Universidade Estadual de Campinas
Instituto de Computação

Rafael Cardoso Fernandes Sousa

Data Coherence Analysis and Optimization

Análise de Coerência de Dados e Otimização

CAMPINAS
2017

Rafael Cardoso Fernandes Sousa

Data Coherence Analysis and Optimization

Análise de Coerência de Dados e Otimização

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Guido Costa Souza de Araujo
Co-supervisor/Coorientador: Dr. Marcio Machado Pereira

Este exemplar corresponde à versão final da Dissertação defendida por Rafael Cardoso Fernandes Sousa e orientada pelo Prof. Dr. Guido Costa Souza de Araujo.

CAMPINAS
2017

Agência(s) de fomento e nº(s) de processo(s): FUNCAMP, 4719.8

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

So85d

Sousa, Rafael Cardoso Fernandes, 1988-
Data coherence analysis and optimization / Rafael Cardoso Fernandes Sousa. – Campinas, SP : [s.n.], 2017.

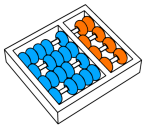
Orientador: Guido Costa Souza de Araújo.
Coorientador: Marcio Machado Pereira.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computador. 2. Compiladores (Computadores). 3. Computação heterogênea. 4. Kernel, Mapeamento de. I. Araújo, Guido Costa Souza de,1962-. II. Pereira, Marcio Machado,1959-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Análise de coerência de dados e otimização
Palavras-chave em inglês:
Computer architecture
Compiling (Electronic computers)
Heterogeneous computing
Kernel mapping
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Guido Costa Souza de Araújo [Orientador]
Fernando Magno Quintão Pereira
Sandro Rigo
Data de defesa: 24-03-2017
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação



Rafael Cardoso Fernandes Sousa

Data Coherence Analysis and Optimization

Análise de Coerência de Dados e Otimização

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araujo (Supervisor/*Orientador*)
  IC/UNICAMP
- Prof. Dr. Fernando Magno Quintão Pereira
  Universidade Federal de Minas Gerais (UFMG)
- Prof. Dr. Sandro Rigo
  Institute of Computing - UNICAMP

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 24 de março de 2017

Agradecimentos

Primeiramente, gostaria de agradecer a minha família, por estarem sempre do meu lado, me auxiliando e me guiando nos meus momentos mais difíceis. Em especial, agradeço-lhes por todo o apoio e auxilio dado, visto que sempre foram de grande importância nas minhas decisões e conquistas.

Gostaria também de agradecer a minha namorada, que além de ter me apoiado por durante todo esse tempo, esteve também sempre me dando conselhos e sugestões nas minhas tomadas de decisões. Sobretudo, esteve também durante todo esse período presente, auxiliando no planejamento do nosso futuro.

Em especial, gostaria de agradecer tanto o meu orientador quanto o meu coorientador pelo aprendizado que tive advindo de ambos. Vale ressaltar que a paciência e experiência de ambos foram sempre de grande importância no desenvolvimento da minha dissertação.

Agradeco fortemente o apoio de todos os amigos que fiz no IC e, em especial, àqueles do Laboratório de Sistemas Computacionais (LSC), por sempre se disponibilizarem, auxiliando na resolução das minhas dúvidas.

Por fim, gostaria de agradecer a Samsung pelo apoio financeiro dado durante todo o desenvolvimento do projeto. E, em especial, toda a secretária do IC/UNICAMP, por terem me auxiliado, pacientemente, em todos os processos burocráticos da UNICAMP.

Resumo

Embora a computação heterogenea tenha permitido ganhos de desempenho (speed-ups) impressionantes, o conhecimento sobre a arquitetura dos dispositivos aceleradores para colher todos os benefícios de seu hardware ainda é algo crítico. A programação em cima dessas arquiteturas é complexa, propensa a erros e geralmente é feita por meio de linguagens especializadas (por exemplo, CUDA) ou bibliotecas (por exemplo, OpenCL). Em particular, para os programadores não especialistas, o custo de mover e manter dados coerentes entre host e o dispositivo acelerador (device) pode facilmente eliminar quaisquer ganhos de desempenho alcançados pela aceleração. Esta tese propõe Análise de Coerência de Dados (DCA), uma simples e útil técnica de análise de fluxo de dados que determina como as variáveis são usadas pelo host/device em cada ponto do programa. Ela também introduz a Otimização de Coerência de Dados (DCO), um algoritmo baseado em DCA que: (a) usa informações das variáveis para alocar buffers OpenCL compartilhados entre o host e o device; e (b) inserir chamadas de função OpenCL apropriadas em pontos do programa de modo a minimizar o número de operações de coerência de dados. O DCO foi implementado no compilador GPUClang LLVM que é capaz de traduzir automaticamente os loops anotados do OpenMP 4.X para kernels OpenCL, escondendo assim toda a complexidade da programação direta no OpenCL. Os resultados experimentais revelam que, enquanto GPUClang mostra desempenho de até 78x, GPUClang com DCO consegue speed-ups de até 84x em programas do benchmark Polybench rodando em um Exynos 8890 Octacore CPU com ARM Mali-T880 MP12 GPU e até 92x em um Processador dual core Intel Core i5 de 2,4 GHz equipado com uma unidade Intel Iris GPU.

Abstract

Although heterogeneous computing has enabled some impressive program speed-ups, knowledge about the architecture of the target device is still critical to reap the full benefits of its hardware. Programming such architectures is complex, error-prone and is usually done by means of specialized languages (e.g. CUDA) or complex function libraries (e.g. OpenCL). In particular, for non-expert programmers the cost of moving and keeping host/device data coherent can easily eliminate any performance gains achieved by acceleration. This dissertation proposes Data Coherence Analysis (DCA) a simple and yet useful data-flow analysis technique that determines how variables are used by host/device at each program point. It also introduces Data Coherence Optimization (DCO), a DCA-based algorithm that: (a) uses variable information to allocate OpenCL shared buffers between host and devices; and (b) inserts appropriate OpenCL function calls into program points so as to minimize the number of required data coherence operations. DCO was implemented in the *GPUClang* LLVM compiler which is capable of automatically translating OpenMP 4.X annotated loops to OpenCL kernels, thus hiding all the complexity of directly programming in OpenCL. Experimental results reveal that while *GPUClang* shows performance of up to *78*x, *GPUCLang* with DCO can achieve speed-ups of up to *84*x on programs from the Polybench benchmark running on an Exynos 8890 Octacore CPU with ARM Mali-T880 MP12 GPU and up to *92*x on a 2.4 GHz dual-core Intel Core i5 processor equipped with an Intel Iris GPU unit.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

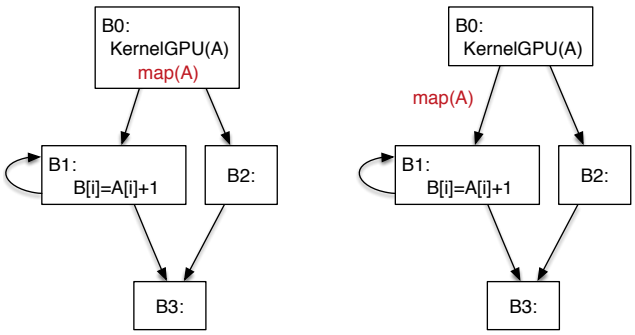
With the advent of heterogeneous computing many parallel programming models have emerged seeking to leverage the performance of sequential code by *offloading* computation *kernels* from a *host* machine (e.g. CPU) to an acceleration device (e.g. GPU). Computation offloading is typically achieved by 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 [4] and OpenMP [5] or specialized language and libraries as in CUDA [1] and OpenCL [2] 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.

One option to avoid the offload mechanism is to have unified virtual and physical address spaces between CPUs and accelerators. Unified address spaces enjoy many benefits; they make data structures and pointers globally visible between CPU and GPU, obviating the need for expensive memory copies between CPUs and accelerators. Unified address spaces do, however, also require architectural support for virtual-to-physical address translation [13].

A discrete GPU (dGPU) resides on the PCIe interface and has traditionally required data to be moved from the host memory to the GPU memory via PCIe. In certain applications, the overhead of these data transfers between memory spaces can nullify any performance gains achieved from faster computation on the GPU [16, 36, 37]. To overcome the data-transfer overheads, recent GPU advancements enable GPUs to directly access data from the host memory as well as enable CPUs to directly access data from the GPU memory. Such kind of GPU is called "integrated GPU" (iGPU for short). Therefore, shared variable are required to be marked as zero-copy during allocation and data coherence must be guaranteed by the system software so as to avoid computation inconsistencies on both sides (CPU and device) [33] [34].

Although there has been a number of efforts to address data coherence across host-device boundaries [17, 18, 19, 20] no universal hardware coherence protocol standard has yet been defined for heterogeneous systems, particularly due to the fact that different devices demand very distinct data block sizes. Therefore, coherence has to be performed in software by means of specific `map`/`unmap` function calls that copy variables modified by the device/host back to the host/device, thus squashing any old copies of the data that

Figure 1.1: Inserting `map` instruction to keep array `a` coherent between CPU and GPU.



they might be holding. Such operations are typically done by the programmer by means of calls to functions from specialized libraries (e.g. OpenCL) or by a compiler that naively inserts such calls at the entry/exit of the kernels. Given that most accelerators use very large data blocks a non-optimal insertion of `map`/`unmap` calls can result in unnecessary coherence operations thus impacting the overall program performance.

In order to reduce the offloading/coherence overhead it is important that optimizing compilers targeting heterogeneous systems identify variables that can be allocated in the shared memory between CPU and GPU and perform code transformations to: (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 devices coherent. Finding the best locations to insert coherence `map`/`unmap` calls into source code can be casted as the *Data Coherence Analysis and Optimization* problem. Hence, solving the DCAO problem involves: (1) identifying the blocks of code where shared variables are used by different devices (e.g. CPU or GPU) and (2) insert `map`/`unmap` calls so as to minimize the need of data coherence operations among host and devices. Since both coherence and data offloading impact program performance, these problems are inter-dependent and should be addressed together.

In order to exemplify a solution to DCAO please consider the *Control-Flow Graph* (CFG) of Figure 1.1(a)-(b). For the sake of simplicity this work will consider a CPU host and a GPU acceleration device, although all the ideas discussed herein can be applied to any other devices. In Figure 1.1(a) basic block `B0` dispatches and executes kernel `KernelGPU` which modifies a large shared array `A`. In order to keep `A` coherent with the CPU host, a non-expert programmer could insert a `map(A)` instruction at the end of `B0` as shown in Figure 1.1(a). Also notice that a naive compiler could also insert a `map` call at the end `B0` to automatically assure data coherence. This will make the GPU update its copy of `A` after `KernelGPU` finishes and the flow of execution takes the program 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` call on the edge that connects `B0` to `B1`, instead of inserting it at the end of `B0`.

In order to address the above described problem this work makes the following contributions:

- It proposes the *Data Coherence Analysis* — DCA, a program data-flow analysis

algorithm to determine the usage of data by heterogeneous devices at each program point; this allows to detect which variables are shared between host and devices.

- It introduces the *Data Coherence Optimization* — DCO, a DCA-based algorithm that performs two tasks: (a) replaces standard host memory allocation mechanisms (e.g. `malloc` and `calloc`) for specialized OpenCL shared buffer allocation; and (b) calls OpenCL functions `map` and `unmap` into program points so as to minimize the amount of data coherence operations required between host and device.

The rest of the work is organized as follows. Chapter 2 details the costs of data offloading and coherence operations in a typical heterogeneous platform. Chapter 3 introduces the data items required by DCA and describes its mathematical formulation. Section 3.2.2 discusses the implementation details of the corresponding LLVM optimization pass that implements a solution to DCO. Chapter 4 discusses related work and Chapter 5 shows the experimental evaluation. DCAO is implemented within *GPUClang*, a LLVM based compiler [10] capable of automatically translating OpenMP 4.X annotated loops to OpenCL kernels. GPUClang flow is discussed in Section 5.1. Chapter 6 concludes the work and points to future directions.

Chapter 2

Background

Heterogeneous computing has shown that specialized acceleration devices (e.g. GPUs) can provide significant performance improvement for a range of applications [38]. 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 memory shared 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.

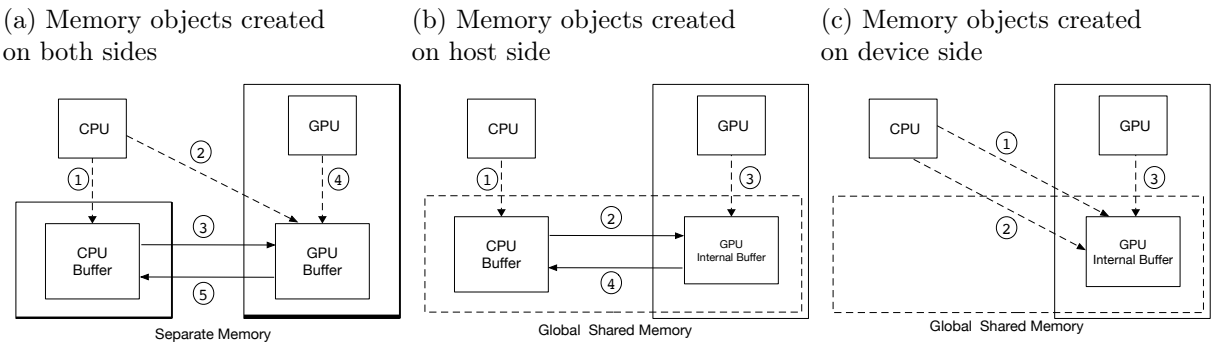
Unlike CUDA, which runs only on NVIDIA devices, OpenCL allows programmers to execute their code on a broad range of heterogeneous platforms. It enables them to use any acceleration hardware for which there exist an OpenCL runtime. Nevertheless, although OpenCL uses high-level function calls, a typical programmer has to undergo a steep learning curve to be able to write efficient OpenCL code.

OpenMP has been extensively used as a parallel programming standard for multi-core architectures [39]. In order to extend OpenMP ability to program heterogeneous architectures the OpenMP Accelerator Model [14] has been recently released; it adds to the standard a new set of clauses target to programming heterogeneous devices. One way to enable programmers to leverage on OpenMP easy of programmability and tap on OpenCL nice heterogeneous programming features is to generate OpenCL kernels from C/C++ OpenMP code. This work uses an LLVM/Clang compiler called *GPUClang* that does exactly that (Section 5.1).

No matter if the OpenCL kernel is directly programmed by a programmer or if it is synthesized from OpenMP code, the resulting kernel can still suffer from some recurrent problems. One of the most complex issues involved in using OpenCL is the need to minimize the overhead of data offloading while keeping up with the coherence between host and device.

In order to understand how coherence is maintained between CPUs and devices it is relevant that programmers have a good knowledge of the architecture of modern heterogeneous platforms. A typical heterogeneous node includes one multi-core CPU chip connected to one accelerator through a PCIe bus, where each chip is connected to separate physical memories. Data-transfers in current commercial heterogeneous nodes are based on Direct Memory Access (DMA) hardware, which is typically exposed to applications

Figure 2.1: 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.



programmers through memory copy routines [22] [23]. Most programming models for heterogeneous systems, such as NVIDIA CUDA [1] and OpenCL [2], assume that the code running in the CPU (i.e., host) and the accelerator (i.e., device) has access to separate virtual address spaces. Therefore, any data communication between the CPU and accelerators, or between accelerators, requires explicit data transfers calls (e.g., *cudaMemcpy* in CUDA, and *clEnqueueCopyBuffer* in OpenCL) to copy data across address spaces. The optimal implementation of these data transfer calls heavily depends on the underlaying hardware organization. Hence, the behaviour of applications have to be adapted to the hardware topology where they are being executed.

In order to reduce the offloading/coherence costs modern CPU/GPU architectures use a shared physical memory by means of a Shared Virtual Memory (SVM) 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 devices (e.g. ARM7/Mali) which need to minimize as much as possible the amount of energy consumed. In such cases, useless data-movements between CPU and GPU represent an unacceptable overhead.

Today CPUs and GPUs typically use separate virtual and physical address spaces. Main memory may be physically shared, but is usually partitioned, or allows unidirectional coherence (e.g., ARM allows accelerators to snoop CPU memory partitions but not the other way around) [13]. More recently, processor vendors like Intel, AMD, ARM, Qualcomm, and Samsung are embracing integrated CPU/GPUs and moving towards fully unified address space support, as detailed in Heterogeneous Systems Architecture (HSA) specifications [9] [12].

2.1 OpenCL Data Offloading/Coherence

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

2.1.1 Host/Device Buffers

Memory objects form the most fundamental architectural unit in OpenCL programming. These memory objects (or buffers) refer to any type of contiguous data location that can be used by the kernels during execution. 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 `alloca`. But, that's where the similarity ends. For instance, host variables can span multiple non-contiguous pages in host virtual memory whereas the target device operates on contiguous physical memory. To deal with this memory issue, OpenCL provides the 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:

CL_MEM_READ_WRITE : The buffer is created in the device global memory and can be read and written by the kernel.

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 buffer back to the host.

CL_MEM_ALLOC_HOST_PTR : The buffer is allocated at the device memory, can be mapped to the host memory and accessed by it.

Now, the question which could arise in the reader's mind is how different are the usages of these buffer objects and how they affect the costs of offloading and coherence? Figures 2.1(a)-(c) show the typical ways in which host and devices use OpenCL buffers to communicate. As described below, there are basically two ways of storing OpenCL buffers, in separate or shared memories. In Figure 2.1, dashed lines represent host/device actions on the buffers and full lines are data movement operations.

- **Separate Host/Device Memories**
  Figure 2.1(a) 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` with the `CL_MEM_READ_WRITE` memory flag to create 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 2.1(b) and 2.1(c) show the buffering approaches that can be more efficient when the host and device share the same global

memory ("physical memory"). In this case, the device memory is mapped on the global shared memory space. In Figure 2.1(b), prior to calling `clCreateBuffer` with flag `CL_MEM_USE_HOST_PTR`, 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. After calling the function `clCreateBuffer` the host invalidates its own pointer to the buffer, turns over the CPU buffer control to the OpenCL driver. Depending on the OpenCL implementation, the driver transparently 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 `map` call (see Section 2.1.2). In this case data is automatically transferred to the host and remains there until an `unmap` call occurs ④. In Figure 2.1(c), memory objects are created at the device memory by the `clCreateBuffer` using memory flag `CL_MEM_ALLOC_HOST_PTR` ①. If the host needs to access the data on the buffer, it calls the `map` function ②. As detailed on Section 2.1.2, the `map` function transfers data ownership to the host and the device cannot access it until the host calls `unmap` , and releases the device to access it ③.

Although there are many ways to handle buffer mapping in OpenCL, their efficiency depends on the OpenCL driver implementation and programmer expertise. For example, `clCreateBuffer` with the `CL_MEM_USE_HOST_PTR` flag uses the host memory as a buffer location, but when it comes to accessing the data at the device side, the OpenCL implementation may pin this memory and then transfer the data over to the device. But in the case of `CL_MEM_ALLOC_HOST_PTR` flag, the OpenCL implementation may allocate memory directly on the pinned memory location which the OS uses for data transfer using DMA. This may be faster when compared to previous `clCreateBuffer` call with the `CL_MEM_USE_HOST_PTR` memory flag. Nevertheless, it is a consensus that the programming model of Figure 2.1(c) gives significant improvement in performance when compared to a regular `clCreateBuffer` call with the `CL_MEM_READ_WRITE` memory flag, and thus the *GPUClang* implementation of DCO is based on it.

2.1.2 Host/Device Coherence Calls

In order to keep the coherence of the data between host and device, OpenCL provides four function calls that enables the host to switch access to the shared buffer. These functions are:

- `map`: this call is implemented using the OpenCL function `clEnqueueMapBuffer`. It hands the shared buffer pointer from the device to the host. Before releasing the pointer 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. For example, this operation is always required after the execution of a GPU kernel that modifies a given shared buffer v that is used in sequence by the CPU. The `map` function can be mapped for read or write, depending on the behavior taken by the program. The `map` function must be

used in two different scenarios. First, it is necessary to call the `map` function for any operation that occurs in shared buffer by the CPU, whether it is a read or write operation on buffer. Following a control flow execution of the application, if there is more than one sequential use of the buffer, without any GPU execution between them, this means that only one `map` function is required during this execution flow. The second way to apply the `map` function occurs after a GPU execution that writes in the buffer defined as shared between CPU and GPU.

- `unmap`: this call hands the shared buffer pointer from the host to the device. It is implemented by means of the OpenCL function `clEnqueueUnMapMemObject`. Before releasing the pointer to the device it also flushes into the shared buffer all the data modified by the host. For example, this call is always required before a GPU execution that uses a given shared buffer v that was modified by the CPU before its execution. The `unmap` function must be called before a kernel execution on the GPU; Applying `unmap` functions is always necessary when the kernel execution writes in the shared buffer. This is also necessary when the buffer is written by the host before calling the kernel execution. Note that this is not necessary when host and GPU computing are read-only, since the buffer contents are not modified. If the `unmap` function is not called, the execution becomes undefined. If the CPU attempts to access the contents of a pointer that has been unmapped previously, the execution also becomes undefined.

- `read`: this call is executed by means of the OpenCL function `clEnqueueReadBuffer`. This function copies the specified data from the device buffer to the host buffer.

- `write`: this call is implemented using the OpenCL function `clEnqueueWriteBuffer`. This function copies the specified data from the host buffer to the device buffer. This function does not add any additional functionality in terms of maintaining coherence between host and device, as a similar behavior can be attained by means of the `map`/`unmap` function calls.

When working with buffers shared between CPU and GPU, a data coherence protocol must be applied to avoid an undefined execution. Therefore, after making one buffer shared, it becomes necessary to insert `map` and `unmap` functions to maintain the data coherent between CPU and GPU. When applying it in an iGPU, the `map` function causes the shared buffer to becomes visible to the CPU so that when called, it returns a pointer to the buffer that was previously defined as shared between the CPU and GPU. The action taken by the `map` function is to invalidate all pages that the buffer has in the CPU cache, since there is no automatic coherence of the CPU for the GPU. However, there is coherence in the opposite direction. The `unmap` function basically serves to deallocate a pointer used by the CPU so a write-back operation is done in the CPU cache, making the buffer available exclusively for GPU use.

The read and write functions are commonly used in applications that require data offloading. Applications running in a dGPU (e.g. NVIDIA Tesla), usually call both functions to transfer data via PCI-e into and out of the device, since it has its own internal memory used during kernel computation. It is important to note that the use of

read and write calls in an iGPU may generate unnecessary overheads, since two buffers are created, even though it can be run in only one. The CL_MEM_USE_HOST_PTR flag can also be used by dGPU, where `map` and `unmap` functions instead of maintaining the cache coherence – as the case of iGPU, it performs data offloading between CPU and GPU memories.

Chapter 3

Data Coherence Analysis and Optimization

Nowadays with GPU and CPU sharing the same physical memory, programmers must perform a series of manual code modifications based on the characteristics of the architecture and application algorithms. This task is typically error-prone, since the programmer needs to have a very good knowledge of the underlying hardware and of the application data layout and execution flow.

This research work makes two main contributions. The first one is an analysis algorithm, called Data Coherence Analysis (DCA), whose main focus is to gather information from the program execution flow. The second is Data Coherence Optimization (DCO), an algorithm that inserts `map` and `unmap` calls into the program in order to maintain data coherence between CPU and GPU, based on the informations gathered DCA.

The main goals of this work are threefold: (a) to identify variables that are used by both host and device and make then shared; (b) to allocate buffers for shared variables; and (c) to discover which program points need `map`/`unmap` OpenCL calls for a given shared variable that is used at different moments by CPU and GPU.

DCAO is performed in two passes. First, DCA analysis is used to determine at each program point which devices (e.g. CPU/GPU) accesses live program variables, as well as the type of the accesses – read (`R`), write (`W`), or read and write (`RW`). In the second pass, DCO optimization performs two code transformation steps: (a) *shared buffer allocation* which detects those CPU allocated variables (through *malloc* or *calloc*) that are also accessed from within GPU kernels; allocation of such variables at the CPU side are then replaced by allocation of OpenCL buffers that are shared by both CPU and GPU; (b) *coherence call insertion* which inserts `map` and `unmap` function calls to keep CPU-GPU shared buffers coherent during program execution.

3.1 Data Coherence Analysis (DCA)

In the *GPUClang* compiler, the host adopts a standard offloading mechanism; OpenCL functions are called to copy kernel's input data to the GPU memory and return the kernel's output to the CPU.

Data Coherence Analysis (DCA) is an intra-procedural analysis that gathers information to identify program execution points where transitions occur between CPU and GPU usage. The analysis must be performed for variables used by the CPU that are also accessed by the GPU kernel. However, for reasons of efficiency in this work, these variables are restricted to arrays and pointer variables.

For each variable `v`, it is necessary to identify if there exist any use of variable `v` before the dispatch of the kernel execution to GPU, and if there is an access to `v` after the kernel execution. Overall DCA seeks to answer the following questions for each variable v at each program point p:

- What is the first access to variable v reachable from p?
- Which type of access is performed: read, write, or read and write?
- Which devices access v: CPU or GPU?

For example, in Figure 1.1 if the compiler would know at the end of block `B0` that a CPU access to array A is performed at its successor block `B1` it could automatically insert a `map` instruction to make array A available to the CPU after it is modified by the GPU through KernelGPU(A) .

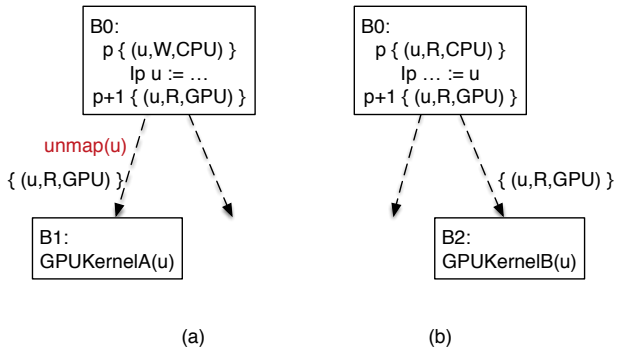
To assemble this type of information at any program point p the compiler has to know: (a) all *variables* v that are live at any execution path P that starts at point p and reaches the end of the program; (b) the type (read/write) of the first access to v on P; and (c) the device that accesses v. In order to capture this information one can define the *access tuple* A (Equation 3.1) as the *coherence item* that will work as the basic element during DCA. The first component of A, named v identifies the variable being analyzed. The second component t describes the *type* of the access to v, i.e. whether it is a read (R), a write(W), or a read and write (RW) operation. The third component d identifies if the `device` that performs the access is `CPU`, `GPU` or `X`, i.e. unknown at that program point. Hence, at each program point p there exist a *coherence set* S_p of tuples which contains the coherence items for all variables that are reachable on all paths starting at p. For example if A_p = `(x,R,GPU)` is an item of S_p (i.e. $A_p \in S_p$) one can say that there is a read of variable x from GPU at some path P starting at program point p.

$$(variable, type\texttt{[R,W,RW]}, device\texttt{[CPU,GPU,X]}) \tag{3.1}$$

To illustrate how the access in Equation 3.1 can be used to identify coherence optimization opportunities, please consider from Figure 3.1(a) the coherence sets S_p = {`(u,W,CPU)`} and S_{p+1} = {`(u,R,GPU)`} where $p+1$ is the point just after instruction I_p in basic block `B0`. Given these two sets, one can say that: (a) instruction I_p is a CPU instruction that writes to variable u; and (b) at point $p+1$ just after I_p starts an execution path that reaches kernel `GPUKernelA(u)` which reads u (at basic block `B1`).

Observe that the transition from p to $p+1$ is a CPU to GPU transition, which means that it is necessary to `unmap` variable u after $p+1$ and before the host CPU dispatches kernel `GPUKernelA(u)` to the GPU as `GPUKernelA(u)` reads to variable u in `B1`. Now

Figure 3.1: Using DCA tuples to insert `map` and `unmap` instruction to keep `u` coherent between CPU and GPU.



consider in Figure 3.1(b) that variable u is read by the CPU at instruction I_p and that the flow of execution leaves `B0` and goes to `B2` where it finds a kernel `GPUKernelB(u)` that only reads u. Hence, set $S_{p+1} = \{$`(u,R,GPU)`$\}$ and it is not necessary to `unmap` variable u before dispatching kernel `GPUKernelB(u)` given that the GPU does not modify it.

In order to define the DCA data-flow equations one has to answer three basic questions:

- What is the flow of the analysis in DCA? As discussed above, the accesses to program variables should move upward on the CFG, i,e. contrary to the program execution flow. Hence, DCA is a *backward* data-flow analysis;
- What is the "combiner" operator ($\uplus$) in DCA? An access that points to a sentence $p+1$ is combined with its predecessor p using the combiner operator to generate the set S_p. Its use is as follows: from a sentence s, it takes an access tuple A_i from $GEN[s]$, and another access tuple A_j of $OUT[s]$, considering that both have same variable `v`, to generate an access tuple A_k to compose set $IN[s]$.
- What is the meet operator ($\uplus$) in DCA? All accesses that arrive at a point $p+1$ just after an instruction I_p should be combined into a single coherence set S_{p+1}. Thus the DCA meet operator is the union operator, slightly changed to handle the cases when accesses to the same variable with different types and devices meet at $p+1$ (detailed in Section 3.1.2).

Given the answers above one can think of DCA as an extended form of *Liveness Analysis* [15], where the information item is an access containing not only the name of the variable but also the type of the access and the device that performed it.

3.1.1 Local Data Coherence Analysis

In order to formulate a data-flow analysis problem one has to define the information items generated ($GEN[s]$) and killed ($KILL[s]$) at each program statement s. This requires analyzing each possible statement type in the compiler *Intermediate Representation* (IR) to determining such sets. For the sake of simplicity this section shows at Equations 3.2 – 3.4 definitions for $GEN[s]$ and $KILL[s]$ for the generic three-address code statement `s: v = a op b` which computes the operation `op` on two operands `a` and `b` and stores the

result into v. In those equations the set of accesses performed on variable v is represented by A_v.

$$GEN[s] = \begin{cases} \{(v, \mathtt{W}, d), (a, \mathtt{R}, d), (b, \mathtt{R}, d)\}, \\ \qquad \texttt{if } v \neq a \ \wedge v \neq b \quad (a) \\ \{(a, \mathtt{RW}, d), (b, \mathtt{R}, d)\}, \texttt{if } v = a \quad (b) \\ \{(a, \mathtt{R}, d), (b, \mathtt{RW}, d)\}, \texttt{if } v = b \quad (c) \end{cases} \tag{3.2}$$

$$KILL[s] = A_v - GEN[s] \tag{3.3}$$

$$A_v = \{\texttt{set of accesses to v}\} \tag{3.4}$$

For the computation of $GEN[s]$ three cases need to be treated in Equation 3.2, as follows: 3.2(a) in this case a and b are read and a different variable v is written; thus $GEN[s]$ will contain accesses to a and b and the access (v,W,d) which defines that variable v is written during the execution of s. 3.2(b) this case occurs when variable a is read, operated with variable b and the result is stored into v = a. In such case, $GEN[s]$ will have an access (b,R,d) which informs that variable b is read by device d, and access (a,RW,d) meaning that variable a is read and written by d; 3.2(c) this is a similar case as in Equation 3.2(b) but this time it refers to variable b; similarly as in liveness analysis, in Equation 3.3 statement s kills from set A_v all accesses that read or write to v, but those accesses that are in $GEN[s]$.

The $OUT[s]$ is defined as a function applied on all successors of $IN[S]$, according to the Equation 3.5, where S is the successor of sentences s. It uses the meet operator to combine tuples. The meet operator ($\uplus$) is explained in more details in the Section 3.1.2, where its complexity is in fact used; but, considering that Local DCA only applies it at the level of sentences, where one sentence has only one successor, then the tuples in $IN[s_{i+1}]$ are directly attributed to $OUT[s_i]$.

$$OUT[s] = \biguplus_{S \in Succ[s]} IN[S] \tag{3.5}$$

Finally, given that DCA is a backward analysis one can define the $IN[s]$ as a function of $OUT[s]$ according to the Equation 3.6. Therefore, for a given basic block B one can compute $IN[s_i]$ as a function of $OUT[s_i]$, $GEN[s_i]$, and $KILL[s_i]$, where $s_0, s_1, \ldots, s_{k-1} \in B$ are the k statements of block B.

$$IN[s] = GEN[s] \ \dot{\cup} \ (OUT[s] - KILL[s]) \tag{3.6}$$

The operation $S_1 \dot{\cup} S_2$ is defined in Equation 3.7 as the relation $S_1 \odot S_2$ for which all elements $A_i \odot A_j \in S_1 \dot{\cup} S_2$ are computed according to the Equation 3.8. Operation $A_i \odot A_j$ basically takes two accesses and determines the access which should result when they are combined.

$$S_1 \bigcup\!\!\!\!\bigcdot S_2 = \bigcup_{\forall i,j} \{A_i \odot A_j\}, \forall A_i \in S_1 \wedge A_j \in S_2 \tag{3.7}$$

$$A_i \odot A_j = \begin{cases} \{A_i, A_j\}, \texttt{ if } v_i \neq v_j & (a) \\ \{(v, t_i \odot t_j, d_i \odot d_j)\}, \texttt{otherwise} & (b) \end{cases} \tag{3.8}$$

Tables 3.1 and 3.2, define, respectively, the confluence operator for access devices and access types. This operator is applied to combine two tuples that have the same variable v of a sentence s in the $IN[s]$ computation.

In order to clarify how $\odot$ works, consider for example sentence s, that uses variable `v`, and , where $GEN[s] = \{A_i\}$ and A_i=`(v,RW,CPU)`. Assume also that $OUT[s] = \{A_j\}$, where A_j=`(v,RW,GPU)`. When applying the confluence operator $A_i \odot A_j$, a new tuple A_k=`(v,RW,CPU)` results, and thus $IN[s] = \{A_k\}$. The operator ($\odot$) first checks where the computation is being applied, and which type of access is performed both A_i and A_j before generating A_k. It is important to notice that this operation depends on the tuple's order, i.e., operator ($\odot$) is not commutative.

Table 3.1: The confluence operator $\odot$ for access devices.

Confluence Access Devices		
d_i	d_j	$d_i \odot d_j$
CPU	[CPU,GPU,X]	CPU
GPU	[CPU,GPU,X]	GPU

Table 3.2: The confluence operator $\odot$ for access types.

Confluence Access Types		
t_i	t_j	$t_i \odot t_j$
R	R	R
R	[W,RW]	if (d_i == CPU && d_j == GPU) R else RW
W	[R,RW]	RW
W	W	W
RW	[R,W,RW]	RW

By using the above described formulation Local DCA of basic block `B` (containing n sentences) is performed as follows. It starts execution from the last sentence s_n of `B` until it reaches the first sentence s_1, using at each sentence s Equation 3.6 to compute $IN[s_n]$. In sequence, the sentence s_n spreads all tuples of $IN[s_n]$ to $OUT[s_{n-1}]$ of its successor s_{n-1}, so that s_{n-1} can apply the same data flow equation.

3.1.2 Global Data Coherence Analysis

After $GEN[B]$ and $KILL[B]$ are computed for each basic block B of the program *Global Data Coherence Analysis* (*Global DCA*) can determine the set of accesses S_p that reach

program point p. As described in Section 3.2.2 this is the information required to correctly insert the `map` and `unmap` calls needed to keep CPU and GPU data coherent.

As discussed above, Global DCA is a backward analysis and as such one can define at the end of each basic block B the *meet* operator $\uplus$ that merges together sets $IN[S]$, for all basic blocks $S \in Succ[B]$, where $Succ[B]$ is the set of blocks that are successors of B. This can be represented by Equation 3.9. The operator $(\odot)$ is also applied in Global DCA to generate the tuples of $IN[B]$ of a basic block B, according to the Equation 3.10.

$$OUT[B] = \biguplus_{S \in Succ[B]} IN[S] \tag{3.9}$$

$$IN[B] = GEN[B] \biguplus (OUT[B] - KILL[B]) \tag{3.10}$$

Hence, for each basic block B of the program, Equations 3.9 – 3.10 compute the sets $IN[B]$ and $OUT[B]$ at the entry and exit of each block. This can be achieved using a standard fixed-point iteration algorithm very similar to Liveness Analysis and for which there exist well-know proofs of convergence [15].

On the other hand, unlike Liveness Analysis which uses a standard set union as a meet operator, in Global DCA the meet operator is more complex. The reason is that in Liveness Analysis the information item is just the name of the variable v while in DCA it is the access tuple $A = (v, t, p)$ which contains not only the variable name v, but also the type of access t and the device d that performed this access. Thus one has to define how two accesses should be merged in the final union. This is simple when both accesses refer to different variables: one just needs to add both accesses to the resulting meeting set. But this is not obvious when the accesses refer to the same variable.

In order to explain that, and without loss of generality, please consider the case when a basic block B has two successors S_1 and S_2. Also for the sake of simplicity assume that $A_i \in S_1$ and $A_j \in S_2$ are accesses from sets S_1 and S_2 respectively and that $A_i = \{(v_i, t_i, d_i)\}$ and $A_j = \{(v_j, t_j, d_j)\}$.

The operation $S_1 \uplus S_2$ is defined in Equation 3.11 as the relation $S_1 \oplus S_2$ for which all elements $A_i \oplus A_j \in S_1 \uplus S_2$ are computed according to Equation 3.12. Operation $A_i \oplus A_j$ basically takes two accesses and determines the access which should result when they are combined.

$$S_1 \biguplus S_2 = \bigcup_{\forall i,j} \{A_i \oplus A_j\}, \forall A_i \in S_1 \wedge A_j \in S_2 \tag{3.11}$$

$$A_i \oplus A_j = \begin{cases} \{A_i, A_j\}, \texttt{ if } v_i \neq v_j & (a) \\ \{(v, t_i \oplus t_j, d_i \oplus d_j)\}, \texttt{otherwise} & (b) \end{cases} \tag{3.12}$$

As shown in Equation 3.12(a) if A_i and A_j do not access the same variable v they will both be added as elements of $S_1 \oplus S_2$. On the other hand, if they access the same variable as in Equation 3.12(b) (i.e. $v_i = v_j = v$) the operator $\oplus$ needs to define the resulting type and device that accesses v. As shown in Table 3.3 if the types of t_i and t_j are the same the resulting type is given by $t_i \oplus t_j$ and is either `R`, `W`, or `RW`. On the other hand, if one of

Table 3.3: The meet operator ⊕ for access types.

Meeting Access Types		
t_i	t_j	$t_i \oplus t_j$
R	R	R
R	[W,RW]	RW
W	W	W
W	RW	RW
RW	RW	RW

Table 3.4: The meet operator ⊕ for access devices.

Meeting Access Devices		
d_i	d_j	$d_i \oplus d_j$
CPU	CPU	CPU
GPU	GPU	GPU
GPU	[CPU,X]	X
CPU	[GPU,X]	X
X	[CPU,GPU,X]	X

the types is `RW` then $t_i \oplus t_j$ = `RW` meaning that if some successor at the end of B performs a read and write to v this information should be carried up to the statements in block B.

Regarding the resulting device $d_i \oplus d_j$ of the accesses that meet at the end of block B, Table 3.4 reveals that if both accesses are the same the resulting device is `CPU` or `GPU`. On the other hand, if the devices are not the same one cannot at compile time determine which device will access v. Thus the resulting device will be `X` meaning that it is not possible to statically compute the accessing device at the meet point. This information will propagate upwards in the CFG and will later be used, during the optimization pass (see Section 3.2.2) to determine the need to insert `map` or `unmap` instructions on the edges that take block B to its successors.

Global DCA uses a standard backward data-flow analysis algorithm. It starts computing $GEN[B]$ and $KILL[B]$, through Local DCA inside each basic block of the CFG. The transfer function at each statement is defined in Section 3.1.1. The out-states ($OUT[B]$) result from applying the meet operator on the in-sets ($IN[S]$) of the successors S of basic block B.

3.1.3 Computing GEN and KILL

The GEN and KILL sets are generated for all LLVM IR sentence that read or write a variable v, either by the `CPU` or `GPU`. As discussed before, it is restricted to variables that are arrays or pointer variables.

The sentences s_i that DCA consider as a CPU's scope are those that perform a read or write on a given variable v by the `CPU`, according to the Table 3.5. The *call* instruction is considered of type `W` to maintain the conservativeness, given that DCA is an intra-procedural algorithm.

The sentences s_i that DCA considers as in GPU's scopes are those that execute a kernel computation according to the Table 3.6. However, as those sentences do not carry

Table 3.5: GEN[s] and KILL[s] table for CPU's sentences

s_i	GEN[s]	KILL[s]
load	(v, R, CPU)	$(u, x, y)\ \forall\ u = v\ \mid$ $x\ \in \{R, W, RW\}$ $y\ \in \{CPU, GPU, x\}$
store call	(v, W, CPU)	$(u, x, y)\ \forall\ u = v\ \mid$ $x\ \in \{R, W, RW\}$ $y\ \in \{CPU, GPU, x\}$

enough informations to classify the access type done by the kernel execution as `R` or `W`, DCA works looking for sentences s_j that the program uses to create GPU buffers, used by the kernel, and it also looks for sentences that performs data offload before and after the kernel execution.

Table 3.6: GEN[s] and KILL[s] table for GPU's sentences

s_i	s_j	GEN[s]	KILL[s]
_cl_execute_tiled_kernel _cl_execute_kernel	_cl_create_read_only _cl_offloading_read_only _cl_read_only	(v, R, GPU)	$(u, x, y)\ \forall\ u = v\ \mid$ $x\ \in \{R, W, RW\}$ $y\ \in \{CPU, GPU, x\}$
	_cl_create_write_only _cl_create_read_write _cl_read_buffer _cl_offloading_read_write _cl_offloading_write_only	(v, W, GPU)	$(u, x, y)\ \forall\ u = v\ \mid$ $x\ \in \{R, W, RW\}$ $y\ \in \{CPU, GPU, x\}$

Note that for both Tables 3.5 and 3.6 the KILL function eliminates all previous tuples for the same variable. Also, notice that for all other LLVM IR that are not in the tables have empty $GEN[s]$ and $KILL[s]$ sets.

3.1.4 Running DCA

Before starting Global DCA, a Local DCA pass is used to compute the initial value of $IN[B]$ from $GEN[B]$, as expressed in Equation 3.13. In sequence, a global data-flow fixed-point algorithm based on Equations 3.9 – 3.10 is then executed. After that, another Local DFA pass takes place for each block B that uses $OUT[B]$ to compute the access sets at each program point internal to B. For the sake of clarity, the pseudo-code of DCA is shown in Listing 3.1 which details each one of these steps. Moreover, the code of the Figure 3.2 is also used to illustrate the working of DCA.

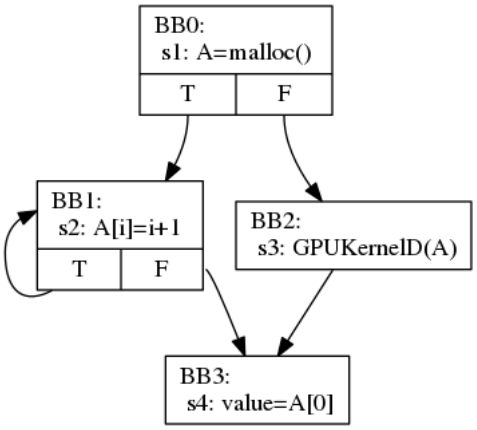
$$\begin{aligned} GEN[B] &= IN[s]\ \mid\ s\ \in\ FIRST[B] \\ IN[B] &= GEN[B] \end{aligned} \tag{3.13}$$

The DCA algorithm described in Listing 3.1 runs in three steps for each function F in the Module. A module has list of functions, a list of global variables, and others useful informations that composes a source file. The first step of DCA performs local analysis for all sentences of each basic blocks that belongs to the function F, according to lines 3—8. For each basic block BB, it starts from its last sentence, and propagate

Listing 3.1: DCA algorithm

```
DCA(Module){
  foreach F in Module {
    foreach BB in CFG {
      foreach s {
        BB.s.OUT = ⊎_{S ∈ Succ[s]} IN[S];
        BB.s.IN = GEN[s] ⊎ (OUT[s] - KILL[s]);
      }
    }

    foreach BB in CFG
      s0 = BB.First();
      BB.IN = BB.s0.IN;
      BB.GEN = BB.IN;
    }

    do{
      check = FALSE;
      foreach BB in CFG {
        BB_old.OUT = BB.OUT;
        BB_old.IN = BB.IN;

        BB.OUT = ⊎_{S ∈ Succ[B]} IN[S];
        BB.IN = GEN[B]⊎(OUT[B] - KILL[B]);
        if(BB_old.OUT ≠ BB.OUT || BB_old.IN ≠ BB.IN)
          check = TRUE;
      }
    }while (check);

    foreach BB in CFG {
      sx = BB.Last();
      BB.sx.OUT = BB.OUT;
    }

    foreach BB in CFG {
      foreach s in BB {
        BB.s.OUT = ⊎ IN[Succ s];
        BB.s.IN = GEN[s] ⊎ (OUT[s] - KILL[s]);
      }
    }
  }
```

Figure 3.2: Example to illustrate DCA, SBA and DCO algorithms.



accesses backwards until it reaches the first sentence of the block. In this first step, the last sentence of the basic block starts with OUT[s] empty, as accesses at this points were not propagated in the CFG yet. Sets $OUT[s]$ and $IN[s]$ are described, respectively, by Equations 3.5 and 3.6.

The second step of DCA starts setting $IN[B]$ and $GEN[B]$ of each basic blocks equal to the $IN[S]$ of its first sentence, according to lines 10—14. After that, it runs the global analysis, propagating informations through the basic blocks, according to the lines 16—27.

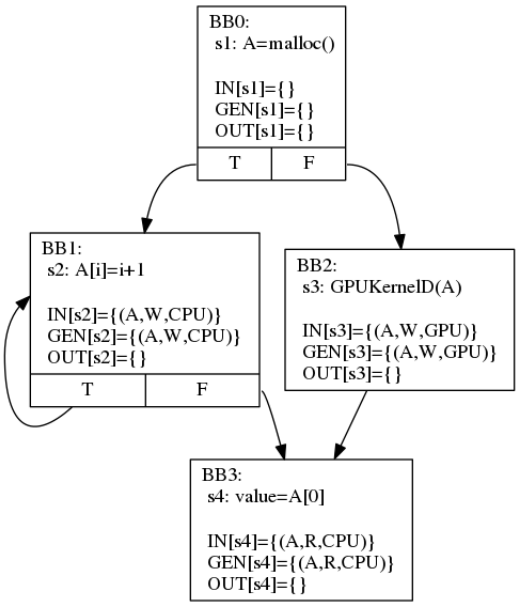
The first and second steps described above, respectively implement Local and Global DCA, by applying the meet operator according to the Table 3.3 to define a new access type and the Table 3.4 to define the new access device. Both also apply the combiner operator to generate the $IN[s]$ of all sentences and $IN[B]$ of all basic blocks.

The last step applies Local DCA again; but, before it, the OUT[s] of the last sentence of each basic block is updated according to the $OUT[B]$ of the basic block.

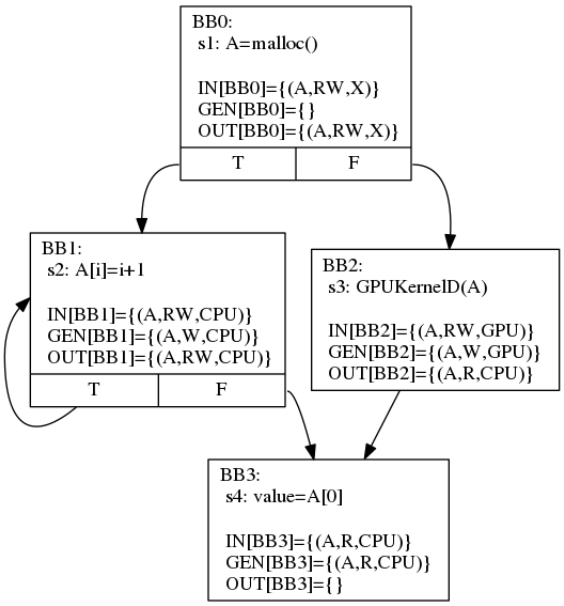
For sake of clarity, consider the Figure 3.3 as a example of the three steps of the DCA algorithm. To keep it simple, we are using only one sentence per basic block. Figure 3.3a shows the sets generated by applying the first step of DCA. One particularity of this first step is that the last sentences of each basic block started its $OUT[s]$ as an empty set. The second step, illustrated at Figure 3.3b, shows how the global DCA applies the data flow equations at the basic block level. Observe that the basic block $BB3$ started its $IN[BB3]$ and $GEN[BB3]$ equal to the $IN[s]$ of its first sentence, that is $IN[BB3]$=$GEN[BB3]$={`(A,R,CPU)`}. Also in the second step, notice that the data flow equation propagates the $IN[BB3]$ to $OUT[BB2]$, so that $OUT[BB2]$={`(A,R,CPU)`}. The last step, as illustrated at Figure 3.3c, applies Local DCA again; however, it considers the $OUT[s]$ of the last sentence of block B equal to $OUT[B]$. For example, observe that $OUT[s3]$ is equal to $OUT[BB2]$. The last step of the DCA algorithm is applied in order to update the sentences of each basic block with the data that was captured during the global analysis.

Figure 3.3: Result of each step of the DCA algorithm.

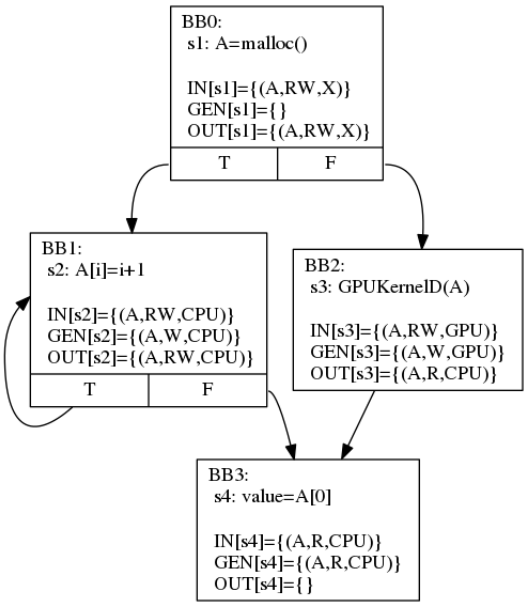
(a) First Step — Local DCA



(b) Second Step — Global DCA



(c) Third Step — Local DCA



3.2 Data Coherence Optimization (DCO)

The set of accesses at each program point resulting from DCA is used to implement Data Coherence Optimization (DCO). This is done by means of two optimization passes that run back-to-back on the program LLVM IR: (a) *Shared Buffer Allocation* (SBA) (see Section 3.2.1) — this pass replaces CPU allocated data (through `malloc` or `calloc`) by OpenCL shared buffers if the variables associated to the data are used by both CPU and GPU; and (b) MapUnmap Insertion (MUI) (see Section 3.2.2) to insert calls to OpenCL so as to keep variables coherent.

3.2.1 Shared Buffer Allocation (SBA)

The main goal of the SBA algorithm is to share buffers between CPU and GPUs. SBA works by analyzing the program to identify CPU and GPU buffers that can be merged into just one. Notice that DCA+DCO works upon the code generated by *GPUClang* compiler, transforming the host-side code, which uses offload as a coherence method, into a code that shares the same buffer between CPU/GPU.

Before starting running MUI algorithm, a tracking is performed to identify all CPU buffers used by any GPU computation so that it identifies which *malloc* or *calloc* the CPU buffer is associated. This is done to identify which buffers can be transformed in a shared buffer between CPU and GPU. This is an inter-procedural analysis, since one GPU execution can be in a function that does not allocate the CPU buffer. The pseudo-code to identify shared buffers was shown on Listing 3.2.

The SBA algorithm starts by looking at sentences that call data offload functions to identify which CPU variables are passed as a parameter to GPU kernels in `map` OpenMP clauses (line 3). With the result, it takes all CPU's variables that are passed as argument to these calls, and for each variable, it runs a function called *recursive_ud_chain* (line 13), that checks recursively the ud-chain of the variable until it reaches the *malloc* or *calloc* associated to it. This process is similar to the Reach Definitions Algorithm. This function takes the variable v identified by calling *identifyGPUBuffers* and applies a ud-chain on it. The result of this operation is a sentence s_i that is the definition of variable v. If the definition found is not an argument of a function, or a sentence that calls *malloc* or *calloc*, then a new ud-chain is applied for each variable that the sentence s_i uses. If one sentence that matches a *malloc* or *calloc* is found, then this sentence is stored in a set that is returned when the function finishes its execution. Otherwise, if it finds an argument of a function as a result of the *recursive_ud_chain*, then all caller functions are also analyzed, running on the same way, but starting from the argument passed as parameter. At the end, after no more iterations of ud-chain is possible to be executed, the function takes all variable pointers that have been reached and apply du-chain on them to try to identify, forward on program flow, if some of those pointers are associated with some *malloc* or *calloc*. This is necessary because it is possible that the CPU buffers are allocated in different functions that the variable pointers are defined. After applying all theses steps, and finding the CPU buffers associated to variable v, all reachable *malloc* or *calloc* are returned and analyzed to checked if they are suitable to be replaced by OpenCL

Listing 3.2: Shared Buffer Allocation algorithm

```
1  SBA(DCA_result){
2    foreach F{
3      variables[] = identifyGPUBuffers(DCA_result);
4      foreach v in variables[]{
5        cpu_buffer[] = recursive_ud_chain(v);
6
7        if(cpu_buffer[].size() > 1 && checkOffload(cpu_buffer))
8          return 0;
9
10       buffer = createSharedBuffer(cpu_buffer);
11       map(buffer);
12       update_DCA(v, buffer, DCA_result);
13     }
14   }
15 }
```

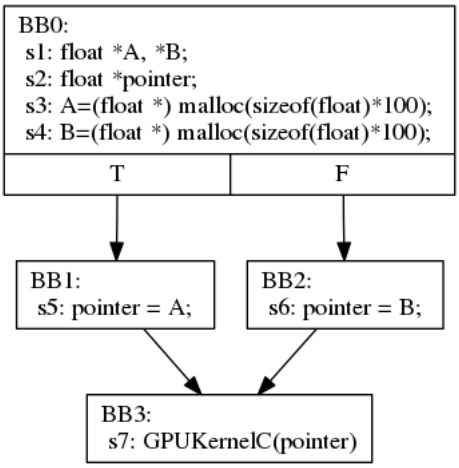
buffer creation calls.

Our goal with SBA is to share buffers between the CPU and the GPU if and only if a given variable, at the end of this process, points to only one *malloc* or *calloc* function. Lines 7—8 check if the *recursive_ud_chain* function returns more than one CPU pointer. If the variable points to more than one CPU buffer, then it is necessary runtime support to handle it, and SBA cannot be applied. If a given variable can point to more than one buffer, the identification of the current buffer can only be done at runtime. To illustrate better when this happens, consider the Figure 3.4. The SBA algorithm applying the *identifyGPUBuffers* function will identify variable *pointer*, at sentence $s7$. Applying function *recursive_ud_chain* to variable *pointer*, the first ud-chain will find sentences $s5$ and $s6$, where $s5$ points to A and $s6$ points to B. Applying one more time a ud-chain chain, but now upon variables A and B, the function will find sentences *s3* and *s4*, respectively, i.e., the algorithm reaches two CPU buffers. If SBA transforms both buffers into a shared buffer, it will be necessary to identify at runtime which flow is taken to set the argument to the correct buffer.

SBA is restricted to some programs where the data offload is performed only in a part of the CPU buffer. In order to keep the kernel function the same, without any transformation, SBA must guarantee that the data offload occurs from the first element of the CPU buffer being offloaded to the GPU. If this does not happens, then some modifications must be done in the kernel function so that its computation works in the correct positions of the shared buffer being create (adding offset in the buffers accesses). SBA does not work in these cases in order to avoid any kind of modifications in the kernel function. All these checks are done in line 7 of SBA algorithm by calling the *checkOffload* function.

Finally, once identified only one buffer that variable v points to, then a shared buffer between CPU and GPU is created overwriting the *malloc* or *calloc* by *clCreateBuffer* function using flag CL_MEM_ALLOC_HOST_PTR. A `map` upon the shared variable created is done in the sequence. Line 12 of the SBA algorithm updates all tuples of DCA_result, by taking all tuples that have variable v, and modifying them to *buffer*,

Figure 3.4: A variable pointing to more than one CPU buffer.



since that is the information used by the following MUI pass to insert `map` and `unmap` functions.

To clarify how the algorithm works, consider the Figure 3.4 with BB2 as an empty basic block. Taking variable *pointer* at $s7$; after applying ud-chain on it, sentence $s5$ results as the variable that defines *pointer*. Applying one more time ud-chain, but on variable A, that is used by sentence *s5*, it finds sentence $s3$, that creates a CPU buffer by calling *malloc*. This *malloc* at sentence *s3* is returned so that the shared buffer is created and mapped to CPU use.

The most complex case when applying SBA is when it needs to track programs that have CPU buffer allocation in a function other than the one running the GPU kernel. Another complexity resides in the applications where the allocation of a variable is made in a function different from the function that the variable was defined. In the case of programs that define the execution kernels for the GPU in the same function that creates the CPU buffers, usually the algorithm reaches the CPU buffer directly by applying a single ud-chain.

3.2.2 Map/Unmap Insertion (MUI)

After DCA analysis is performed each program point p has associated to it a set S_p containing all accesses that can be reached at some path starting at `p`. Therefore, one can use S_p to determine which future accesses a variable might have and thus identify the need to insert `map` or `unmap` instructions at `p`, an optimization pass of DCO that is called Map/Unmap Insertion (MUI). In order to achieve that, MUI visits the CFG starting at its first basic block towards its end. At each point `p` it compares the accesses in S_p against the value of *Current Access* (`CA`), a tuple that is used by MUI to indicate who owns the pointer to the shared buffer, and thus who controls its access. In a typical heterogeneous platform `CA = (v,[R,W,RW],[CPU,GPU])`, and before MUI starts `CA = (v,W,CPU)`, meaning that the CPU controls the first access to the buffer of shared variable `v`.

The application of `map` and `unmap` can generate unnecessary overheads when applied redundantly. The time taken by those functions are the time to maintain the CPU and

GPU cache memory updated, respectively. For example, the application of the `unmap` function can increase the time taken by the GPU computation when in fact it could be executed in the values that were in the cache memory previously its apply. MUI works conservatively applying `map` and `unmap` functions when necessary in order to maintain the program's semantic. MUI also applies both functions in a way to reduce as much as possible their redundant application.

Table 3.7: Call insertion during MUI.

CA	Tuple	Inserted Call	New CA
(v,R,CPU)	(v,R,CPU)	–	(v,R,CPU)
	(v,W,CPU)	–	(v,W,CPU)
	(v,RW,CPU)	–	(v,RW,CPU)
	(v,R,GPU)	–	(v,R,CPU)
	(v,W,GPU)	unmap	(v,W,GPU)
	(v,RW,GPU)	unmap	(v,RW,GPU)
(v,W,CPU)	(v,R,CPU)	–	(v,W,CPU)
	(v,W,CPU)	–	(v,W,CPU)
	(v,RW,CPU)	–	(v,RW,CPU)
	(v,R,GPU)	unmap	(v,R,GPU)
	(v,W,GPU)	unmap	(v,W,GPU)
	(v,RW,GPU)	unmap	(v,RW,GPU)
(v,RW,CPU)	(v,R,CPU)	–	(v,RW,CPU)
	(v,W,CPU)	–	(v,RW,CPU)
	(v,RW,CPU)	–	(v,RW,CPU)
	(v,R,GPU)	unmap	(v,R,GPU)
	(v,W,GPU)	unmap	(v,W,GPU)
	(v,RW,GPU)	unmap	(v,RW,GPU)

CA	Tuple	Inserted Call	New CA
(v,R,GPU)	(v,R,CPU)	map	(v,R,CPU)
	(v,W,CPU)	map	(v,W,CPU)
	(v,RW,CPU)	map	(v,RW,CPU)
	(v,R,GPU)	–	(v,R,GPU)
	(v,W,GPU)	–	(v,W,GPU)
	(v,RW,GPU)	–	(v,RW,GPU)
(v,W,GPU)	(v,R,CPU)	map	(v,R,CPU)
	(v,W,CPU)	map	(v,W,CPU)
	(v,RW,CPU)	map	(v,RW,CPU)
	(v,R,GPU)	–	(v,RW,GPU)
	(v,W,GPU)	–	(v,W,GPU)
	(v,RW,GPU)	–	(v,RW,GPU)
(v,RW,GPU)	(v,R,CPU)	map	(v,R,CPU)
	(v,W,CPU)	map	(v,W,CPU)
	(v,RW,CPU)	map	(v,RW,CPU)
	(v,R,GPU)	–	(v,RW,GPU)
	(v,W,GPU)	–	(v,RW,GPU)
	(v,RW,GPU)	–	(v,RW,GPU)

Now, consider for example that `CA = (u,W,CPU)` at the entry of `B0` in Figure 3.1(a). When MUI reaches `p` it notices that $S_p = \{$`(u,W,CPU)`$\}$ and thus nothing needs to be done. On the other hand, when MUI moves to `p+1` it notices that $S_{p+1} = \{$`(u,R,GPU)`$\}$ and thus a future read access will be performed by the GPU. Hence, an `unmap` operation needs to be inserted on the edge to `B1` in order to flush the values of `u` updated by the CPU and to hand the shared pointer to the GPU. As a result DCO updates the value of `CA` to `(u,R,GPU)`. Assume that the flow of execution moves from `B0` to `B1`. Given that `CA = (u,R,GPU)` nothing needs to be done at the entry of `B1`, and `CA` value continues `(u,R,GPU)`.

MUI uses Table 3.7 in order to generalize the insertion of `map` and `unmap` calls at each program point. That table shows for each value of `CA` and access $A_i \in S_p$ at program point `p` the resulting call which should be inserted at that point. Observe that `map` and `unmap` calls not only have the role of keeping coherence but are also used to transfer buffer control between CPU/GPU.

How MUI works is discussed below. For clarity, the pseudo-code for MUI is shown in Listing 3.3.

MUI starts its computation from the information captured by DCA. It also uses all the shared buffers that were created by SBA. To each shared buffer, MUI makes the coherence of it at each function by inserting `map` and `unmap` functions. In this version, the algorithm is conservative, that is, it ensures that every function of the program starts and ends with the shared buffers mapped to CPU.

Lines 5–7 in the pseudo-code shows that the first basic block from function F is taken to be optimized. The algorithm starts setting its CA with access device equal to `CPU`

and access type equal to `W`. Following, lines 9—12, some sets are initialized to hold some important data during MUI execution. Set *reachableBB* is used to identify basic blocks that are ready to be optimized so that it applies `map` and `unmap` functions between its sentences and its successors. A basic block BB is said ready to be optimized when all its successor's were optimized by MUI; in other words, when all its successors' CA are defined. The *visitedBB* set is used to track all basic blocks already optimized and the *checkAfter* set holds all basic blocks that are reachable, but do not have the CA defined. Lines 15—90 do the insertion of `map` and `unmap` function calls for each basic block BB that belongs to function F, starting from its first basic block. Lines 26—29 identify the transitions CPU↔GPU between the sentences of the basic block BB, and insert `map` or `unmap` according to Table 3.7.

The next step is to identify which basic blocks are ready to be optimized. MUI starts by checking all successors of BB seeking its devices captured during the DCA algorithm. The possible devices are `CPU`, `GPU` and `X`. As shown in lines 35—44, if the successor's device is defined as `CPU`, and the CA of BB has been finished with `GPU`'s device, then it is necessary to insert a `map` call between BB and its successor. Otherwise, as shown in lines 45—55, the inverse occurs when the successor's device is defined as `GPU`, being necessary to insert an `unmap` call between BB and its successor. Observe that in both cases a new basic block may be necessary, depending on the number of predecessors of BBSucc. This is necessary to avoid conflicts in different execution flows. Finally, when the successor's device is undefined, i.e., defined as `X` by DCA, some more checks become necessary. First, if the number of predecessors of BBSucc is just one (lines 57—59), it means that only one execution flow is possible to reach BBSucc so that its CA can be updated using the CA of its unique predecessor. Otherwise, if the number o predecessors of BBSucc is larger than one, it is necessary to check if all predecessors of BBSucc were optimized, so that it is possible to identify if it is necessary to insert `map` or `unmap` calls between BBSucc and its predecessors. To check this, line 61 identifies if all predecessors of BBSucc were optimized. If at least one predecessor was not optimized at this point, BBSucc is inserted into *checkAfter* set. Notice that line 63 identifies if there is a combination of two CA equal to `CPU` and `GPU` among the predecessors of BBSucc. If so, then the access device of BBSucc is set with device equal to `CPU`, so that it becomes necessary to check the need of inserting a `map` call between its predecessors that finished with CA equal to `GPU`.

By definition, any variable enters and leaves all functions mapped to `CPU`. As so, MUI certifies, before starting to analyze the next variable, if the current variable v of function F is mapped to `GPU` at the last sentence of F, according to lines 92—95. If so, a `map` function is called to ensure that the variable finishes its execution mapped to the `CPU`.

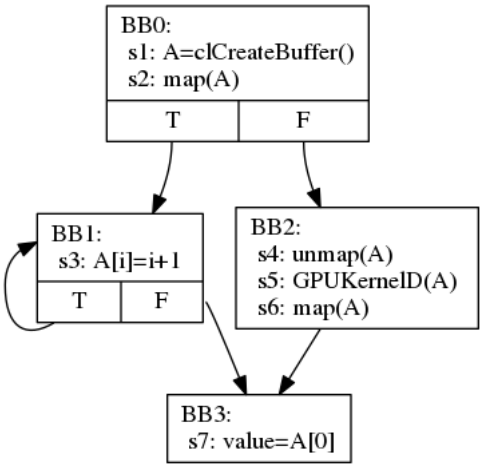
The algorithm finishes when all functions with all its basic blocks were already analyzed and optimized according to the variables that were previously identified and created as shared buffer between CPU and GPU.

Listing 3.3: DCO algorithm

```
MUI(DCA_result, Module){
  foreach F in Module {
    foreach v in variables[] {
      FirstBB = F.front();
      CA.device = CPU;
      CA.type = W;
      updateTopCA(FirstBB, v, CA);

      reachableBB.clear();
      visitedBB.clear();
      checkAfter.clear();
      reachableBB.insert(FirstBB);

      counter = 0;
      do{
        if(!reachableBB[counter]){
          BB = checkAfter.push_back();
          reachableBB.insert(BB);
          CA = checkPredecessors(BB);
          updateTopCA(BB, v, CA);
        }
        BB = reachableBB[counter];
        CA = getTopCA(BB, v);
        visitedBB.insert(BB);

        foreach s in BB{
          hasChanged = setMapOrUnmapIfNecessary(CA, s);
          updateCA(CA, hasChanged);
        }
        updateBottomCA(BB, v, CA);
        BBSuccs[] = getSuccessor(BB);
        foreach BBSucc in BBSuccs[] {
          numPred = getNumberOfPredecessor(BBSucc);

          if(getINScopeBB(BBsucc, v) == CPU){
            if(CA.device == GPU){
              if(numPred > 1){
                newBB = createNewBB(BB, BBSucc);
                map(newBB, v, newBB.front());
              }
              else map(BBSucc, v, getLastSentence(BBSucc));
            }
            updateTopCA(BBSucc, v, CA);
          }
          else if(getINDeviceBB(BBsucc, v) == GPU){
            if((CA.device == CPU && CA.type == W) ||
               (CA.device == CPU && getINTypeBB(BBsucc, v) != R)){
              if(numPred > 1){
                newBB = createNewBB(BB, BBSucc);
                unmap(newBB, v, newBB.front());
              }
              else unmap(newBB, v, getLastSentence(BBSucc));
```

```
53              }
54              updateTopCA(BBSucc, v, CA);
55            }
56            else if(getINDeviceBB(BBsucc, v) == X){
57              if(numPred == 1){
58                updateTopCA(BBSucc, v, CA);
59              }
60              else if(numPred > 1){
61                if(checkIfAllPredWereVisited(BBSucc)){
62                  newCA.type = getCATypeCombination(BBSucc);
63                  hasCombination = getCombination(BBSucc);
64
65                  if(hasCombination){
66                    newCA.device = CPU;
67                    BBPreds[] = getPredecessors(BBSucc);
68                    foreach BBPred in BBPreds[]{
69                      if(BBPred.bottom.v == GPU){
70                        createNewBB(BBPred, BBSucc);
71                        map(newBB, v, newBB.front());
72                      }
73                    }
74                  }
75                  else newCA.device = CA.device
76
77                  updateTopCA(BBSucc, v, newCA);
78                  if(checkAfter.find(BBSucc))
79                    checkAfter.remove(BBSucc);
80                }
81                else checkAfter.insert(BBSucc);
82              }
83            }
84          }
85
86          foreach BBSucc in BBSuccs[] {
87            if(!visitedBB.find(BBSucc) && !checkAfter.find(BBSucc))
88              reachableBB.insert(BBSucc);
89           counter++;
90        }while(!visitedBB.allBB())
91
92        BB = getLastBasicBlock(F);
93        CA = getBottonCA(BB,v);
94        if(CA.device == GPU)
95          map(BB, v, getLastSentence(BB));
96      }
97    }
98  }
```

Figure 3.5: Result of applying SBA and MUI after DCA.



The result achieved by applying MUI, after applying DCA and SBA, in the CFG of the Figure 3.2, is showed in Figure 3.5. Before, applying SBA on the code, variable A used by function *GPUKernelD* is taken, so that it attempts to find the sentence where the CPU buffer is created. Observe that it is found in sentence *s1* of the code of Figure 3.2, where the CPU buffer is created by calling *malloc*. Once found, the *malloc* is overwritten by *clCreateBuffer*, so that this buffer becomes shared between CPU and GPU. After that, a `map` function is called, according to sentence s2 of Figure 3.5. With the shared buffer created, MUI is then applied. $BB0$ starts its CA equal to (CPU,W). Basic block $BB1$, that has a tuple with variable A with CPU device in its $IN[BB1]$, does not require neither `map` and `unmap` calls, since basic block $BB0$ has finished its CA with device also pointing to CPU. The contrary happens with $BB2$, that has variable A with device GPU in its $IN[BB2]$; hence it requires an `unmap` function call. In sequence, $BB2$ requires a `map` call since basic block $BB3$ has variable A pointing to CPU.

Chapter 4

Related Works

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* architecture. Nilakant *et al.* [31] 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.* [21] showed a 30% improvement in the overall execution time when running the real-time image processing application SLAM on an integrated CPU-GPU architecture of a mobile device [31]. Shen *et al.* [33] also reached good speed-ups by reducing more than 80% of the transfer time through an adequate usage of the OpenCL memory flag `CL_MEM_ALLOC_HOST_PTR`. These works require the programmer to directly deal with the problem of sharing and making the data between CPU-GPU coherent. In *GPUClang*/DCO this task is automatically handled by the compiler.

Jablin *et al.* [27] proposed an approach to automatically manage and optimize CPU-GPU communication. They proposed a system called CPU-GPU Communication Manager (CGCM) that removes from the programmer the task of manually manage data transfers in a separate CPU-GPU architecture. Their solution maps a buffer to GPU by allocating memory and copying it to the GPU memory automatically. Their approach has a function called unmap, to update the CPU buffer with the data modified by the GPU. Likewise, our work uses `map` /`unmap` functions, but our goal is to make coherence of a buffer physically shared between CPU and GPU.

Jablin *et al.* [26] 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; moreover, the data transfer from CPU to GPU only happens when needed, and the data transfer from GPU to CPU only occurs on data that was modified by the GPU. *GPUClang*/DCO avoids the automatic data transfer between CPU and GPU thus ensuring coherence, as only one buffer is created and shared between them.

Various approaches have been tried before to raise the level of abstraction of OpenCL programming. Thouti *et al.* 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 [35]. The limitation is that it only works on function calls. Thouti has mentioned about the use of shared buffer between CPU and GPU, but they chose to use offloading.

To make GPU programming easy, Lee *et al.* [28] developed a source-to-source solution from OpenMP directives to CUDA. Their solution extract marked regions to be executed on GPUs. They developed an algorithm that reduces CPU-GPU memory transfers by only copying back the data modified inside the kernel region that is needed by the CPU afterwards. They also developed similar features as in [24]. Different from their solution, our work shares data buffers between CPU and GPU, without the need of memory transfers.

Said *et al.* developed hiCL [32], an abstraction layer that was developed on top of OpenCL in order to reduce the programming burden, simplifying the memory management and the kernel execution. They developed functions to map buffers physically shared between the CPU and GPU; however, their solution only abstracts the OpenCL complexity, letting the programmer the work of make the coherence annotations in the code.

Becchi *et al.* [40] developed a runtime library that automatically schedule kernels to the GPU. Besides that, their solution identifies when it is worth to send a computation to the GPU; it analyzes the data size, during runtime, to decide if the computation will be executed on CPU or GPU. Also, their solution manages data placement, so that data offload is done only when it is necessary. The solution requires data offload and besides that, it also uses a runtime library to decide when to offload the data to GPU and when to execute a computation in both CPU and GPU.

Some solutions that translate OpenMP to OpenCL or CUDA have been developed. Grewe *et al.* [41] developed a solution using this approach that generates an optimized kernel. Also, their approach is capable to determine when to move a computation to the GPU by using machine learning. Ferrer *et al.* [42] developed a solution called OMPSs that uses the same approach to offload execution. However, unlike the DCO optimization described in this work, both solutions also require offloading data to the GPU.

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) [22]. Their approach maintains a shared logical memory space for CPUs to access an object in the physical memory of the accelerator, but not vice versa. They also focused on portability so that their library run on different devices. This assures coherence between CPU-GPU by means of duplicated memory spaces, one that is hosted on the device and other on the host; hence, so that memory copies are necessary to update both buffers. Furthermore, in their solution the programmer needs to manually assign coherence annotations. Our work proposes an approach in which CPU and GPU share the same address space and programmers do not need to insert coherence annotations.

Henry *et al.* proposed the SOCL platform that enables the coordination of multiples devices [25]. SOCL performs data transfers and kernel scheduling automatically. Their solution handles load-balancing issues and maintains the data coherency across all devices by performing appropriate data transfers among all devices. Their solution works on multiples devices that need memory transfers while our solution aims at integrated CPU-GPU architectures with shared memory.

Han *et al.* developed directives called hiCUDA, a high-level directive-based language for CUDA programming [24]. Their solution works with pragmas, using several CUDA features as well as shared scratchpad memories. Their approach performs a source-to-source translation that relieves the programmer from the sometimes tedious and error-prone task of writing device code. Like other approaches, their solution needs explicit data transfers before dispatching the kernel and after performing the computation.

NVIDIA have included new mechanisms as Unified Memory and Page Migration Engine in CUDA 8 [11]. Buffers created using CUDA 8 works as a single pointer accessible anywhere (by both CPU/GPU). With this resource, programs running using CUDA 8 works without the need of memory transfer. They have also created ways to access the single pointer simultaneously by both CPU and GPU. Their solution works in a page-fault fashion, where the coherence is made by the CUDA driver in page granularity. Their solution has already solved the coherence between CPU/GPU for discrete GPUs. Our work propose a solution to solve coherence in software for integrated GPU.

Chapter 5

Experimental Evaluation

This section presents an experimental evaluation of the Data Coherence Optimization (DCO) implementation in *GPUClang*. Section 5.1 describes the infrastructure and methodology used in the experiments and Sections 5.2 – 5.3 report their results.

GPUClang and DCO have 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 well-known Polybench benchmark suite with standard input sizes [6]. The programs have been re-written in OpenMP 4.X. For the sake of simplicity we refer to this set of modified programs as the *Polybench* suite.

5.1 GPUClang Environment

Although OpenCL provides a library that eases the task of offloading kernels to devices, its function calls are complex, have many parameters and require the programmer to have some knowledge of the device architecture's features (e.g. block size, memory model, etc.) in order to enable a correct and effective usage of the device. Hence, OpenCL can still be considered a somehow low-level library for heterogeneous computing.

Introduced through OpenMP 4.0 the new *OpenMP Accelerator Model* [14] proposes a number of new clauses aimed at speeding up the task of programming heterogeneous architectures. 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. Although most OpenMP directives used for multicore hosts can also be used inside target regions, the new accelerator model easies the tasks of identifying data-parallel computation.

GPUClang is an LLVM/Clang based compiler aimed at implementing the OpenMP Accelerator Model. It adds an *OpenCL runtime library* to LLVM/CLang that supports OpenMP offloading to devices like GPUs and FGPAs. The kernel functions are extracted

from the OpenMP region and are dispatched as OpenCL [2] or SPIR [3] code to be loaded and compiled by OpenCL drivers before being executed by the device. This whole process is transparent and does not require any programmer intervention.

GPUClang 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.

Listing 5.1 presents two loops from `mvt` program of the Polybench [6] benchmark suite after they have been annotated with OpenMP 4.X 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 5.1, 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. For example in the first kernel of Listing 5.1 inputs (`a` and `y1`) are mapped *to* the GPU, and array `x1` is mapped *to/from* the GPU. This means that array `x1` is read and written during the kernel execution in the GPU. This strategy offers maximal flexibility to the developer decide what part of the code is profitable to run on which architecture.

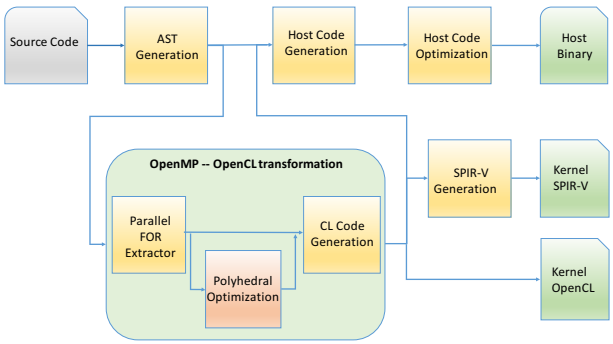
Listing 5.1: Piece of Polybench mvt benchmark application

```
1  // Problem size
2  #define N 8192
3
4  void mvt_gpu(float* a, float* x1, float* x2,
5  float* y1, float* y2)
6  {
7    int i,j;
8
9    #pragma omp target device (GPU)
10   #pragma omp target map(to: a[:N*N], y1[:N]) map(tofrom: x1[:N])
11   #pragma omp parallel for
12   for (i=0; i<N; i++)
13     for (j=0; j<N; j++)
14       x1[i] = x1[i] + a[i*N + j] * y1[j];
15
16   #pragma omp target map(to: a[:N*N], y2[:N])  map(tofrom: x2[:N])
17   #pragma omp parallel for
18   for (i=0; i<N; i++)
19     for (j=0; j<N; j++)
20       x2[i] = x2[i] + a[j*N + i] * y2[j];
21 }
```

Figure 5.1 shows the *GPUClang* compiler pipeline. The OpenMP-OpenCL transformation pass generates two outputs. First, *GPUClang* extracts annotated loops from the AST and transforms them to OpenCL kernels in source code format. The generated kernel can also go through the SPIR generation pass to produce kernel bit code in the SPIR format. The second output describes which parameters should be passed to the kernel and the number of blocks and threads that it will run in the device. This information is used by the compiler code generator to increase kernel performance. *GPUClang* leverages on ISL [30] polyhedral model optimizations [29] 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 GPU accelerators. As an example, tiling can be directly applied to the loops of Listing 5.1, as outermost loops can be executed in parallel because their iterations update disjoint parts of the `x1` and `x2` arrays.

Figure 5.1: *GPUClang* compiler pipeline.



5.2 DCO Performance Analysis

One of the claims of this work is that DCO brings an improvement over the regular data offloading/coherence mechanism. To evaluate that, the OpenCL runtime library, a component of the *GPUClang* compiler, 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 5.2a – 5.2d: (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 5.2a – 5.2b 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 5.2c – 5.2d show the results after applying DCO to the *Polybench* 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. For example, the *mvt* application in Listing 5.1 (Section 5.1) has two kernel functions that are dispatched to the GPU. The first kernel computation on Intel/Iris spent *187* ms on *8192* square array of float values while offloading takes *43* ms (*23*%). The second kernel computation is *173 ms* while offloading consumes *47* ms (*27*%). Overall the time to offload data to these two kernels is *25*% of *mvt* total execution time (*360* ms) and this is entirely eliminated after DCO is applied (Figures 5.2c – 5.2d).

Figures 5.2a – 5.2b also reveal that the `OpenCL` driver takes an astonishing share of the total execution time on most *Polybench* programs (`OpenCL` bar). As shown in the figure, this effect is more pronounced in the ARM/Mali architecture meaning that the OpenCL driver for this architecture needs some performance improvement.

Figures 5.2a – 5.2c and Figures 5.2b – 5.2d show that DCO can remove most of the program offloading overhead. In order to evaluate its corresponding performance gains,

Figure 5.2: The breakdown of total execution time: (a) & (b) before DCO optimization (c) & (d) after DCO optimization

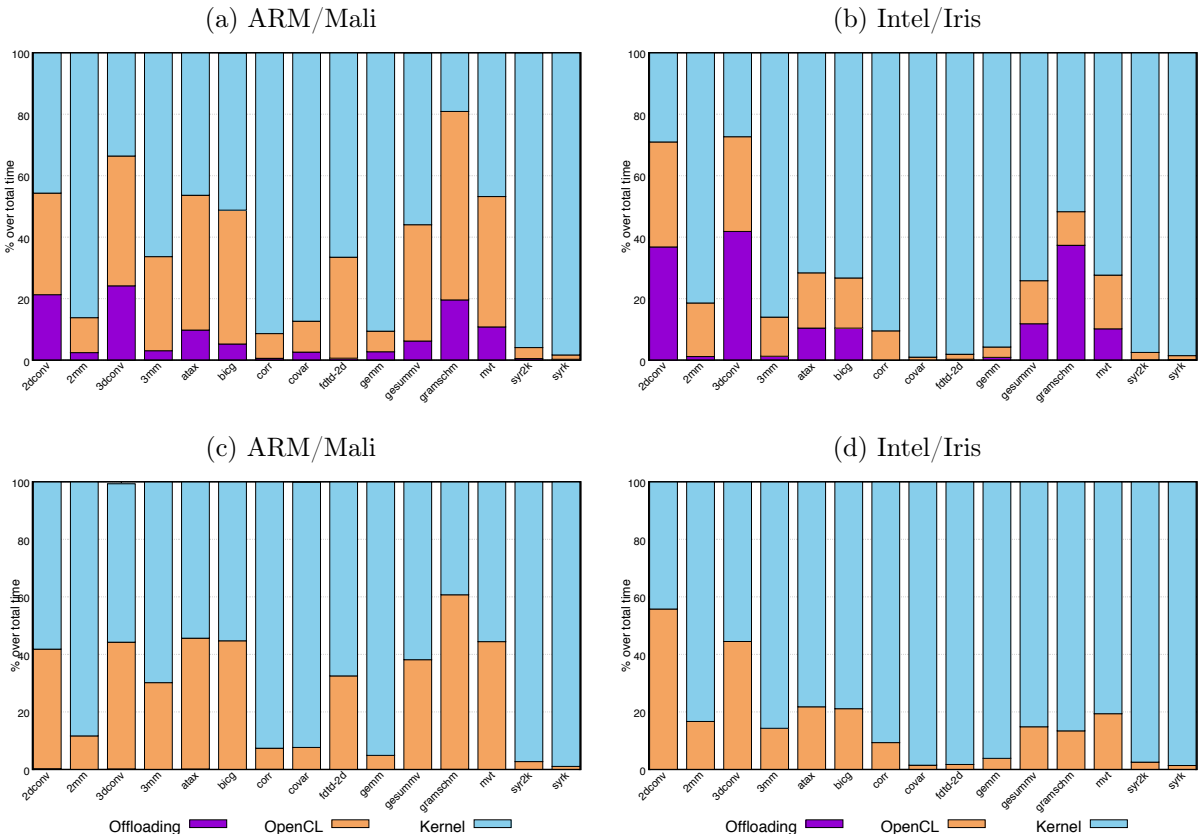


Figure 5.3: *GPUClang*+DCO Speedup with respect to *GPUClang* (both -O2 -opt-tile).

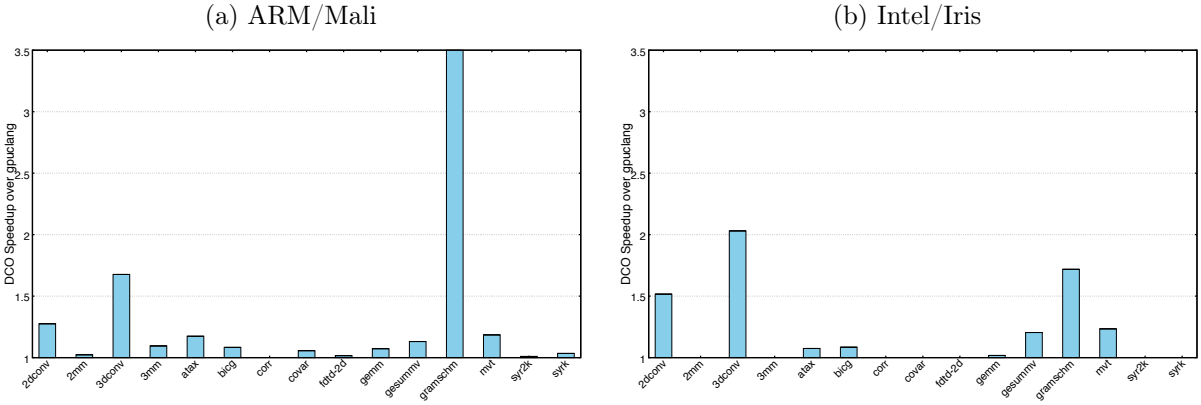


Table 5.1: Absolute runtime & speed-ups for *Polybench* benchmark suite.

Benchmarks		ARM/Mali					Intel/Iris				
		Sequential (-O2)	GPUClang (-O2 -opt-tile)		GPUClang+DCO (-O2 -opt-tile)		Seq. (-O2)	GPUClang (-O2 -opt-tile)		GPUClang+DCO (-O2 -opt-tile)	
name	size	Time (sec)	Time	Speedup	Time	Speedup	Time	Time	Speedup	Time	Speedup
2dconv	8192	1.71	4.54	0.38	3.56	0.48	0.37	0.22	1.66	0.15	2.52
2mm	1024	123.19	2.33	52.87	2.27	54.27	12.04	0.29	41.18	0.30	40.45
3dconv	512	7.20	10.77	0.67	6.42	1.12	0.86	0.55	1.54	0.27	3.14
3mm	1024	39.54	1.50	26.36	1.37	28.86	18.03	0.34	53.56	0.37	49.19
atax	8192	0.29	2.16	0.13	1.84	0.16	0.10	0.34	0.28	0.32	0.30
bicg	8192	0.49	1.98	0.25	1.82	0.27	0.23	0.34	0.65	0.32	0.71
correlation	1024	5.33	3.17	1.68	3.17	1.68	1.35	0.64	2.10	0.66	2.06
covariance	2048	94.74	3.75	25.26	3.55	26.69	69.39	3.71	18.68	3.77	18.42
fdtd-2d	2048	12.99	12.73	1.02	12.53	1.04	3.17	2.81	1.12	2.83	1.12
gemm	2048	198.35	2.55	77.78	2.37	83.69	85.36	0.95	89.64	0.94	91.23
gesummv	8192	0.50	4.14	0.12	3.66	0.14	0.33	0.60	0.54	0.50	0.66
gramschmidt	512	1.05	14.28	0.07	2.72	0.39	0.95	0.55	1.73	0.32	3.00
mvt	8192	2.15	2.14	1.00	1.81	1.19	0.16	0.36	0.43	0.29	0.54
syr2k	1024	2.93	4.67	0.63	4.63	0.63	2.51	2.06	1.21	2.10	1.20
syrk	2048	11.96	16.41	0.73	15.87	0.75	9.42	3.65	2.57	3.82	2.47

we measured the speedup of each application with DCO over the same application without applying this optimization. The results in Figures 5.3a – 5.3b indicate that DCO reduces the total execution time of benchmarks with large read and write data sets. Examples are *2dconv* (up to *1.5x* on Intel/Iris) and *3dconv* (*2.0x*), *gramschmidt* (up to *5.3x* on ARM/Mali), *atax* (*1.2x*) and *mvt* (*1.2x*).

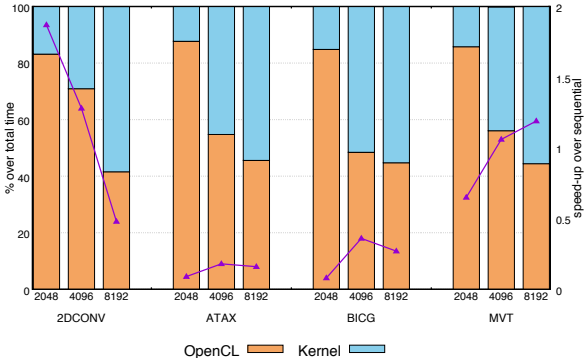
The high speed-up achieved by *gramschmidt* is caused by the elimination of multiples data offload that is performed during its execution, where most of them are performed without any need. By eliminating all those data offload, the GPU can works better since it may be benefited by the data being in cache among others architectural issues related. In our experiments, we have noticed that besides eliminating almost 2.8 seconds of the time taken by data offload, DCO also have reduced the time taken by the GPU execution from 2.73 to 1.07 seconds. It also benefits from buffer creation, since that *GPUClang* create and release the buffers for each iteration.

5.3 Data Size Analysis

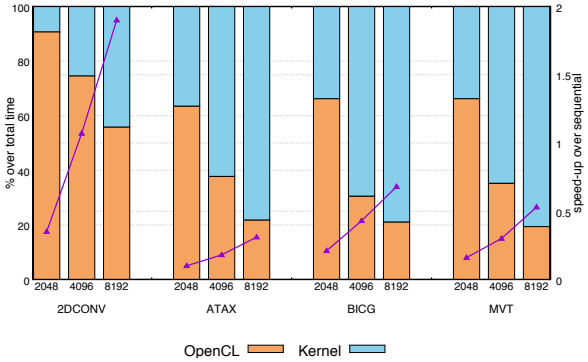
Table 5.1 shows absolute runtime numbers for the *Polybench* programs, in three different setups: (a) Sequential execution; (b) Parallel execution using *GPUClang*; and (c) Parallel execution using *GPUClang* with DCO. As expected, substantial speed-ups have been produced in some of the programs that run the longest times. The slowdowns that can be observed in benchmarks such as *2dconv*, *atax*, *bicg* and *mvt*, happened in instances that execute for a very short time. In such cases, the extra parallelism achieved by *GPUClang* with or without DCO optimization is not enough to pay off for the time to create and manage buffers in shared memory. 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, Figures 5.4a – 5.4b show that longer executions times can amortize the OpenCL overhead. The immediate effect is a decrease of slowdown or even an increase in the speed-up relative to the sequential execution for the Intel/Iris architecture, as shown in Figures 5.4a – 5.4b represented by points and lines on the graphs. For instance, *2dconv* benchmark shows a slowdown of *0.35x* for data size equals to *2048* and a speed-up of *1.90x* when the data size is *8192*. However is is not true on ARM/Mali architecture. The increase

Figure 5.4: `OpenCL` overhead variation with the data set size.

(a) ARM/Mali



(b) Intel/Iris



of slowdown for greater sizes, mainly in the *2dconv* benchmark can be explained by the memory limitation of mobile devices.

Also according to the Table 5.1, it is possible to observe that the DCO algorithm enables to achieve speed-up in programs that have slowdown with *GPUClang* without optimization. As an example, consider the execution of *3dconv* on ARM/Mali. The *GPUClang* with DCO achieves speed-up of 1.12x. Otherwise, when this application was compiled with *GPUClang* without the DCO optimization, we obtained a slowdown of 0.67x.

The benefit of DCO becomes clear as the complexity of the algorithms and the sizes of the data sets increase. "Gram-Schmidt decomposition" (*gramschmidt*) is an example. The kernel functions were extracted from inner loops and the offloading version looses performance with repeated data movement and GPU buffer creation and destruction, which 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.

DCO also improves the usage of memory space when compared to standard host size allocation since the buffers are created only in the device shared memory. For example, without DCO the execution of the benchmark *3dconv* with 512 elements in each dimension requires 2.5 GB of memory space, since the data is stored in the CPU memory and then copied to GPU memory upon execution. When applying DCO CPU and GPU share the same buffer and thus the memory requirement for execution is reduced to 1.5 GB.

When you increase the program data size, the relative time represented by the *kernel execution* tends to increase, while the relative time represented by the *OpenCL* tends to reduce. An experiment performed on ARM/Mali (see Figure 5.5) show the *OpenCL associated overheads are those that achieve significant reductions*. This time becomes insignificant when compared with the time of *kernel execution* and *data offload*. Moreover, for most programs the time taken by data offload remains almost constant when increasing the program data size.

Moreover, we also noticed that programs that spend more time executing the kernel computation do not benefit from DCO optimization. In such cases, as program data size

Figure 5.5: `Data Offload` overhead variation with the data set size (-opt-vectorize).

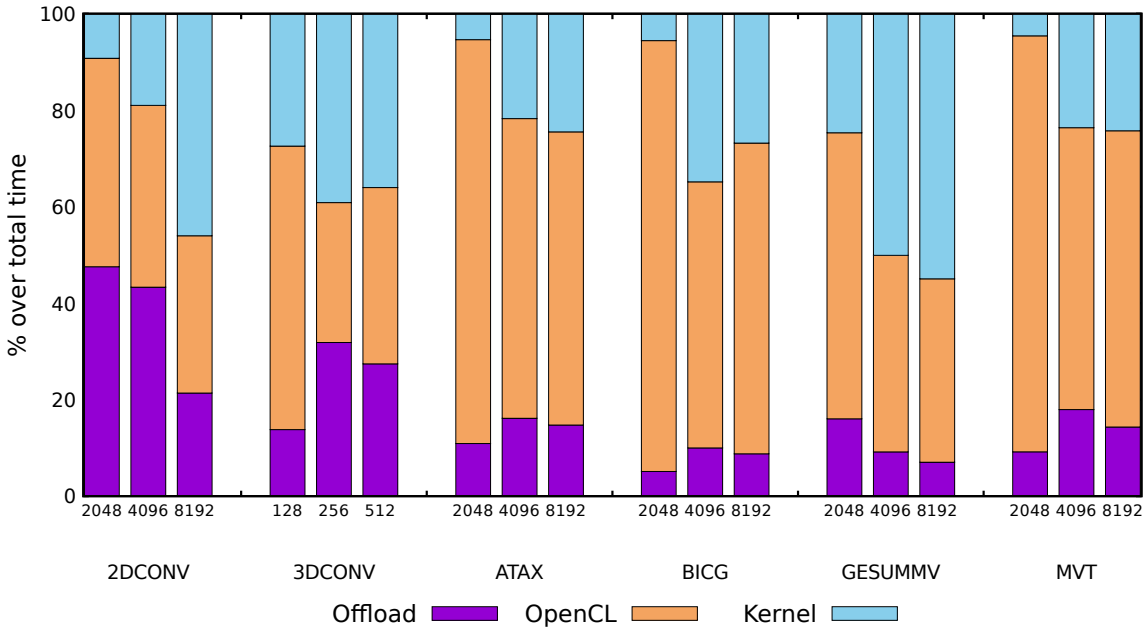


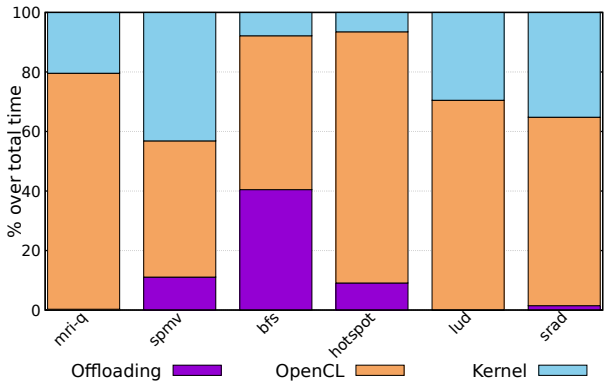
Table 5.2: Absolute runtime & speed-ups for *Parboil* and *Rodinia* benchmark suite.

Benchmarks name	Sequential Time (sec)	GPUClang Time	GPUClang Speed-up	GPUClang+DCO Time	GPUClang+DCO Speed-up	Speedup GPUClang/DCO
mri-q	25.33	5.32	4.75	5.07	4.99	1.05
spmv	2.77	0.61	4.52	0.45	6.17	1.37
bfs	0.40	1.34	0.29	0.49	0.82	2.75
hotspot	0.09	0.89	0.09	0.78	0.11	1.14
lud	2.03	5.80	0.34	5.76	0.35	1.01
srad	2.84	4.59	0.61	4.30	0.66	1.07

increases, the time taken by the *kernel execution* also increases, sometimes exponentially, as for example in*syr2k* and *syrk* applications. On the other hand, when GPU computation complexity is low, and data offloading is high, increasing the size of the data increases the speed-up achieved by applying the DCO optimization. Now, if the memory used by the application is extremely large so that it generate cache misses, page fault, etc, DCO optimization can become a solution because it reduces the data size by combining the CPU and GPU buffers in just one shared buffer.

DCO also works well when applied on more complex benchmarks, such as Rodinia[8] and Parboil[7]. We did an experimental evaluation of these benchmarks on ARM/Mali, where each program was executed with the highest data set available. Table 5.2 shows the results for absolute runtime and speed-ups and Figure 5.6 shows, for each program, how the time is distributed among *OpenCL Offloading* and *Kernel*. Notice that for all programs, most of the time is taken by *OpenCL*. It happens because all programs execute their kernel thousands of times, since the GPU computation is inside a loop generating high overhead for the OpenCL Driver to manage it. Programs *bfs* and *hotspot* deserve more attention because their kernels are called thousand of times, and thus at each iter-

Figure 5.6: The breakdown of total execution time without DCO optimization.



ation data offload is performed. This overhead can be removed when applying the DCO optimization.

One can observe that after DCO is applied to *bfs* and *spmv* speed-ups of 2.75x and 1.37x respectively result. Also notice that offload represents a big portion of the execution times of both programs, what is due to the fact that buffers are created multiple times within a loop. After applying DCO the resulting optimized program creates shared buffers only once what explains the improved speed-ups. The other programs were not affected by DCO because all them create buffers before entering the loop that executes multiple times the same kernel.

Chapter 6

Conclusions and Future Works

This dissertation described DCAO, a Data Coherence Analysis and Optimization, that has its root in the observation that making variables used by both CPU and GPU shared, one can avoid unnecessary data offload. DCAO built atop of the *GPUClang* compiler was evaluated using the well-known and widely accepted *Polybench* benchmark suite. Preliminary results show that DCAO indeed improves the speedup of applications with large datasets or medium-to-large kernel duration. We also did some tests with *Rodinia* and *Parboil* benchmarks and the preliminary results are well aligned with the *Polybench*.

The experimental evaluation indicates that implementing DCAO integrated with *GPUClang* can yield significant performance benefits. By combining dataflow analysis, shared buffer allocation and map/unmap insertion the resulting optimization pass overcame the data movement costs and delivery non-trivial performance gains in heterogeneous architectures. We achieved up to 5.3x of the speed-up when compared with *GPUClang* without DCAO. When compared with the serial version, we have achieved up to 91.2x speed-up.

The results obtained with *Polybench* also confirm that the proposed approach points to the right direction. As a future work, one can modify DCAO algorithm to use interprocedural data-flow information. With this, instead of generating the $GEN[s]$ of a given call instruction s with a tuple equal to A_i=`(v,W,CPU)`, the DCA algorithm should first analyze the called function F so that all tuples that reaches its first sentence can form the $GEN[s]$ for all call instructions of the callee function that call F. By applying interprocedural DCA could avoid the need to have the initial function device access mapped always to the CPU. As a result, the called function can follow the execution flow of the callee function. This could also avoid the need to apply DCO optimization on a function basis. In such cases, one could expand the called function, so that it can avoid finishing with the shared buffer pointing to the CPU. Hence, after analyzing the entire called function, the CA of the last sentence of the called function can be propagated to the next instruction of the callee function, improving the analysis.

Most programs used in the experiments initialize its buffers pointing to CPU before calling data offload functions to send its content to the GPU. When applying the DCAO optimization besides the buffer being created, a `map` function is inserted in the sequence to give the pointer of the shared buffer to the CPU. By considering that DCAO algorithm inserts a `map` function after creating a buffer, it does not generate extra overhead if the CPU first touches the buffer before making it available to the GPU. Otherwise, when the

GPU is the first device that touch the buffer, after being created, then an extra overhead is generated, since a `unmap` function should be called unnecessarily to make the buffer available to the GPU. This should be modified so that SBA can create a shared buffer without invoking a `map` function in the sequence. For that, DCAO must be modified so that CA accepts an undefined device (X).

As DCAO is applied in only one source file, Link Time Optimization (LTO) can be a solution to perform DCA+DCO in more than one source file, since there are several benchmarks that the kernel execution is in a separated source file. Other issue that should be evaluated, is when working in a benchmark that have complex data structures, e.g. vector of buffers, since DCA+DCO works only with simples *mallocs* and *callocs* structures.

Finally, another point that was observed when applying DCAO, is that *GPUClang* generates some extra overheads when creating a kernel inside a loop. This overhead happens since for each iteration, *GPUClang* repetitively calls some basic *OpenCL* functions that could be called just once. One alternative is to apply some heuristics based on Loop Invariant and Code Motion to move these *OpenCL* functions out of the loop, thus resulting only one call to the kernel execution.

Bibliography

[1] CUDA: A Parallel Computing Platform and Programming Model, 2015, *NVIDIA.* https://developer.nvidia.com/cuda-zone

[2] OpenCL: The Open Standard for Parallel Programming Language of heterogeneous Systems, 2010, *Khronos Group.* http://www.khronos.org/opencl

[3] SPIR: An OpenCL Standard Portable Intermediate Language for parallel compute and graphics. 2014, *Khronos Group.* https://www.khronos.org/spir

[4] OpenACC: Application Programming Interface. 2011, *Khronos Group.* http://www.openacc-standard.org/.

[5] OpenMP API Specification for Parallel Programming. Version 4.5 2015, *OpenMP ARB* http://openmp.org/wp/openmp-specifications/.

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

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

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

[9] Heterogeneous System Architecture. Version 1.1 2016, *HSA Foundation* http://www.hsafoundation.com/.

[10] The LLVM Compiler Infrastructure 2016, *LLVM* http://www.llvm.org/.

[11] CUDA 8 2016, *NVIDIA* https://devblogs.nvidia.com/parallelforall/cuda-8-features-revealed/.

[12] Phil Rogers and AC Fellow. Heterogeneous system architecture overview. *In Hot Chips*, volume 25, 2013.

[13] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Architectural support for address translation on GPUs: designing memory management units for CPU/GPUs with unified address spaces. *ACM SIGPLAN Notices*, 49(4):743?758, 2014.

[14] 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

[15] Alfred V.Aho, Ravi Sethi and Jeffrey D.Ullman *"Compilers: Principles, Techniques, and Tools", Addison-Wesley 1988*

[16] 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 – ICPADS '13*. pages 275–282. IEEE, 2013.

[17] 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.

[18] 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. *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 494-506. IEEE, 2016.

[19] Kim, Junghyun and Lee, Yong-Jun and Park, Jungho and Lee, Jaejin Translating device constructs to OpenCL using unnecessary data transfer elimination. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE Press, 2016.

[20] Pai, Sreepathi and Govindarajan, R and Thazhuthaveetil, Matthew J. Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 33–42, ACM, 2012.

[21] 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), 2015 International Conference on*, pages 1–8. IEEE, 2015.

[22] Isaac Gelado, John E Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wenmei 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.

[23] Marc Jorda, Ivan Tanasic, Javier Cabezas, Lluis Vilanova, Isaac Gelado, and Nacho Navarro. Auto-tuning of data communication on heterogeneous systems. In *Embedded Multicore Socs (MCSoC), 2013 IEEE 7th International Symposium* on, pages 135-140. IEEE, 2013.

[24] 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.

[25] Sylvain Henry, Alexandre Denis, Denis Barthou, Marie-Christine Counilh, and Raymond Namyst. Toward opencl automatic multi-device support. In *Euro-Par 2014 Parallel Processing*, pages 776–787. Springer, 2014.

[26] 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.

[27] Thomas B Jablin, Prakash Prabhu, James A Jablin, Nick P Johnson, Stephen R Beard, and David I August. Automatic cpu-gpu communication management and optimization. In *ACM SIGPLAN Notices*, volume 46, pages 142–151. ACM, 2011.

[28] 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.

[29] Bastoul, C. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques – PACT '04*. IEEE Computer Society, 2004

[30] Verdoolaege, S. isl: An integer set library for the polyhedral model. In *K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, editors, Mathematical Software - ICMS 2010, volume 6327 of Lecture Notes in Computer Science*. Springer, 2010

[31] 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.

[32] 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.

[33] 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.

[34] Rubasri Kalidas, Mayank Daga, Konstantinos Krommydas, and Wu-chun Feng. On the performance, energy, and power of data-access methods in heterogeneous computing systems. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 871?879. IEEE, 2015.

[35] Krishnahari Thouti and SR Sathe. A methodology for translating c-programs to opencl. *International Journal of Computer Applications*, 82(3), 2013.

[36] 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.

[37] Daga, Mayank and Aji, Ashwin M and Feng, Wu-chun. On the efficacy of a fused CPU+ GPU processor (or APU) for parallel computing. In *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pages 141–149, IEEE, 2011.

[38] Ryoo, Shane and Rodrigues, Christopher I and Baghsorkhi, Sara S and Stone, Sam S and Kirk, David B and Hwu, Wen-mei W. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM, 2008.

[39] Terboven, Christian and Mey, Dieter and Sarholz, Samuel. OpenMP on Multicore Architectures. In *Proceedings of the 3rd International Workshop on OpenMP: A Practical Programming Model for the Multi-Core Era – IWOMP'07*, pages 54–64. Springer-Verlag, 2008.

[40] Becchi, Michela and Byna, Surendra and Cadambi, Srihari and Chakradhar, Srimat. Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures.*, pages 82–91. ACM, 2010.

[41] Grewe, Dominik and Wang, Zheng and O'Boyle, Michael FP. Portable mapping of data parallel programs to OpenCL for heterogeneous systems In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, IEEE, 2013.

[42] Ferrer, Roger and Planas, Judit and Bellens, Pieter and Duran, Alejandro and Gonzalez, Marc and Martorell, Xavier and Badia, Rosa M and Ayguade, Eduard and Labarta, Jesus. Optimizing the exploitation of multicore processors and GPUs with OpenMP and OpenCL. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 215–229, Springer, 2010.