkhi, S. S., and Stone, S. S., and Kirk, D. B. and Hwu, W. W. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA. In *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming - PPoPP'*, pages 73–82. ACM, 2008.

[24] Terboven, C. and Mey, D., and Sarholz, S. OpenMP on Multicore Architectures. In *Proceedings of the 3rd International Workshop on OpenMP – IWOMP'07*, pages 54–64. Springer-Verlag, 2008.