OptCL: A Middleware to Optimise Performance for High Performance Domain-Specific Languages on Heterogeneous Platforms

Jiajian Xiao[1,2(✉)], Philipp Andelfinger[4], Wentong Cai[3], David Eckhoff[1,2], and Alois Knoll[2,3]

[1] TUM CREATE, Singapore, Singapore
{jiajian.xiao,david.eckhoff}@tum-create.edu.sg
[2] Technische Universität München, Munich, Germany
knoll@in.tum.de
[3] Nanyang Technological University, Singapore, Singapore
aswtcai@ntu.edu.sg
[4] University of Rostock, Rostock, Germany
philipp.andelfinger@uni-rostock.de

Abstract. Programming on heterogeneous hardware architectures using OpenCL requires thorough knowledge of the hardware. Many High-Performance Domain-Specific Languages (HPDSLs) are aimed at simplifying the programming efforts by abstracting away hardware details, allowing users to program in a sequential style. However, most HPDSLs still require the users to manually map compute workloads to the best suitable hardware to achieve optimal performance. This again calls for knowledge of the underlying hardware and trial-and-error attempts. Further, very often they only consider an offloading mode where compute-intensive tasks are offloaded to accelerators. During this offloading period, CPUs remain idle, leaving parts of the available computational power untapped. In this work, we propose a tool named OptCL for existing HPDSLs to enable a heterogeneous co-execution mode when capable where CPUs and accelerators can process data simultaneously. Through a static analysis of data dependencies among compute-intensive code regions and performance predictions, the tool selects the best execution schemes out of purely CPU/accelerator execution or co-execution. We show that by enabling co-execution on dedicated and integrated CPU-GPU systems up to 13× and 21× speed-ups can be achieved.

Keywords: Heterogeneous hardware · OpenCL · Domain-Specific Language · CPU · GPU

This work was financially supported by the Singapore National Research Foundation under its Campus for Research Excellence and Technological Enterprise (CREATE) programme.
Financial support was provided by the Deutsche Forschungsgemeinschaft (DFG) research grant UH-66/15-1 (MoSiLLDe).

Y. Lai et al. (Eds.): ICA3PP 2021, LNCS 13157, pp. 772–791, 2022.
https://doi.org/10.1007/978-3-030-95391-1_48

1 Introduction

Today, hardware environments have become increasingly parallel or heterogeneous to meet growing computational demands. A personal computer nowadays is commonly equipped with a multi-core CPU with an integrated or a discrete GPU. Some cloud service providers also offer instances equipped with Field Programmable Gate Arrays (FPGAs). Programming such accelerators used to be cumbersome, as they required in-depth knowledge of the hardware, e.g., mapping computing task to pixel shaders for GPUs or to logic gates for FPGAs. The emergence of programming languages such as the Open Computing Language (OpenCL) vastly simplifies the programming efforts by abstracting away most of the hardware details. It enables programming data parallel code targeting various hardware platforms using a C-like standard. However, challenges still remain as developers need to learn yet another programming language and port sequential legacy code to a parallel OpenCL programming style.

A number of High-Performance Domain-Specific Languages (HPDSLs) such as SYCL, OpenABL [5,30], or Habanero-C [11] further simplify the adoption of OpenCL. They act as an intermediate layer between OpenCL and common programming languages such as C, allowing users to program in their familiar sequential style. Users are only required to annotate the compute-intensive functions (using pragma *STEP* in OpenABL) or putting the compute-intensive parts in a special function (*submit* in SYCL or *launch* in Habanero-C). HPDSL-specific compilers are finally responsible for translating those functions to OpenCL kernels.

Although these HPDSLs simplify the parallel programming, they still require the user to decide which parts should be offloaded to accelerators. The selection of tasks for offloading to accelerators is not trivial (e.g., [31]), since thorough knowledge of the hardware is again required to optimally map workloads to accelerators. This limits the benefits of HPDSLs. Frameworks such as Polly-ACC [7] are proposed to relieve the users of this burden by automatically detecting compute-intensive hotspots and translating them to OpenCL kernels. However, these frameworks typically follow a so-called offloading mode, where the hotspots are offloaded to accelerators and CPUs remain idle during the offloading periods, leaving their computational power temporarily untapped.

In this paper, we propose a middleware called OptCL (**O**ptimise **p**erformance **t**argeting high-performance domain-specifi**C** **L**anguages) for existing HPDSLs. It combines a series of traditional or well-known approaches in the field of data-dependency analysis and performance profiling, customising them to the features of HPDSL code. In addition to the offloading mode, which has already been achieved in many original HPDSL compilers or with the help of other frameworks, the middleware enables a co-execution mode in which CPU and accelerators work simultaneously if a performance benefit is expected, which is determined through data dependency analysis and performance predictions on the available hardware. OptCL focuses on data dependencies crossing High Performance Regions (HPRs), defined as the code regions compiled to OpenCL kernels, tailored to the structure of HPDSLs. Further, guided by the same data

dependency analysis, OptCL also reduces kernel invocation overheads. OptCL is based on Clang, operating on an Intermediate Representation (IR) level called Abstract Syntax Tree (AST). It can be seamlessly plugged into a wide range of C-based HPDSLs with little installation effort. OptCL complements an original HPDSL compiler, enabling it to also work for closed-source HPDSLs, provided that the HPDSL can output IR of OpenCL kernels in the shape of Standard Portable Intermediate Representation (SPIR), a binary OpenCL IR for CPUs and AMD GPUs or Parallel Thread Execution (PTX) for NVIDIA GPUs. The main contributions of this paper are:

- To the best of our knowledge, we are the first to propose a tool for existing HPDSLs to generate OpenCL code that enables co-execution.
- A comparison study between a sampling-based and a machine-learning based approach to estimate the performance of OpenCL kernels on hardware.
- We present two case studies of applying OptCL to two HPDSLs: OpenABL, an open-source Domain-Specific Language (DSL) for Agent-Based Simulations (ABSs) in which we have full control of the code generation workflow; and ComputeCpp, a closed-source commercial implementation of SYCL where we have less control of the code generation workflow.

The remainder of this paper is organised as follow: In Sect. 2, we present background and an overview of related work. In Sect. 3, we describe the OptCL middleware in detail. We present our case studies and evaluate the performance of OptCL in Sect. 4. Section 5 concludes the paper.

2 Background and Related Work

2.1 SYCL

SYCL is a specification developed by Khronos targeting heterogeneous hardware platforms. It allows the same code to be executed on accelerators such as GPUs, FPGAs as well as CPUs. SYCL abstracts away from hardware details, enabling developers to code parallel programs in a regular C++ style. Most SYCL implementations generate OpenCL code.

Algorithm 1 shows the example of summing up two vectors using SYCL. Users are required to first select a device to run the kernels on (L. 1–2), followed by allocating memory space on the device-side (L. 3–5). L. 6–12 is a special function marked by the keyword *submit* defining a kernel in the form of a lambda expression by first declaring the inputs and outputs using a data type called *accessor*. L. 10–11 implements the logic of a kernel. Many SYCL implementations also support heterogeneous platforms by providing facilities such as asynchronous scheduling of kernels by overlapping data transfer and computation.

ComputeCpp is a commercial implementation of SYCL (a free community edition is also available) [4]. Following the SYCL specification, ComputeCpp consists of two parts: a device compiler and a run-time library, both of which are closed source. Only AMD GPUs and x86/ARM CPUs are fully supported.

Experimental support for NVIDIA GPUs was removed in the most recent version. Given SYCL code as input, ComputeCpp outputs SPIR or PTX as well as an integration file to be loaded by the ComputeCpp run-time. During compilation, users choose which IR (SPIR or PTX) to output. Therefore, NVIDIA GPUs cannot be used together with other accelerators in a heterogeneous setting.

Although ComputeCpp supports offloading to multiple devices, the assignment of individual kernels to the hardware is left to the user. Further, as ComputeCpp focuses more on portability than performance, it may not always fully exploit the hardware's capabilities [29]. We will show that by plugging OptCL into ComputeCpp, the performance of the generated OpenCL code can indeed be improved, both through co-execution and by reducing the kernel invocation overhead. OptCL constructs OpenCL programs that support a mixed use of PTX and SPIR binaries. As a side effect, we also re-enable using NVIDIA GPUs with ComputeCpp, and more importantly enable the use of NVIDIA GPUs along with other accelerators unsupported by the original ComputeCpp.

Algorithm 1 SYCL code

```
gpu_selector device_selector;
queue deviceQueue(device_selector);
buffer<float, 1> dev_a(a, range<1>(N));
buffer<float, 1> dev_b(b, range<1>(N));
buffer<float, 1> dev_c(c, range<1>(N));
deviceQueue.submit([&] (handler& ch) {
  accessor a = dev_a.get_access<mode::read>(ch);
  accessor b = dev_b.get_access<mode::read>(ch);
  accessor c = dev_c.get_access<mode::write>(ch);
  auto kernel = [=](id<1> wid) {
    c[wid] = a[wid] + b[wid]; }
});
```

Algorithm 2 OpenABL code

```
agent Car {
  float velocity;
  position float2 pos;}
param int num_of_agents = 1000;
param int sim_steps = 100;
Lane { int laneId; float length; }
environment {env: Lane roads[4096]}
step car_follow(Car in -> out) {...}
step lane_change(Car in -> out) {...}
void main() {
simulate(sim_steps){car_follow,
lane_change} }
```

2.2 OpenABL

OpenABL is an open-source DSL to generate high-performance ABS programs from sequential code written in a C-like language for various hardware platforms [5,30]. Unlike SYCL, which is designed for general computing, OpenABL specialises in the field of ABS. The OpenABL framework is comprised of two parts: a *frontend* and a *backend*. Algorithm 2 illustrates a skeleton to implement a simple traffic simulation using OpenABL. Users may define agents with a mandatory position member (keyword `agent`, L.1-3), constants (keyword `param`, L. 4–5), simulation environments with user-specific type (keyword `environment`, L. 6–7), step functions (marked by pragma `step`, L. 8–9) which will be later translated into OpenCL kernels, and a main function (keyword `main()`, L. 10–12).

The OpenABL compiler first generates an AST from the above code, the same IR used in our middleware. The backend later rebuilds simulation code from the AST and parallelises step functions. So far, five backends are supported by OpenABL, including OpenCL. The workflow of OpenABL overlaps partially with our proposed middleware. The middleware can thus be completely integrated into the OpenABL compilation flow to generate high-performance OpenCL code.

2.3 Related Work

The presented middleware targets heterogeneous hardware platforms. Challenges and state-of-the-art solutions in this context can be found in [12] for general-purpose computing and in [32] for agent-based simulations.

Previous efforts parallelise sequential code using a set of pre-defined rewrite rules [27], code templates [28] or special syntax for loops [2,14], enabling the translation to programs in OpenCL, CUDA, or Threading Building Blocks code. These frameworks pursue a goal of generating parallel programs from sequential representations similar to HPDSLs. Necessary structural amendments have to be made for existing HPDSL programs to use these frameworks, while our OptCL aim to parallelise existing programs without any changes required.

Several works [7,24,26] automatically detect parallelisable loops in sequential representations and translate the loops to OpenCL kernels. The existing approaches targeting OpenCL follow a similar workflow as OptCL: a sequential program is first converted into an IR. Data parallel loops or static control regions are detected in the IR and translated to OpenCL kernels. These existing works assume that parallelisable regions are explicitly stated as loops in the sequential code. However, this assumption may not hold for HPDSL programs. As HPDSLs abstract away implementation details, they usually only require users to code loop bodies, i.e., the HPRs. The iterative behaviour is handled internally by the HPDSL runtime (and eventually by the OpenCL runtime). Further, none of these approaches consider co-execution opportunities. In contrast to the existing works on automatic parallelisation, our approach relies on HPDSL-level code segments to identify regions for co-execution. The combination of existing automatic parallelisation methods with our approach for mapping computations to heterogeneous hardware is an interesting avenue for future work.

There exist a handful of DSLs providing different levels of native co-execution support. Habanero-C (HC) [11] features shared virtual memory and smart data layout to achieve performance portability on CPU-GPU systems. CnC-HC [25], which maps the Concurrent Collections (CnC) model to the HC runtime, extends the supported hardware to include FPGAs. HC and CnC-HC both use work-stealing approaches to achieve load-balancing between CPU and accelerators. Unlike our work, which automatically determines data dependencies, in HC, the data dependencies are ensured by the users based on *async* and *finish* constructs. Performance on the individual piece of hardware is also estimated based on a user-specified machine description. The work reported in [20] extends PetaBricks language which allows users to specify multiple algorithmic paths for the same input and output. The compiler then chooses the path leading to the best performance given hardware settings determined using an evolutionary mechanism. However, users still need to produce parallel code explicitly.

Two co-execution schemes are extensively used in the literature: data partitioning and task partitioning. Tasks partitioning has been carried out offline using performance analytical models [3] or using machine learning-based approaches [6,10,13,17]. OpenABLext [30], an extension of the OpenABL framework, carries out an online partitioning of the workload. It executes the first few

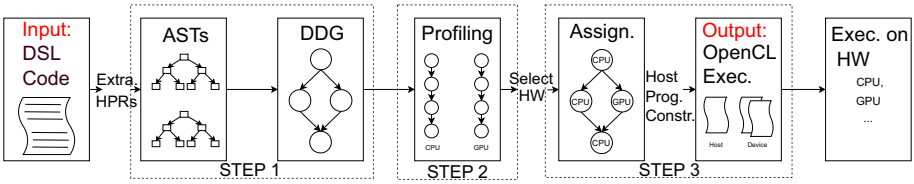


Fig. 1. An overview of OptCL.

simulation steps to profile the available hardware to determine the hardware assignment for the remainder of the simulation. This approach is closely tailored to the OpenABLext workflow and requires additional user input. In the evaluation section, we will conduct a comparison study between a representative offline machine learning-based approach and a sampling-based approach similar to [30] but applicable beyond the ABS domain. Other works [14,18,21] proposed novel adaptive scheduling mechanisms to partition the workload at the data level aiming at achieving load-balancing or power-saving. Many frameworks built based on these designs [8,16,19] can operate directly on OpenCL kernels. They typically start with a small portion of workload assigned to CPUs and the remainder to the accelerators (or vice versa). A balancing phase then incrementally balances the workload assigned to the CPUs and to the accelerators until convergence is reached. If the workload is irregular, this balancing phase can be re-triggered if certain imbalance criteria are met. Data layout and transfer among different devices are also dealt with by the frameworks automatically.

A study comparing data partitioning and task partitioning schemes on a CPU-FPGA system is carried out in [9]. The authors concluded that both schemes can be beneficial. OptCL partitions the workload at the task level, which is a natural choice following the specification of HPDSLs, as each HPR will be typically translated to one OpenCL kernel. Data partitioning will only be used in a special case as an additional optimisation (cf. Sect. 3.4). Future work could explore the combination of task and data partitioning in OptCL by employing one of the aforementioned designs.

3 The OptCL Middleware

An overview of OptCL is given in Fig. 1. OptCL generates OpenCL code in three steps: 1) data dependency analysis, 2) profiling, 3) hardware assignment, and code reconstruction. In the first step, OptCL identifies the sub-AST containing HPRs from the AST generated for the entire program. The sub-AST is further split up into smaller ASTs, each representing an HPR or the code region between two HPRs. A Data Dependency Graph (DDG) derived from those small ASTs identifies the data dependencies and distinguishes HPRs that are free of interdependencies, which can thus be co-executed. The second step profiles the kernels generated out of the HPRs on the available hardware. Based on the profiling

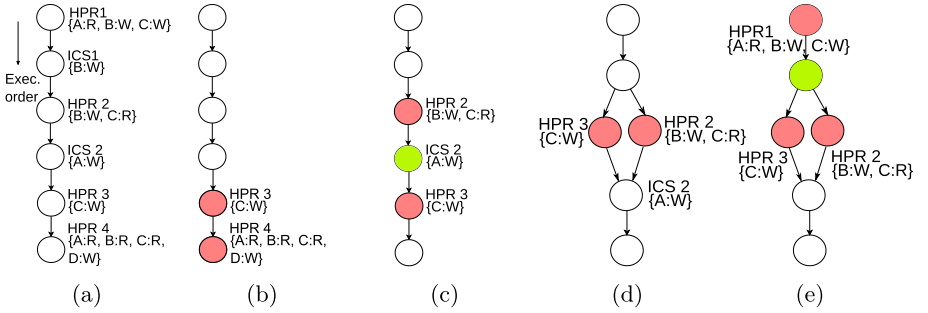


Fig. 2. DAG generation. (a) Original DDG (b) Processing HPR3&4 (c) Proc. HPR2&3 w. ICS2 (d) Parallelising HPR2&3 (e) Proc. HPR 1, 2&3 w. ICS1

results, the third step decides on the execution scheme and reconstructs the program accordingly.

Users are required to specify two inputs to OptCL at installation time: the keyword used to annotate the start of HPRs (HPR keyword), and the data type used to define in- and output (in other words, to allocate memory on the devices) of those HPRs (e.g., *accessor* in SYCL or *agent* in OpenABL), which is referred to as Device Variable Keyword (DVK). Notably, this setup is done once per language and is application-agnostic. OptCL can run in a fully automated way after this setup. In what follows, we will discuss each step in detail.

Algorithm 3 SYCL code

```
1:  //d-dev vars, h-host counterpart of dev vars
2:  HPR1(global_id i) {
3:    B_d[i] = A_d[i] + 1;
4:    C_d[i] = A_d[i] * 5;
5:  }// A:Re, B:Wr, C:Wr
6:  for (each element B_h[i] in B_h) {
7:    B_h[i]++;
8:  } //ICS1 B:Wr
9:  HPR2(global_id i) {
10:   B_d[i] = B_d[i] + C_d[i];
11: }//B:Wr, C:Re
12: for (each element A_h[i] in A_h) {
13:   A_h[i]++;
14: } //ICS2 A:Wr
15: HPR3(global_id i) {
16:   C_d[i]++;
17: }//C:Wr
18: HPR4(global_id i) {
19:   D_d[i] = A_d[i] + B_d[i] + C_d[i];
20: }//A:Re, B:Re, C:Re, D:Wr
```

3.1 Step 1: Data Dependency Analysis

Common compilers also produce ASTs during compilation, from which data dependencies for the entire code base are deduced. OptCL only focuses on a portion of the entire AST i.e. the sub-AST around the HPRs and eliminates code snippets that are not targeted for parallel execution, e.g., reading inputs, etc. This can reduce evaluation complexity, tailored to the structure of HPDSL code.

Step 1 starts from a partial compilation of the input HPDSL code. This assembles code if multiple source files are available and sorts HPRs according to

the required execution order. The partial compilation stops when the AST for the entire program is generated. At a glance, it seems to be redundant with the HPDSL's own compilation process. However, it is essential to enable OptCL on closed-source HPDSLs, since OptCL does not have access to the intermediate compiling stages of a closed-source compiler.

The AST of the entire program is first traversed to locate HPRs and usage of device variables. The traversal is implemented using the *RecursiveASTVisitor* class of Clang. HPRs are identified in the raw code using the HPR keyword inputted by the users. The scope of each HPR is to the end of the function (e.g., for OpenABL) or lambda expression (e.g., for SYCL) following the keyword. Device variables are identified by the DVK for both their device part (variables which are mapped to the device's memory space), their host counterpart (variables declared on the host to initialise device variables or to store the results read from the device), and their aliases determined by the clang alias analysis. There can be code snippets between two consequent HPRs where the host counterpart of a device variable is amended. Data dependency can thus also occur, preventing the neighbouring HPRs from parallel execution. Therefore, during data dependency analysis, those code snippets, further referred to as In-between Code Snippets (ICSs), should also be taken into consideration.

While traversing the AST of the entire program, it can be identified whether an HPR or an ICS is within a loop. This *IsWithinALoop* information is also recorded to be applied later. With all HPRs detected, unit ASTs (uASTs), defined as sub-ASTs representing an HPR or an ICS, are extracted from the overall AST.

Within one uAST, we trace read and write operations on the device variables based on three patterns and combinations thereof that may indicate a read or write operation:

- Assignment statement ($A = B$ or $A = \text{func}(B, C)$): the former case entails a write operation on A and a read operation on B. The latter entails a write operation on A. As we do not analyse the function func(), we conservatively assume that func() may cause both read and write on B and C.
- Binary operation ($A = B + C$): a binary operation incurs a write operation on A and read operations on B and C respectively.
- Unary operation ($A{+}{+}$ or $!A$): for unary operations, we distinguish between increment ($A{+}{+}$ or ${+}{+}A$) and decrement ($A{-}{-}$ or ${-}{-}A$) on the one hand, which incur both read and write operations on A; and other unary operations which incur only read operations on A on the other hand.

Algorithm 3 illustrates an example HPDSL program with four HPRs and two ICSs. HPRs or ICSs within a loop (identified using *IsWithinALoop*) are treated as a single node in DDG, as the data dependency remains unchanged in each iteration. By applying the above rules, we can identify which type of operations is imposed in a HPR/ICS for each device variable (e.g. HPR1 reads from device variable A and writes B, so $A : Re$ and $B : Wr$ with Wr always overrides Re). The generation of DDG starts with assuming dependencies everywhere, resulting in a DDG representing sequential execution (Fig. 2a). OptCL then tries to parallelise as many

HPR nodes as possible. ICS nodes are not considered for parallelisation, as they by user's design have to be executed sequentially on the host. However, as explained above, they play an essential role to determine the data dependencies of the surrounding HPRs. The attempt begins with the last HPR nodes (e.g., HPR3&4 in Fig. 2a). Two consecutive HPR nodes can touch the same device variable, for example, HPR1 and HPR2 touch variable A, B and C, yielding four types of dependencies: Read-After-Read (RAR), Write-After-Read (WAR), Read-After-Write (RAW) and Write-After-Write (WAW).

Two or more consecutive HPR nodes can be parallelised if: 1) There is no ICS node in between carrying write dependencies of the device variables that are used in the latter node in the DDG graph; *and* 2) They touch a disjunct set of device variables *or* all device variables they touch have either RAR or WAR dependency.

While RAR intuitively causes no dependency, WAR also results in no dependency. This is because when executing in parallel on different devices, each device keeps a local copy of the variable. Modifying the local copy on one device has no effect on other devices. For instance, as shown in Algorithm 3, when co-executing HPR2 and HPR3 on different devices, HPR2 and HPR3 each keep a local copy of C which reflects the values after HPR1. The write operations on C (e.g., $C[i]$++) in HPR3 do not change the values of C in HPR2.

As illustrated in Fig. 2b, HPR4 cannot be parallelised with HPR3 due to RAW dependency on C (HPR3 writes to C and then HPR4 reads from C). Node HPR3 can be parallelised with node HPR2 (Fig. 2c and 2d), because only WAR dependency exists (HPR2 reads from C and HPR3 writes to C) and ICS2 carrying write-dependency to A while A is not used in the latter node i.e. node HPR3. Node HPR1 cannot be parallelised with node HPR2 and node HPR3 (Fig. 2e) owing to two reasons. First, there is RAW dependency on device variable B and C. Second, ICS1 modifies variable B which is overwritten in HPR2.

This parallelisation process traverses a DDG iteratively until no more nodes can be parallelised.

3.2 Step 2: Profiling

We propose two design options for the profiling stage: a sampling-based profiling approach executing a small portion of the application to estimate the performance of the whole program and an offline mechanism using a machine-learning based approach to predict the performance. The usability and accuracy of two different approaches will be compared in Sect. 4.

Sampling-Based Profiling. The sampling-based profiling mechanism extends the mechanism introduced in an earlier work, OpenABLext [30], to generalize from the ABS case to other applications. Device programs encapsulating OpenCL kernels (e.g. PTX or SPIR) are generated using the HPDSL's own compiler. OptCL produces one temporary host program per device (including CPUs), assuming a sequential execution order and a single device environment. These temporary host programs are functionally similar to the ones generated by the original HPDSL

compiler with extra utilities to measure the execution time of individual kernels, including both kernel invocation and data transfers (for using CPUs as accelerators, only kernel invocation time is counted, as there is no data transfer). Further, after each kernel invocation, the data is transferred back to the host in order to measure the data transfer overhead.

Each kernel is profiled with the real data with a timer or the time used for one full kernel invocation, whichever takes longer. In the case multiple rounds of kernel invocations can be done within the time limit set by the timer, the throughput is recorded. If one kernel invocation takes longer than the timer, the execution time is used to indicate the performance.

Offline Profiling. The offline approach implements an established performance prediction model using so-called architecture-independent features introduced in work [10]. Each OpenCL kernel is abstracted as a series of architecture-independent features ranging from opcode counts to branch deviation entropies. The Architecture Independent Workload Characterization (AIWC) tool [10] developed based on the idea is employed to characterise the OpenCL kernels generated by the HPDSL's compiler. The AIWC tool acts as a plugin to an OpenCL simulator, Oclgrind, which has been widely used to debug OpenCL kernels [22]. Oclgrind simulates the execution of OpenCL kernels on the IR level, and therefore it is hardware independent. During the simulation, Oclgrind generates events, e.g., on a conditional branch, based on the encountered IR instructions. AIWC acts as an event handler which counts the appearance of certain events. When a full kernel invocation is completed, AIWC conducts a statistical summary of the counters and produces metrics. These features are later fed into a prediction model based on a Random Forest (RF) [1] to predict the execution time on a CPU, a GPU or any other accelerator. By iteratively partitioning the data, the algorithm builds decision trees. Then, the forest, which is an ensemble of decision trees, provides the prediction based on the mean among the trees, providing a more reliable and robust prediction result. The experiments conducted by the authors in [10] showed an average of 1.2% deviation between the predicted execution time and measured time.

3.3 Step 3: Hardware Assignment and Program Reconstruction

In this step, a hardware assignment to maximise the performance is determined based on the profiling stage, following these rules:

a) If the throughput/execution time of a kernel on one device outperforms other devices, the kernel is assigned to this winning device.
b) When co-execution is possible as indicated by the DDG: 1) If the co-executed kernels have different winning devices, they are assigned to their respective winning devices. 2) If some of them share a winning device 1, the second best device 2 of a kernel is chosen, as long as the total execution time of these kernels on device 1 is larger than co-execution on device 1 and 2 (cf. Fig. 3).

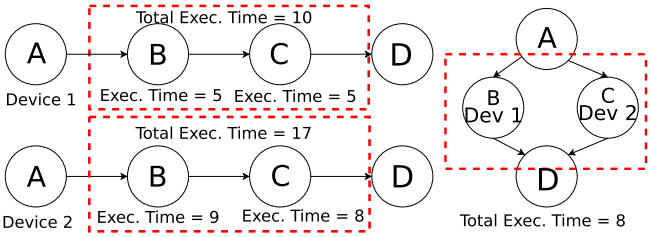


Fig. 3. Co-exec. leads to a gain even if sub-optimal hardware is chosen.

Based on the hardware assignment, OptCL reconstructs the HPDSL program. OptCL employs the Clang rewrite method to replace the HPRs with their OpenCL kernel invocations, together with the necessary initialisation and data transfer. Other parts such as ICSs, I/Os are copied over from the original HPDSL program. During the reconstruction, a couple of measures are taken to reduce the kernel invocation overhead: firstly, all OpenCL kernels are compiled only once prior to the start of the first HPR. The compiled binaries stay in memory and are used when needed. Secondly, we optimise for the situation where HPRs reside in a loop. This can cause data transfer redundancies if every single call to the HPR in a loop iteration is treated as a new OpenCL kernel, which entails bi-directional data transfer to/from the host. The *IsWithinALoop* information is collected in Step 1. In case all HPRs residing in the same loop are assigned to the same device and there is no ICS in the loop, the data transfer between host and device is extracted and executed outside the loop to eliminate redundant data transfer.

In a multi-device execution environment, different devices usually do not share the same memory space. OptCL also inserts code that is responsible for allocating memory on the respective devices. In the case where a shift of devices is required between two consecutive kernels, a data path is built in between. There are special cases where data exchange can be avoided, e.g., when an Accelerated Processing Unit (APU) is used. We will showcase this in the evaluation section. OptCL smartly decides which data should be copied over to the other devices and perform the copies only when it is necessary based on the data dependency information collected in Step 1 as well as the hardware type.

After the co-execution, the host may receive inconsistent outputs from different devices. The correct output is then restored using the dependency information recorded in Step 1. A *merge_function* is inserted into the host program in two scenarios: In a WAR dependency scenario, the *merge_function* picks the output from the device conducting the write operation. In the scenario where co-executed kernels write to different device variables, the *merge_function* gathers the individual outputs from the devices that carried out the write operation and merges them together on the host. For example, HPR2 and HPR3 illustrated in Algorithm 3 can be co-executed on different devices. The *merge_function* then merges device variable B outputted by HPR2 and device variable C by HPR3 on the host. The merged data is re-distributed to the devices if the next kernel is not executed on the host (CPUs).

A piece of clean-up code concludes the host program, freeing buffers and kernels as well as outputting the results if required. Eventually, OptCL lets Clang compile the reconstructed code and generate the final executables.

3.4 Optimisation

Enhanced Dependency Detection. Device variables can be declared using user-defined structures. For example, when using OpenABL to program a traffic ABS, a device variable can be an array of car agents where each car possesses its identifier, position, velocity, etc. Dependencies may be overestimated if OptCL were to treat the structure as a whole. In a given program, it may occur that two consecutive kernels write to disjoint sets of members of the same structure. In this case, these two kernels can potentially still be co-executed. However, given the dependency detection rules described in Sect. 3.1, they would be identified as having a WAW dependency.

To solve this issue, an enhanced dependency detection is introduced. Device variables declared as structures are broken down to the member level. Given device variable A defined using structure $T\{type\ member1, type\ member2, ...\}$, the dependency detection rules described in Sect. 3.1 traces the read and write operations on $A.member1$, $A.member2$, etc.

User-Specified *Merge_Function*. If extra logic is provided to resolve dependency conflicts, parallelisation of HPRs with RAW or WAW dependencies is also possible. In some use cases, RAW or WAW dependency may even be tolerated, e.g., in stochastic ABSs [23].

To fulfil such needs, we allow users to define their own *merge_functions* following the naming convention *kernel1_kernel2_merge_function* in the respective HPDSL's syntax. Once a RAW or WAW dependencies are detected, OptCL will search for the existence of an optional *merge_function*. Users can also define an empty *merge_function* to allow RAW or WAW dependency to exist. This also implies that OptCL will not check if the dependency conflicts are resolved by applying the user-specified *merge_functions*. Users are then responsible for ensuring the logic of the program is still correct by providing the *merge_functions*. An example of a user-specified *merge_function* will be given in Sect. 4.

Single Kernel. When an HPDSL program contains only a single HPR, co-execution is still possible if the data partitioning scheme is used. It is safe to do so because the input data is processed in parallel even on one device. Each device receives a subset of the data proportional to its computational power as profiled in Step 2. After processing, the output is transferred back to the host for merging.

To prevent discrepancy to the outcome of the HPDSL programs, this optimisation only applies to single kernels possessing no intra-read-and-write dependency. That means, e.g., if the kernel touches device variable A[i], there has to be no write to A[j] where $i \neq j$ in the same kernel. This is because A[i] and A[j] can

be potentially processed on different devices where there is no guaranteed synchronisation. This rule can be imposed by doing an additional intra-dependency check during the data dependency analysis in Step 1.

4 Evaluation

We evaluate OptCL by plugging it into two HPDSLs: SYCL and OpenABL. Both of the studied HPDSLs target CPU-GPU heterogeneous platforms. Our evaluation was, therefore, conducted on CPU-GPU systems. To cover possible hardware configurations, we include two types of CPU-GPU systems: a dedicated CPU-GPU (dCPU-GPU) platform equipped with an Intel Core i5-11400F CPU with 16 GB of RAM and an NVIDIA GTX 1070 graphics card with 8 GB of RAM and an integrated CPU-GPU (iCPU-GPU) platform equipped with an Intel i5-7400 CPU with 16 GB of RAM and an integrated Intel HD 630 iGPU. Both systems run Ubuntu 18.04. The key difference between the two platforms is that while data transfer is often required between the CPU and the GPU in the dCPU-GPU setting, physical memory is shared between the CPU and the iGPU in the iCPU-GPU setting. Thus, the CPU and the iGPU can directly access each other's data, eliminating the data transfer overhead.

For SYCL, six applications covering domains ranging from physics simulation to machine learning were selected to evaluate the performance of OptCL. The applications are: **Backpropagation (BP)**, an algorithm used for training neural networks in supervised machine learning tasks; **K-Means (KM)**, a clustering algorithm that partitions N nodes into K clusters; **Speckle Reducing anisotropic diffusion (SR)**, a noise removal algorithm, e.g., for ultrasonic and radar images; **Hot Spot (HS)**, a simulation of a processor's thermal dynamics; **CirCle (CC)**, a molecular simulation of the potential and relocation of molecules driven by mutual forces in a 2-D space; **hierarchical-Matrix-Vector multiplication (MV)** [15], a method that decomposes a large matrix-vector multiplication and computes the result iteratively.

The tested applications all follow a pattern where HPRs are executed iteratively. We were not aware of any off-the-shelf SYCL implementations of these applications. Therefore, we implemented them from scratch, applying our best efforts to optimise for a general SIMD architecture such as those of GPUs.[1]

4.1 Profiling Approaches Comparison

First, we compare the sampling-based approach and the offline approach introduced in Sect. 3.2. The goal is to select the approach with higher accuracy while also evaluating the usability. The evaluation is done on the dCPU-GPU system.

No setup is needed for the sampling-based approach. For the offline approach, the RF must be pre-trained with an extensive number of kernel patterns. We trained the model using the OpenDwarfs Extended Benchmark Suite,[2] the

[1] https://github.com/xjjex1990/OptCL.
[2] https://github.com/BeauJoh/OpenDwarfs.

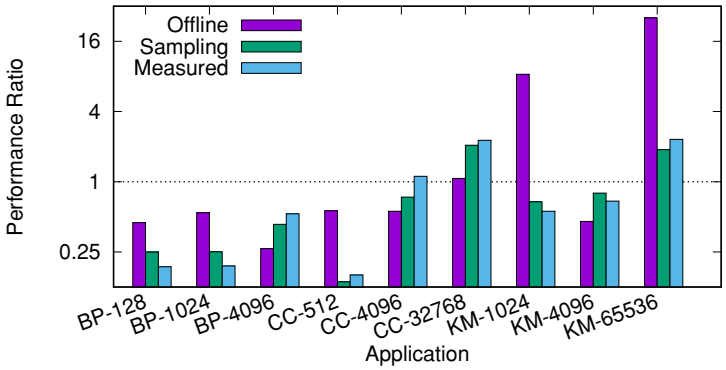


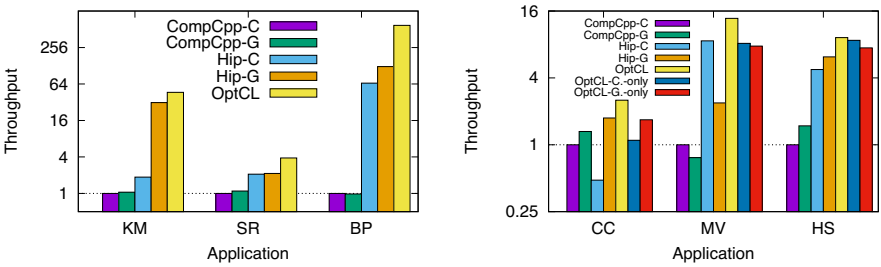
Fig. 4. Comparison of different $R_{C/G}$ results.

same suite used for training in the original AIWC work [10]. The suite covers applications ranging from solvers to mathematical problems to image processing algorithms, demonstrating versatile kernel patterns. By varying the input sizes, the suite provided ∼5,000 kernel patterns and their execution times.

The RF intakes three major parameters. *num.trees*: the number of trees in a forest; *mtry*: number of possible independent features; and *min.node.size*: the minimal node size per tree. We employed the values suggested by the AIWC authors: $num.trees = 505$, $mtry = 30$ and $min.node.size = 9$. With these parameters, the training process took less than 20 min on a 32-core workstation. For the sampling-based approach, we profile the applications with a timer set to 1 s which is the only overhead of this approach.

The profiling step is to guide the hardware assignment. Therefore, it is more important to learn the relative performance comparison between different hardware types rather than the individual execution time. Further, for the sake of creating an application-independent metric to quantify the two profiling approaches, we employ a *performance ratio* metric $R_{C/G}$ defined as the execution time on the CPU divided by the time on the GPU. In case throughput is taken, $R_{C/G}$ is defined as the throughput on the GPU divided by the one on the CPU.

Figure 4 illustrates $R_{C/G}$ outputted by the sampling-based approach, the offline approach as well as the measured $R_{C/G}$ on the CPU and the GPU. The x-axis labels are of the form *NAME-SCALE*, where *NAME* refers to the name of the application and *SCALE* is the input size. Due to limit space, here we demonstrate three applications: BP, CC, and KM, but varying different input scales. As depict in Fig. 4, a base line $R_{C/G} = 1$ (meaning the performance on the CPU and the GPU is equal) splits the space into two sides. The sampling-based approach managed to identify correctly the better-performing hardware between the CPU and the GPU in all cases but one (CC-4096), albeit with a certain estimation error (defined as the estimated ratio divided by the measured ratio). It failed in the CC-4096 case because the amount of workload in each iteration of CC depends strongly on the changing positions of the molecules, and,



(a) Group 1 applications (b) Group 2 applications

Fig. 5. Throughput of SYCL applications on the dCPU-GPU system.

therefore, it changes from iteration to iteration. Using the first few iterations to estimate the full execution time thus leads to deviations. Although the offline approach also succeeded in estimating the performance deviation in most of the cases (7 out of 9), it came with much bigger estimation errors (82% on average versus 22% with the sampling-based approach). Significant errors are observed in KM-1024 as well as in KM-65536, as the training data lacked of such kernel pattern or input size.

In summary, the sampling-based approach led to better accuracy in all tested applications with zero setup effort, compared to a moderately trained (training data size/test data size = 555.6) offline machine-learning based approach. Hence, we will stick to use the sampling-based approach with timer set to 1 s in the rest of this paper. However, the offline approach could still produce valid results when trained to more comprehensively cover the possible kernel patterns and input sizes, which we defer to future work.

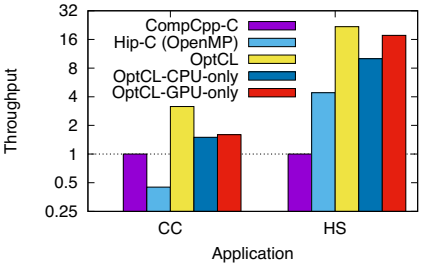


Fig. 6. Throughput of SYCL applications on the iCPU-GPU system.

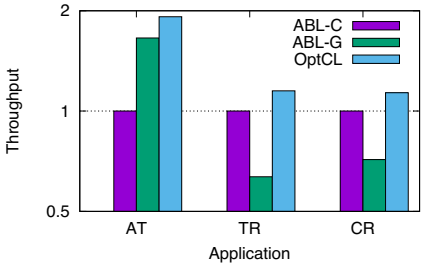


Fig. 7. Throughput of OpenABL applications.

4.2 Case Study 1: SYCL

We relied on ComputeCpp version 2.21, which still supports NVIDIA GPUs. All the tested applications were compiled with `-O3` optimisation.

Six applications are categorised into two groups based on the observed speed up types: **Group 1: BP, KM and SR.** These applications consist of several kernels. Owing to the dependencies between kernels, co-execution is not feasible. However, performance improvements are still possible through kernel invocation overhead reduction introduced in Sect. 3.3 as well as by executing them on the best suitable hardware. **Group 2: HS, CC and MV.** Co-execution is feasible. In addition to the performance benefits achieved by less invocation overhead, further performance improvements are observed due to co-execution. We report the performance of these two groups separately.

To put the performance of OptCL into perspective, we provide another baseline. The performance of the same SYCL code compiled by hipSYCL, an open-source SYCL implementation using OpenMP for CPU and CUDA for GPU as the backends. hipSYCL supports NVIDIA graphics cards (via clang-CUDA), and for CPUs, it uses OpenMP, which incurs only little invocation overhead.

Figure 5a and 5b show the throughput of CompCpp-C/G (the throughput of the SYCL code compiled by ComputeCpp and executes on CPU/GPU), Hip-C/G (the same code compiled by hipSYCL and runs on CPU/GPU) with each application normalised to the throughput of its respective CompCpp-C. As depicted in the two figures, OptCL achieved the best performance in all settings. The performance was not optimal for the ComputeCpp variants as shown in Fig. 5a, due to the incomplete support of NVIDIA GPUs and the kernel invocation overheads as the HPR in each iteration was treated as a new kernel. The runtime freshly compiles the kernel and redundantly transfers data to accelerators each time an HPR is invoked, which can be verified by running the executables in the Oclgrind simulation. By compiling the same SYCL code using hipSYCL, substantial speed-ups up to 120× were achieved. The performance was improved further by employing the OptCL middleware, thanks to the kernel invocation overhead reduction and using the most suitable hardware.

To better illustrate the power of co-execution, in Fig. 5b we also show the 'otherwise' scenarios, where we suppose OptCL would assign the kernels to only CPU (OptCL-C.-only) or GPU (OptCL-G.-only). Notably, Hip-C (OpenMP) performed worse than the ComputeCpp executables in the CC application. This is because in CC, each molecule traverses the global memory for other nearby molecules, causing large numbers of cache misses. As can be seen in the 'otherwise' scenarios, while running the OptCL variants on a single accelerator (OptCL-CPU-only and OptCL-GPU-only) already produced similar or even better performance than other variants, co-execution further boosted the throughputs. A maximum speed-up of 1.67× over the 'otherwise' scenarios and 13× over the baseline CompCpp-C was observed for the MV application.

The performance on the iCPU-GPU system is displayed in Fig. 6. As both ComputeCpp and hipSYCL do not well support iGPU (the support is experimental in hipSYCL), we exclude them from the evaluation. The iGPUs are usually not as powerful as the dedicated ones due to fewer cores and thermal concerns. As a consequence, co-execution for the MV application was not feasible in the iCPU-GPU setting, because of the large throughput deviation between the CPU and the iGPU, causing long execution time on the iGPU even with a

small amount of data. However, for the applications (CC and HS) that are eligible for co-execution, more significant performance benefits were obtained owing to the zero-copy technology reducing the data transfer overhead. We achieved a speed-up of 7× and 5× over the Hip-C executables and 3× and 21× over the CompCpp-C executables in CC and HS, respectively.

4.3 Case Study 2: OpenABL

Independent kernels are common in the domain of ABS where multiple models operate on different attributes of the same agent. Different types of agents in the same simulation can perform individual models that do not rely on the states of other types which presents ample opportunity for co-execution. However, not all of them can benefit from co-execution, as the performance on one type of hardware can dominate the others, resulting in the same situation as the MV case in the iCPU-GPU setting. Due to limited space, we only demonstrate applications that benefit from co-execution.

Three ABS models from three domains were selected: social science, transportation, and biology: **CRowd (CR)**, modelling the flocking behaviour of people following fire wardens to escape from a single-entrance room in case of a fire accident; **TRaffic (TR)**, a traffic simulation comprised of a car following model and a lane changing model. A user-specified *merge_function* is added to enable co-execution of these two models. For experimentation purposes, we set the *merge_function* such that the car following model always overwrites the output of the lane changing model; **AnTs (AT)**, a simulation of the foraging behaviour of ants. We based the evaluation on an extended version of OpenABL which supports generating OpenCL code.[3]

Despite the fact that data transfer overhead was avoided, co-execution was not feasible for the tested applications on the iCPU-GPU system. Similarly to the MV application, this is caused by the large performance deviation between the CPU and the iGPU. Therefore, in what follows, we only report the performance using the dCPU-GPU system. As shown in Fig. 7, compared to running on a single device, co-execution led to the best performance in all three applications. In the CR application, the path finding is slow on the GPU owing to the heavy memory operations, causing an overall performance reduction on the GPU as indicated by the ABL-G bar in Fig. 7. Similarly, for the TR application, the car following model requires memory-intensive search for nearby vehicle which is again slow on the GPU. By co-execution, i.e., assigning the memory-intensive kernel to the CPU and the rest to the GPU, a speed-up of 1.15× and 1.13× over running on the CPU was achieved by OptCL for the TR and CR application.

Although all kernels run faster on the GPU than the CPU in the AT applications due to GPU's massive parallelism, overlapping execution of kernels on the CPU and the GPU can still lead to performance benefits. This is because of the reason explained in Fig. 3. For the AT applications, co-execution was also 1.15× faster than running on the GPU.

[3] https://github.com/xjjex1990/OpenABL_Extension.

5 Conclusion and Future Work

In this paper, we presented OptCL, a middleware to generate efficient co-execution-enabled OpenCL code from existing high-performance Domain-Specific Languages (HPDSLs) code. Through a data dependencies analysis among High Performance Regions (HPRs) and performance predictions, OptCL assigns each kernel to the most suitable hardware device and selects the best execution strategy out of purely CPU-based execution, offloading to an accelerator, or co-execution. Kernel invocation and data transfer overheads are also minimised in the generated code.

The workflow of OptCL consists of three steps. Starting from HPDSLs code, OptCL triggers a partial compilation, where an Abstract Syntax Tree (AST) is generated. After identifying all High Performance Regions (HPRs), we can then only focus on the sub-AST that is related to HPRs. A data dependency graph is built using the information gathered from analysing the sub-AST, revealing the dependencies between HPRs and thus possibilities for co-execution. In Step 2, kernels converted from HPRs are profiled on the available hardware devices. The profiling results are used to assign each kernel to its best suitable hardware and to enable co-execution where possible. Two profiling approaches are studied to estimate the power of each device: a sampling-based approach by executing the applications with a small amount of data and an offline approach using a prediction model. In our comparison study, the sampling-based approach enabled higher performance than the offline approach. Finally, in Step 3, OpenCL executables are generated reflecting the hardware assignment.

We demonstrated the versatility of OptCL by using it with two existing HPDSLs, SYCL and OpenABL in two different hardware settings: a system using a CPU and a discrete GPU as well as an integrated CPU-GPU system. In an extensive study using various applications at different scales, OptCL outperformed existing solutions and exhibited significant speed ups. We showed that OptCL can be used to enable high-performance execution on heterogeneous hardware environments without in-depth knowledge of programming paradigms for the underlying hardware. Maximum speed-ups of 13× and 21× over the original compiler were achieved on the dCPU-GPU system and iCPU-GPU system respectively.

OptCL assumes users are aware of all the parallelisable opportunities and code them in the HPRs accordingly. A possible direction to improve our middleware is to automatically detect parallelisable code snippets. We will further consider extending the hardware support to more accelerators such as FPGAs not yet targeted by most existing HPDSLs.

References

1. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
2. Brown, K.J., et al.: Have abstraction and eat performance, too: optimized heterogeneous computing with parallel patterns. In: 2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Barcelona, Spain, pp. 194–205. IEEE (2016)

3. Chikin, A., Amaral, J.N., Ali, K., Tiotto, E.: Toward an analytical performance model to select between GPU and CPU execution. In: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Rio de Janeiro, Brazil, pp. 353–362. IEEE (2019)
4. Codeplay: Codeplay: ComputeCpp. https://www.codeplay.com/products/computecpp/. Accessed 30 July 2020
5. Cosenza, B., et al.: Easy and efficient agent-based simulations with the OpenABL language and compiler. Future Gener. Comput. Syst. **116**, 61–75 (2021)
6. Grewe, D., O'Boyle, M.F.P.: A static task partitioning approach for heterogeneous systems using OpenCL. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 286–305. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19861-8_16
7. Grosser, T., Hoefler, T.: Polly-ACC transparent compilation to heterogeneous hardware. In: Proceedings of the 2016 International Conference on Supercomputing, Istanbul, Turkey, pp. 1–13. ACM (2016)
8. Guzman, M.A.D., Nozal, R., Tejero, R.G., Villarroya-Gaudo, M., Gracia, D.S., Bosque, J.L.: Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL. J. Supercomput. **75**(3), 1732–1746 (2019)
9. Huang, S., et al.: Analysis and modeling of collaborative execution strategies for heterogeneous CPU-FPGA architectures. In: Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, Mumbai, India, pp. 79–90. ACM (2019)
10. Johnston, B., Falzon, G., Milthorpe, J.: OpenCL performance prediction using architecture-independent features. In: 2018 International Conference on High Performance Computing & Simulation (HPCS), Orleans, France, pp. 561–569. IEEE (2018)
11. Majeti, D., Sarkar, V.: Heterogeneous Habanero-C (H2C): a portable programming model for heterogeneous processors. In: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, Hyderabad, India, pp. 708–717. IEEE (2015)
12. Mittal, S., Vetter, J.S.: A survey of CPU-GPU heterogeneous computing techniques. ACM Comput. Surv. (CSUR) **47**(4), 1–35 (2015)
13. Moren, K., Göhringer, D.: Automatic mapping for OpenCL-programs on CPU/GPU heterogeneous platforms. In: Shi, Y., et al. (eds.) ICCS 2018. LNCS, vol. 10861, pp. 301–314. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93701-4_23
14. Navarro, A., Corbera, F., Rodriguez, A., Vilches, A., Asenjo, R.: Heterogeneous parallel_for template for CPU-GPU chips. Int. J. Parallel Program. **47**(2), 213–233 (2019)
15. Ohshima, S., Yamazaki, I., Ida, A., Yokota, R.: Optimization of hierarchical matrix computation on GPU. In: Yokota, R., Wu, W. (eds.) SCFA 2018. LNCS, vol. 10776, pp. 274–292. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-69953-0_16
16. Pandit, P., Govindarajan, R.: Fluidic Kernels: cooperative execution of OpenCL programs on multiple heterogeneous devices. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, Orlando, FL, USA, pp. 273–283. ACM (2014)
17. Pereira, A.D., Rocha, R.C., Ramos, L., Castro, M., Góes, L.F.: Automatic partitioning of stencil computations on heterogeneous systems. In: 2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), Campinas, Brazil, pp. 43–48. IEEE (2017)

18. Pérez, B., Bosque, J.L., Beivide, R.: Simplifying programming and load balancing of data parallel applications on heterogeneous systems. In: Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit, Barcelona, Spain, pp. 42–51. ACM (2016)
19. Pérez, B., et al.: Auto-tuned OpenCL kernel co-execution in OmpSs for heterogeneous systems. J. Parallel Distrib. Comput. **125**, 45–57 (2019)
20. Phothilimthana, P.M., Ansel, J., Ragan-Kelley, J., Amarasinghe, S.: Portable performance on heterogeneous architectures. In: Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, Houston, Texas, USA, pp. 431–444. ACM (2013)
21. Pérez, B., Stafford, E., Bosque, J., Beivide, R.: Sigmoid: an auto-tuned load balancing algorithm for heterogeneous systems. J. Parallel Distrib. Comput. **157**, 30–42 (2021)
22. Price, J., McIntosh-Smith, S.: Oclgrind: an extensible OpenCL device simulator. In: Proceedings of the 3rd International Workshop on OpenCL, Palo Alto, CA, USA. ACM (2015)
23. Rao, D.M., Thondugulam, N.V., Radhakrishnan, R., Wilsey, P.A.: Unsynchronized parallel discrete event simulation. In: 1998 Winter Simulation Conference. Proceedings (Cat. No. 98CH36274), Washington, USA, vol. 2, pp. 1563–1570. IEEE (1998)
24. Riebler, H., Vaz, G., Kenter, T., Plessl, C.: Transparent acceleration for heterogeneous platforms with compilation to OpenCL. ACM Trans. Archit. Code Optim. (TACO) **16**(2), 1–26 (2019)
25. Sbîrlea, A., Zou, Y., Budimlíc, Z., Cong, J., Sarkar, V.: Mapping a data-flow programming model onto heterogeneous platforms. In: Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems, Beijing, China, pp. 61–70. ACM (2012). https://doi.org/10.1145/2248418.2248428
26. Sotomayor, R., Sanchez, L.M., Blas, J.G., Fernandez, J., Garcia, J.D.: Automatic CPU/GPU generation of multi-versioned OpenCL kernels for C++ scientific applications. Int. J. Parallel Program. **45**(2), 262–282 (2017)
27. Steuwer, M., Fensch, C., Lindley, S., Dubach, C.: Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. ACM SIGPLAN Not. **50**(9), 205–217 (2015)
28. Tillet, P., Rupp, K., Selberherr, S.: An automatic OpenCL compute kernel generator for basic linear algebra operations. In: Proceedings of the 2012 Symposium on High Performance Computing, Orlando, FL, USA, pp. 1–2. ACM (2012)
29. Trigkas, A.: Investigation of the OpenCL SYCL programming model. Master's thesis, The University of Edinburgh, UK (2014)
30. Xiao, J., Andelfinger, P., Cai, W., Richmond, P., Knoll, A., Eckhoff, D.: OpenABLext: an automatic code generation framework for agent-based simulations on CPU-GPU-FPGA heterogeneous platforms. Concurr. Comput. Pract. Exp. **32**, e5807 (2020). https://doi.org/10.1002/CPE.5807
31. Xiao, J., Andelfinger, P., Eckhoff, D., Cai, W., Knoll, A.: Exploring execution schemes for agent-based traffic simulation on heterogeneous hardware. In: Proceedings of the International Symposium on Distributed Simulation and Real Time Applications, Madrid, Spain, pp. 1–10. IEEE (2018)
32. Xiao, J., Andelfinger, P., Eckhoff, D., Cai, W., Knoll, A.: A survey on agent-based simulation using hardware accelerators. ACM Comput. Surv. (CSUR) **51**(6), 1–35 (2019)