Data-flow analysis and optimization for data coherence in heterogeneous architectures

Rafael Sousa a, *, Marcio Pereira a, Fernando Magno Quintão Pereira b, Guido Araujo a

a Institute of Computing, University of Campinas, UNICAMP, Brazil
b Computer Science Department, University of Minas Gerais, UFMG, Brazil

HIGHLIGHTS

• DCA: a set of two data-flow analyses that seek to identify CPU/GPU accesses.
• DCO: creates shared buffers between CPU/GPU and inserts calls to keep data coherence.
• A technique that tries to remove data offloading during GPU computation.
• Speed-up of up to 8.87x on representative benchmarks on integrated and discrete GPUs.

ABSTRACT

Although heterogeneous computing has enabled developers to achieve impressive program speed-ups, the cost of moving and keeping data coherent between host and device may easily eliminate any performance gains achieved by acceleration. To deal with this problem, this paper introduces DCA: a pair of two data-flow analyses that determine how variables are used by host/device at each program point. It also introduces 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. We have used the ACLang compiler to measure the impact of DCA and DCO when generating code from Parboil, Polybench and Rodinia benchmarks for a set of discrete/integrated GPUs. The experimental results showed speed-ups of up to 5.25x (average of 1.39x) on an ARM Mali-T880 and up to 8.87x (average of 1.66x) on an NVIDIA GPU Pascal Titan X.

© 2019 Elsevier Inc. All rights reserved.

1. 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. 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 [5,8]. This is a common problem in a discrete GPUs (e.g. NVIDIA T880), 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 (e.g. ARM Mali), coherence must be assured for shared data so as to avoid inconsistency between the computation in both sides. Integrated GPUs are commonly found in architectures with limited memory resources, and stringent design constraints such as smartphones.

Although there has been a number of efforts to automatically address data coherence across host-device boundaries [1,27], no universal hardware coherence protocol standard has yet been defined for heterogeneous systems. The exception is a recent interconnect architecture called NVLINK [7], proposed by NVIDIA, that uses CUDA8 to enable automatic coherence between CPU and GPU data by means of a page fault mechanism [6].
Outside the NVIDIA/CUDA domain, host-device coherence has no hardware support and needs to be addressed in software by the programmer/compiler. For the case of integrated GPUs, coherence is performed in software by means of specific by the programmer/compiler. For the case of integrated GPUs, no hardware support and needs to be addressed in software

![Diagram](Fig. 1) Inserting map instruction to keep array a coherent between CPU and GPU.

1. **Data Coherence Analysis (DCA)**, a pair of data-flow analyses: (a) Memory Usage Analysis (MUA) that determines which kind of operation is performed with a given variable; and (b) Device Memory Analysis (DMA) that tracks who computes the operation, host or device.

2. **Data Coherence Optimization (DCO)**, a technique that uses 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.

3. **Shared Buffer Allocation (SBA)**, an optimization that detects which data is allocated by the CPU and offloaded to/from an integrated GPU, and translates, whenever possible, CPU allocation calls (e.g. malloc) to OpenCL create buffer calls; the goal is to maximize the usage of the CPU/GPU shared memory, thus minimizing the need of coherence operations.

The first two contributions in the list above have been published in an earlier version of this work [28]. This current version extends that preliminary report with the idea of shared buffer allocation, plus an extensive experimental evaluation. In particular, we now bring a comparative analysis on how DCO performs when applied onto a newer integrated GPU with more computational power than the device we had available in 2017. In such experiments, we have compared the results of ARM/Mali-T880 with its successor ARM/Mali-G71. Furthermore, we discuss the impact of DCO when running on discrete GPUs. To this effect, we have evaluated our ideas on an NVIDIA Pascal Titan X; hence, showing how DCO improves computations on discrete GPUs through the use of pinned memory.

The rest of the paper is organized as follows. Section 2 details the costs of data offloading and coherence operations in a typical heterogeneous platform. Section 3 presents an overview of the AC lang compiler, a LLVM based tool capable of automatically translating OpenMP 4.X annotated loops to OpenCL kernels. Section 4 introduces two data-flow analyses that compose DCA (Data Coherence Analysis) and their mathematical formulation. Section 5 describes how DCA is used to design DCO (Data Coherence Optimization). DCO leverages on DCA to maximize host/device shared memory usage, while minimizes the need of coherence operations. Experimental evaluation is described in Section 6. Section 7 discusses related work and Section 8 concludes the work.

### 2. Background

Heterogeneous computing has shown that specialized acceleration devices (e.g. GPUs) can provide significant performance improvement for a range of applications [24]. 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 [30]. In order to extend OpenMP’s ability to program heterogeneous devices, the OpenMP Accelerator Model [15] has been conceived. This model extends the concept of offloading and enables the programmer to use dedicated directives to define offloading target regions that control data movement between host and devices. By using a new set of clauses to deal with such architectures, one can synthesize OpenCL kernels from C/C++ OpenMP code. This approach enables
programmers to leverage on OpenMP's ease of programmability and tap on OpenCL nice heterogeneous capabilities. The AClang compiler used in this work implements such translation process (Section 3).

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.

In discrete GPUs, such as NVIDIA GPUs, data coherence is performed through a page fault mechanism transparent to the CUDA8 software. By using the cudaMallocManaged allocation primitive, the driver handles all data coherence tasks required to transfer pages between the CPU and the GPU, when necessary. In summary, CUDA uses a Unified Virtual Memory (UVM) that makes a given memory buffer allocation visible to both CPU and GPU [6]. During kernel execution, if it is identified that the buffer is not available in GPU memory, the data is automatically transferred to the GPU so that the kernel can run in sequence. The same applies when running the code on the CPU side.

Still in relation to NVIDIA GPUs, some of the features available in CUDA are not available in OpenCL, such as the Unified Virtual Memory (UVM). Unlike an integrated GPU, which has a shared memory system between CPU and GPU, a discrete GPU contains its own memory system, separate from the CPU memory. Therefore, applying our approach to discrete GPUs results in different behavior. Memory allocation by the host (for example, through the malloc primitive) is paginated by default. Paged memory cannot be accessed directly from the GPU; therefore, when a data transfer is required, a temporary pinned memory (for example, a locked page) must be created. After that, the contents of the paged memory are transferred to the pinned page. Finally, the data can be transferred from the temporary pinned memory to the system memory of the GPU. These overheads can be avoided by using the proposed SBA optimization. SBA directly creates a pinned memory in the memory of the host system so that it is later accessed by the CPU and the GPU. In addition, this mechanism also allows data transfers asynchronously between CPU and GPU, resulting in increased bandwidth [19].

2.1. OpenCL data offloading/coherence

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

Host/device buffers. Memory objects form the most fundamental architectural unit in OpenCL programming. Creating buffer objects is simple in OpenCL and is akin to the way in which one would use C's memory allocation routines such as malloc or calloc. OpenCL provides function clCreateBuffer that creates memory objects based on a set of memory flags. These flags define the properties of the created memory objects and can assume the following values:

(a) \textbf{CL\_MEM\_READ\_WRITE}: the buffer is created in the device global memory and can be read and written by the kernel;

(b) \textbf{CL\_MEM\_USE\_HOST\_PTR}: the buffer to be created uses the memory referred by the host. The function does not allocate any memory at the device; instead, it enables the device to use an existing buffer allocated by the host. This is commonly used when the programmer wants to read the buffer created by the host, process the buffer in the device, and send the modified data back to the host; and

(c) \textbf{CL\_MEM\_ALLOC\_HOST\_PTR}: the buffer is allocated at the device memory and can be mapped to the host memory and accessed by it.

At this point, the reader could ask how the usages of these different buffer types affect the costs of offloading and coherence? Fig. 2a–c show the typical ways in which host and devices use OpenCL buffers to communicate. There are basically two ways of storing OpenCL buffers, in separate or shared memories. In the figures, dashed lines represent host/device actions on the buffers and full lines are data movement operations.

Separate host/device memories. Fig. 2a shows the offloading flow when host and device do not share the memory, a typical scenario when a discrete 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 the GPU buffer in the device memory 2. The host also needs to offload the data from 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.

Still in relation to Fig. 2a, when using OpenCL on discrete GPUs (for example, NVIDIA GPUs), one can use a paged buffer or a pinned buffer (via CL\_MEM\_ALLOC\_HOST\_PTR flag) to help in the offloading process. Even when using pinned memory, transfers are still required to update the data accessed by both CPU and GPU. To access the buffer created in pinned memory it is necessary to use map/unmap primitives. Unlike in pinned memory, paged memories are operated through read/write primitives (clEnqueueReadWriteBuffer). In such case, before offloading data to the GPU the AClang compiler uses pageable memory, which results in some overhead. When compiling code for discrete GPUs, the AClang compiler uses a pinned memory based DCO optimization described in Section 5.

Although the offloading flow described in Fig. 2a is typically used for discrete GPUs (since CPU and GPU have different memory systems), it can also be applied to integrated GPUs. However, when applied to an integrated GPU, such flow creates two buffers of the same size in shared memory when only one buffer is actually needed. This is an obvious disadvantage since it would be necessary to make one additional transfer operation between the two buffers when moving data between host and device.

Shared host/device memory. Fig. 2b–c 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 Fig. 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 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 4) call. In this case data is automatically transferred to the host and remains there until an unmap call occurs 4. In Fig. 2c, memory objects are created at the device memory by the clCreateBuffer 5. If the host needs to access the data on the buffer,\footnote{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.}
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 the SBA optimization (Section 5.1) to exchange a CPU call for an \( \text{GPU} \) 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.

On the correctness of our approach. The insertion of coherence calls in a program does not compromise its semantics: the program still executes correctly, even if we populate it with unnecessary annotations. However, naive implementation of coherence might lead to performance degradation. The wrong insertion of coherence calls (map/unmap) when mapping memory from discrete GPUs can cause very high latencies given that it generates remote accesses — impacting performance directly. The techniques that we describe in Section 5 try to mitigate the possibility of slowdowns. To this effect, we resort to an approach similar to those already seen in Partial Redundancy Elimination \([11]\) to reduce the chance that unnecessary coherence calls end up in the optimized program.

3. The AClang compiler

\( \text{AClang} \) \([21]\) 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 or SPIR code to be loaded and executed by OpenCL drivers.

AClang OpenCL runtime library has two main functionalities: (a) it hides the complexity of OpenCL code from the compiler; and (b) it provides a mapping from OpenMP directives to the OpenCL API, thus avoiding the need for device manufacturers to build specific OpenMP drivers for their GPUs or FPGAs.

The following example shows how AClang works from a programmer perspective. Listing 1 presents two loops from mvt program of the Polybench \([22]\) benchmark suite after they have been annotated with OpenMP 4 clauses. In the first loop, the program computes the matrix–vector multiplication between \( a \) and \( y \) storing the result into vector \( x \). The second loop does a similar task for \( a, y \) and \( x \), where the difference comes from the transpose applied to a before multiplying it with \( y \). 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, \( a \) is a vector that is 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. The OpenCL code generated by the AClang compiler for mvt is shown in the AClang website (www.aclang.org).

// Problem size
#define N 8192

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

Listing 1: Fragment of Polybench mvt. benchmark

Fig. 3 shows the AClang compiler pipeline. After generating the AST from the Source Code \( 1 \), AClang goes into two different flows to generate both OpenCL host-side and kernel codes. The host-side code is responsible for creating calls to perform: OpenCL buffer allocation, kernel invocation and data-offloading. To produce host-code, AClang follows the flow \( 2 \). After this step, at \( 3 \), AClang performs optimizations on the host-side code and generates the Host Binary \( 4 \). The generation of OpenCL Kernels follows flow \( 5 \). In the “OpenMP–OpenCL transformations”, AClang extracts annotated loops from the AST and synthesizes them to OpenCL kernels in source code format (the Polyhedral Optimizations are optional). The OpenCL kernels are generated, according to the flow \( 6 \), in source code format (as a \text{C} \ file). Optionally, we can produce kernel bit code in the SPIR format \([29]\).

In regard to the Polyhedral Optimizations, AClang leverages on ISL \([33]\) and polyhedral model optimizations \([4]\) to transform the extracted loops so that they can be tiled and mapped to the blocks and threads in the OpenCL kernel code. It implements a multilevel tiling strategy tailored to the multiple levels of parallelism and to the memory hierarchy of the target GPU. As an example, tiling can be directly applied to the loops of Listing 1, as outermost loops can be executed in parallel because their iterations update disjoint parts of the \( x1 \) and \( x2 \) arrays.
Our work was applied to the OpenCL host-side code generated by AClang compiler. In Fig. 3, DCO works as a Host Code Optimization, where it is applied as a set of LLVM passes that performs analyses and optimization on the LLVM Intermediate Representation (LLVM IR).

4. Data Coherence Analysis (DCA)

To perform the optimizations that we have introduced in the previous section, we resort to a pair of inter-procedural data-flow analyses. We shall call the combination of these two techniques the Data Coherence Analysis. This data-flow analysis gives 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 is called Device Memory Analysis (DMA). Both algorithms propagate information in a backwards fashion, similar to the classic liveness analysis. 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 Fig. 4 (Top). 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. Instructions that start with the c_ prefix denote operations that take place within the CPU. The g_ prefix, in turn, refers to the GPU. Instructions that end with the suffix _bf command the creation of buffers. Thus, for instance, the instruction c_bf (v) creates a buffer in the CPU, and assign it to pointer v. Notice that we shall not mention the size of the buffer. This information is not used by any of our analyses; hence, it is immaterial to the discussion that follows. The suffix _st denotes write operations (stores); and the suffix _ld denotes read operations (loads). As an example, the instruction c_st (v) writes some data in the buffer v. To be correct, v must be, indeed, in the address space of the CPU. If that is not the case, then v must be mapped there via a map call. We emphasize that this simplistic language is for presentation only: the techniques that we evaluate in this paper can handle the full syntax of OpenCL 2.2–3.

**The Memory Usage Analysis (MUA).** Fig. 4 (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 ⊥, 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 \(IN_v^p\) and \(OUT_v^p\), which represent the state of that array before and after p, respectively. MUA works on a height-two lattice, whose shape is outlined in Fig. 5a. To
solve MUA, we initialize each \( \text{IN}_0 \) and \( \text{OUT}_0 \) to \( \bot \) (See Fig. 5c), and iterate the resolution of a transfer function, until we reach a fixed point.

**Example 4.1.** Fig. 6 illustrates the application of the Memory Usage Analysis. We can see that this analysis implements a may fixed point algorithm: information is joined at meet points in the program's control flow graph, until saturation leads to termination, as abstract states stop changing.

Convergence of the equations used in the implementation of the Memory Usage Analysis is guaranteed, as Theorem 4.2 states. Furthermore, this analysis runs in quadratic time on the size of the program.

**Theorem 4.2.** MUA is guaranteed to terminate in \( O(N^2) \), where \( N \) is the number of program points.

**Proof.** Transfer functions in Fig. 4 are monotonic. Lattices in Fig. 5 are finite, with height two. Thus, the complexity of MUA is proportional to the number of sites where arrays are used, and the number of program points. Quadratic complexity emerges on the worst case because we might have a program formed as a complete directed graph, in which each node creates a new buffer. In this case, we might have \( N \) different definitions reaching each different program point. From this observation, we derive the \( O(N^2) \) worst case complexity. □

The Device Memory Analysis (DMA). As shown in Fig. 4 (Bottom), 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 square of the program size (after an argument similar to the reasoning used in Theorem 4.2). 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 Fig. 5b.

**Example 4.3.** Fig. 7 illustrates an application of the Device Memory Analysis. Notice how the meet operator joins the states \( c \) and \( g \) to form the state \( [c, g] \), at the program point \( \ell \), in the middle of the control flow graph. This state, e.g., \( (v, [c, g]) \), denotes the fact that the program point \( \ell \) can be reached by two different definitions of \( v \), at least one of these definitions comes from the address space of the CPU, and at least one more of them, in turn, happens in the address space of the GPU.

A dense analysis. The reader familiar with the theory and practice of data-flow analyses will notice that our implementation of both, MUA and DMA, is dense. A dense data-flow analysis associates each variable, at each program point, with an abstract state. Yet, these two techniques are very close to the well-known reaching definitions analysis, which, after the Static Single Assignment form, has an efficient sparse implementation [2]. A sparse analysis associates data-flow facts, e.g., \( x, w, c, \bot, \ldots \), with the names of variables. Usually, it has lower asymptotic complexity than its dense counterpart. Nevertheless, in our case, sparse data-flow analyses are hardly applicable. Our elements of interest are pointers. Static Single Assignment form is not used to represent memory locations, because while SSA-form has a side-effect free purely functional semantics, memory locations exist in a program exactly due to the mutable states that they enable. There are techniques to convert pointers to the SSA format, such as the Array Static Single Assignment (ASSA) form of Knobe and Sarkar [13]. However, ASSA was not available in mainstream compilers at the time of this submission.

5. Data Coherence Optimization (DCO)

The AClang compiler leverages on DCA to implement two optimization steps. The first step is named Shared Buffer Allocation (SBA) (Section 5.1) and seeks to maximize the usage of shared memory buffers between CPU and GPU. The second step gives the name of the optimization Data Coherence Optimization (DCO), and it seeks to minimize the insertion of the map and unmap function calls required to maintain data coherence between CPU and GPU. All the transformations performed by these two steps occur in the LLVM Intermediate Representation (IR) after AClang generates OpenCL code from the OpenMP annotations.

5.1. Shared Buffer Allocation (SBA)

Shared Buffer Allocation (SBA) is a technique that has two main goals: (a) to create, whenever possible, shared buffers between CPU and GPU so as to minimize the need of two variable
allocations, on the host and device sides; and (b) to remove
the need of OpenCL function calls (clEnqueueReadBuffer
and clEnqueueWriteBuffer) required to manage the buffers used for
data offloading.

SBA works by tracking for each GPU buffer (created by c1CreateBuffer) the corresponding CPU buffer (created by malloc or
calloc) that are used for data offloading. Once identified, these
buffers are removed and substituted by a single buffer that is
shared between the CPU and GPU. To do so, SBA first identifies
functions that perform data offloading. In Fig. 8a, both statements
s5 and s6 perform data offloading. For the purpose of simplicity
let us just analyze the s6. First, SBA analyzes s6 and captures
the two parameters of this OpenCL function (pointers d_A and h_A)
in order to identify the pointers to the CPU and GPU buffers. As
its next step, SBA uses ud-chain to identify the last definitions of
these buffers in program order. By applying ud-chain to d_A, SBA
detects that d_A is last defined at statement s4 by means of a call
to the OpenCL function c1CreateBuffer. A similar procedure
is performed with h_A which is allocated at statement s1 by
means of a call to malloc. The analysis performed by SBA is
an inter-procedural ud-chain algorithm that is executed recursively
until the buffer allocation calls are found for both CPU and GPU
pointers.

Since the last definition of both variables d_A and h_A are their
corresponding buffer allocation calls (respectively CPU and GPU),
SBA can allocate a single shared buffer for both CPU and GPU.
As shown in the CFG of Fig. 8b, SBA removes both statements s1
and s4 of Fig. 8a and uses a single c1CreateBuffer in statement
s1 of Fig. 8b set with flag CL_MEM_ALLOC_HOST_PTR to create a
single shared buffer that is pointed by dh_A. A similar procedure
is also performed by SBA on d_B and h_B at statement s2 of Fig. 8b.
Then the pointers for the KernelGPU call are re-written to dh_A
and dh_B.

Whenever possible, SBA tries to insert the calls to the functions
that create the CPU/GPU shared buffers to a program point
outside loop bodies. For example, in Fig. 8b the new shared
buffers were created in statements s1 and s2 of B0 in order to
avoid the loop in B1. This allows AClang to reduce unnecessary
overheads associated to multiple buffer creation calls.

To be conservative, SBA uses alias analysis to find all buffers
that a given pointer may point to. If for some particular pointer
it cannot prove that it is alias free, it inserts runtime checks
to evaluate the address pointed by this pointer before its use
to ensure the correct behavior of the program. This occurs, for
example, when the compiler does not have access to the source
code of a given library function that may modify the address
pointed by a pointer.

The two analyses that we have seen in Section 4 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 [11] 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; however, each application is independent. In
other words, our analysis is separable, following the terminology
created by Tavares et al. [2]. Third, we shall introduce new sets,
IN_p and OUT_p, and shall adopt the following notational
simplifications:

\[ (g, x) \in IN_p \Rightarrow Latest \in OUT_p \]

\[ Latest \in OUT_p \quad (g, x) \in IN_p \quad (c, x) \in IN_p \]

\[ Latest \in IN_p \]

\[ (g, x) \in OUT_p \quad (c, x) \in OUT_p \]

\[ Latest \in OUT_p \]

5.2. Data Coherence Optimization (DCO)

Fig. 9. Dataflow equations for Latest Placement Analysis.

With these definitions, equations in Fig. 9 define LPA. The fixed
point of the equations in Fig. 9 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:

\[ p' \in succ(p) \quad Latest \in OUT_p \quad Latest \notin IN_{p'} \]

Insert unmap between p and p'.

Fig. 10 shows examples of programs requiring the unmap call.
In Fig. 10a–b, no redundant path exists: independent on the
program flow, unmap is strictly necessary. Yet, it is still possible
that LPA creates partial redundancies. Fig. 10c 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.
Complexity analysis. DCO runs in $O(N^2)$ worst-case scenario, where $N$ is the number of program points, i.e., basic blocks. To see why such is the case, notice that the rules in Fig. 9 are applied per program point, and each program point might have a linear number of data-flow facts in the $IN_p$ sets. The size of these sets is proportional to the number of variables in the program, and we assume that each program point defines at most one variable.

6. Experimental evaluation

AClang with DCO has been evaluated using three integrated CPU–GPU architectures: (a) a mobile Exynos 8895 with an ARM Mali-G71 MP20 GPU (32 × 850 Mhz) running Android OS, v7.0 (Nougat); (b) a mobile Exynos 8890 Octa-core CPU (4 × 2.3 GHz, 4 × 1.6 GHz Cortex-A53) integrated with an ARM Mali-T880 MP12 GPU (12 × 650 Mhz) running Android OS, v6.0 (Marshmallow); and (c) a laptop with 2.4 GHz dual-core Intel Core i5 processor integrated with an Intel Iris GPU with 40 execution units. Besides that, AClang with DCO has also been evaluated using one discrete GPU: a processor Intel i7-4770 3.40 GHz with an NVIDIA GPU Pascal Titan X. 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 [22], Parboil [20] and Rodinia [23] benchmarks with standard input sizes. The entire Polybench benchmark suite has been re-written to use OpenMP 4.X annotations — the modifications only include the annotations. All experiments have used the loop tiling optimizations available in AClang, and comparisons were performed by comparing the results of CPU and GPU executions.

In this section, we have selected a small collection of benchmarks that benefit from data coherence. This collection includes most of the benchmarks in Polybench, two programs from Parboil: mri-q and spmv, and two programs from Rodinia: bfs and hotspot. These programs combine intensive computation with a high volume of data transfer between host and device.

Other programs from Parboil and Rodinia that could benefit from DCO were not evaluated given some limitations imposed by AClang and DCO. The main limitation comes from AClang when it fails to generate OpenCL code and then has to fallback the execution to CPU. Another limitation that occurs is when applying SBA. After AClang generate both OpenCL host-side and kernel codes, SBA tries to identify, for each GPU buffer (created by using clCreateBuffer), its correspondent CPU buffer (created by using, e.g., malloc). When SBA is not able to find it, SBA decides to use the data offloading version generated by AClang.

6.1. 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 Figs. 11a–11f: (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 Figs. 11a, 11c, 11e and 11g, before applying DCO the Offloading bar is a major component of the total kernel execution time; for example, approximately 40% of the total execution time of 3dconv on the Intel/Iris architecture is spent on offloading data and maintaining coherence.

Figs. 11b, 11d, 11f, and 11h show the results after applying DCO to the Polybench, Rodinia, and Parboil programs. After DCO inserts OpenCL map/unmap calls into the proper program points, in most cases almost all data offloading and coherence overhead (Offloading bar) is removed from the programs. As expected, this happens more effectively on integrated GPUs, although in discrete GPUs the overhead due to data offloading and coherence also suffers a small reduction since it uses pinned memory.

Figs. 11a–11d 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-T880 architecture since the OpenCL driver used in this architecture needs some performance improvement. During the experiments, we noticed that the invocation of the function that creates a GPU buffer (clCreateBuffer) is one of the biggest causes for this overhead.

We also noticed that the memory limitation of mobile devices can result in problems when the program requires a lot of memory. Before applying DCO, AClang creates two buffers, one for the CPU use and another for the GPU use. Without using DCO in AClang, the program 3dconv requires 640MB to be executed. After applying DCO, it requires only 384MB. The execution of this program without DCO on the ARM/Mali-T880, most of the times breaks due to the amount of memory space required by the program. This problem has been solved by applying DCO.

As the experiments revealed, after applying DCO to code running on integrated GPUs most of the program offloading overhead was removed. 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.

In some programs, when the kernel computation dominates most of the total execution time, newer GPUs (with bigger computational power) do a better job in reducing the time taken by the kernel to compute. As a consequence, the share of the total program execution time taken by data offloading increases (e.g. bfs, syrk and syr2k from ARM/Mali-T880 (Fig. 11c) to ARM/Mali-G71 (Fig. 11a)). In such cases, DCO showed to be an even more powerful tool in reducing the offloading overhead. As an example, consider program bfs; it clearly shows that the reduction in the kernel computation time from ARM/Mali-T880 to ARM/Mali-G71, increases the share of data offloading in the total execution time of the program. This explains the speed-up increase from 1.73 on ARM/Mali-T880 (Fig. 13b) to 3.41 on ARM/Mali-G71 (Fig. 13a) when applying DCO, as it effectively removes a proportionally larger offloading overhead in the case of ARM/Mali-G71. In general, OpenCL kernel execution will increasingly benefit from DCO as GPUs become more powerful.

Figs. 11g–11h respectively show the execution time before and after applying DCO to programs running on an NVIDIA/Pascal Titan X, a discrete GPU. It is possible to notice that even after the application of DCO, data offloading does not disappear as in the case of most programs optimized on an integrated GPU. This happens because on a discrete GPU data transfers between CPU and GPU always occurs, even when using a pinned memory. The increase in the percentage of time spent in offloading with
Fig. 11. The breakdown of total execution time: (a), (c), (e) & (g) before DCO optimization (b), (d), (f) & (h) after DCO optimization.
respect to the overall execution time can be explained by the large reduction in the kernel computation time when running on a powerful GPU like the Titan X.

DCO also works well when applied to more complex benchmarks, such as Rodinia and Parboil. Notice that for those benchmark programs, most of the execution time is taken to run the OpenCL driver. This happens because those programs execute their kernels thousands of times from inside loops, thus producing a high overhead for the OpenCL driver to manage the kernel execution. Programs bfs and hotspot deserve more attention because their kernels are called thousands of times, and thus at each iteration data offloading is performed. Most of this overhead can be removed when applying the DCO optimization.

6.2. Kernel data size analysis

In order to evaluate its corresponding performance gains, we measured the speedup of each program with and without DCO optimization. Tables 1–2 show absolute runtime numbers for the Polybench, Parboil and Rodinia programs, in three different setups: (a) Serial execution; (b) Parallel execution using AClang; and (c) Parallel execution using AClang with DCO. As expected, substantial speed-ups have been produced in some of the programs that run the longest times. The slowdowns that we observed in benchmarks such as 2dconv, atax, bics and mvt on the integrated ARM/Mali GPUs, happened in instances that execute for a very short time. In such cases, AClang with DCO optimization is not enough to pay off for the time to create and manage offloading buffers, even in a shared memory space between CPU and GPU. To confirm this analysis, a new experiment was performed with these applications to measure the percentage of the total kernel execution time due to the OpenCL overhead when varying the kernel data sizes. As expected, Figs. 12a–12b show that longer execution times amortize the OpenCL overhead. The immediate effect is a decrease of the slowdown or even an increase in the speed-up relative to the sequential execution, as shown by the points and lines on the graphs of Figs. 12a–12b for the Intel/Iris architecture. For instance, 2dconv benchmark shows a slowdown of 0.35x for data size equals 2048 and a speed-up of

---

**Table 1**

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>ARM/Mali-G71</th>
<th>ARM/Mali-T880</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Size</td>
<td>Serial (-O3)</td>
</tr>
<tr>
<td></td>
<td>Time (s)</td>
<td>Time</td>
</tr>
</tbody>
</table>

---

**Table 2**

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Intel/Iris</th>
<th>NVIDIA/Pascal Titan X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Size</td>
<td>Serial (-O3)</td>
</tr>
<tr>
<td></td>
<td>Time (s)</td>
<td>Time</td>
</tr>
</tbody>
</table>

---
I.90x when the data size is 8192. However, this is not true for the ARM/Mali architecture. The slowdown increase for greater sizes, mainly in the 2dconv benchmark can be explained by the memory limitation of mobile devices.

Still according to Tables 1–2, it is interesting to note that there are several programs that produce slowdowns when compiled without DCO, and that produce speed-ups when compiled with DCO, like bfs on ARM/Mali-G71, 3dconv on ARM/Mali-T880 and 2dconv, atax, bicg, gramschmidt, bfs on NVIDIA/Pascal Titan X. In some cases, it is notable the high speed-ups of DCO when compared to the CPU. For instance, program covariance with DCO on ARM/Mali-G71 was 93.36x faster than when executing on CPU — without DCO it was 78.24x faster. When using ARM/Mali-T880, program gemm with DCO had a speed-up of 83.41x — without DCO it was 77.75x. Intel/IRIS also had high speed-ups; for example in gemm the speed-up was 90.71x when executed with DCO, and 89.66x without DCO. The discrete GPU – NVIDIA/Pascal Titan X, executed program gemm with a speed-up of 77.02x – without DCO it executed it with a 76.81x speed-up.

Figs. 13a–13d show the speed-up of DCO optimization with respect to the original AClang (all of them compiled with -O3) for Polybench, Parboil, and Rodinia programs, on ARM/Mali-G71, ARM/Mali-T880, Intel/IRIS and NVIDIA/Pascal Titan X 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 DCO optimization on the final code results in a speed-up of 1.71x on Intel/IRIS, 2.61x on ARM/Mali-G71, 5.25x on ARM/Mali-T880 and 8.88x on NVIDIA/Pascal Titan X. 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, what occurs during gramschmidt’s execution, generate an 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 execution.

A new experiment using Polybench applications on ARM/Mali-T880 was performed to evaluate how DCO impacts programs resulting from automatic annotation. DawnCC [16,17] is a tool that automatically inserts OpenMP's 4.X annotations to parallelize code. Fig. 14 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.

7. Related work

Previous work has 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. [12] 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. [3] 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. [26] 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. [16,17] 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 an integrated GPU, since DCO uses a shared buffer approach, where memory movements are unnecessary.

Jablin et al. [10] 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. Thouiti et al. [31] proposed a methodology that takes function calls from C language and converts 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. [14] developed a source-to-source solution from OpenMP directives to CUDA. Their solution extracts marked regions 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 [32]. Different from their solution, our work shares data buffers between CPU and GPU, without the need of memory transfers.

Said et al. [25] 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 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.

8. 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. As for future work, we plan to evaluate DCO in an environment with NVIDIA + NVLINK technology. Besides that, mainly in the case of integrated GPUs, we plan to evaluate how DCO impacts energy efficiency on integrated CPU–GPU architectures. We also plan to apply our work to OpenCL host-side codes others than the ones generated by AClang.
Acknowledgments

This work is supported by Samsung, Brazil (grant 4716.08) and FAPESP Center for Computational Engineering and Sciences, Brazil (grant 13/08293-7).

References


Marcio Pereira received the Ph.D. in computer science from the State University of Campinas UNICAMP, Brazil, in 2016. He is currently a Postdoctoral Researcher at the Institute of Computing, UNICAMP. He has proven experience in conducting technological innovation projects and new product development and IT services. Currently, his interests are in the fields of programming languages, dynamic compilation, virtual machines, code optimization, heterogeneous computing, and neural networks.

Rafael Sousa received the M.Sc. in computer science from the State University of Campinas UNICAMP, Brazil, in 2016. He is currently a Ph.D. student at UNICAMP. His research interests are in code optimization, heterogeneous computing, parallel computing/programming, and high performance computing.
Fernando Pereira received the Ph.D. in computer science from University of California, Los Angeles in 2008. He is currently an associate professor at the Department of Computer Science of the Federal University of Minas Gerais, where he coordinates the Compilers Lab. His research mission is to develop principles and techniques that help programmers to use efficiently and safely all the resources of the modern hardware. Together with his students, he published papers in important venues in the field of programming languages, and contributed to several open-source compilers, such as LLVM, GHC and Mozilla IonMonkey.

Guido Araujo received the Ph.D. in electrical engineering from Princeton University in 1997. He is currently a full professor of computer science and engineering at UNICAMP. His current research interests are in the areas of code optimization, transactional memory, cloud computing, parallel programming, and heterogeneous computing, which are explored in close cooperation with the industry.