Customization of OpenCL Applications for Efficient Task Mapping under Heterogeneous Platform Constraints

Edoardo Paone*, Francesco Robino†, Gianluca Palermo*, Vittorio Zaccaria*, Ingo Sander† and Cristina Silvano*
*Politecnico di Milano, Italy. Email: {name.lastname}@polimi.it
†KTH Royal Institute of Technology, Sweden. Email: {frobino, ingo}@kth.se

***Abstract*—When targeting an OpenCL application to platforms with multiple heterogeneous accelerators, task tuning and mapping have to cope with device-specific constraints. To address this problem, we present an innovative design flow for the customization and performance optimization of OpenCL applications on heterogeneous parallel platforms. It consists of two phases: *1)* a tuning phase that optimizes each application kernel for a given platform and *2)* a task-mapping phase that maximizes the overall application throughput by exploiting concurrency in the application task graph. The tuning phase is suitable for customizing parameterized OpenCL kernels considering device-specific constraints. Then, the mapping phase improves task-level parallelism for multi-device execution accounting for the overhead of memory transfers — overheads implied by multiple OpenCL contexts for different device vendors. Benefits of the proposed design flow have been assessed on a stereo-matching application targeting two commercial heterogeneous platforms.**

I. Introduction

OpenCL [6] is a cross-platform programming model designed around the computational paradigm named *single program multiple data* (SPMD) [4], to exploit data parallelism on heterogeneous accelerators. However, while enabling functional portability between different accelerators (e.g. many-core fabrics and FPGAs), the OpenCL API does not ensure robust, cross-platform performance portability, for example in terms of throughput for streaming applications. On the one hand, OpenCL programs should be optimized for the target architecture by customizing platform related parameters in the code; on the other hand, task mapping over the available processing elements must be specific for each platform, leading to complex design space exploration (DSE).

When dealing with multiple heterogeneous devices, OpenCL falls short. While it allows device partitioning for running multiple tasks on the same device, it does not provide an easy way to orchestrate task execution on multiple heterogeneous devices. Still, it is possible to exploit multi-device execution using event-based synchronization, but this requires that i) the developer should write *ad hoc* host code to orchestrate multiple *command queues*, and ii) devices should belong to the same vendor platform. If the last condition is not met, the developer is forced to instantiate separate OpenCL contexts for different devices, taking care of copying data across contexts and using complex synchronization mechanisms.

All of the above conditions make it hard to exploit inter-task parallelism on multiple heterogeneous devices. To overcome these open challenges, in this paper we propose a semi-automatic customization and mapping methodology that brings three main contributions:

- An analytical method to efficiently identify the feasible subset of an OpenCL application design space, reducing the search space up to three orders of magnitude.
- An automated application auto-tuning technique to optimize parameterized OpenCL kernels for a target device, with up to 60% performance improvements.
- A task mapping technique, supported by a constraint solver, which provides optimized software pipelines up to 3x faster than single-device execution on the heterogeneous parallel platforms used in our experiments.

These results have been derived by assessing the proposed methodology on a stereo-matching application consisting of 9 different OpenCL kernels. However, the methodology is not limited to a specific application domain or class of kernels, and it can be generalized to any OpenCL application that consists of multiple parameterized kernels. The scalability of the proposed approach has also been analyzed in the experimental results.

II. Related Work

Optimization of OpenCL streaming applications on heterogeneous platforms has recently been addressed by a number of works based on Domain Specific Languages (DSLs), such as Halide [13], KernelGenius [9] and HIPAcc [10]. These works present different DSLs for the same application domain, namely image processing, and focus on a specific class of kernels, the stencil operation. The generated code outperforms hand-optimized programs, because the high-level description in a DSL enables fine-grained customization by means of source-to-source code transformations. However, these approaches are limited to a specific type of kernel operations and cannot address *kernel heterogeneity* in the application task graph. Moreover, they require a DSL description of the algorithm, so they do not enable porting of existing OpenCL application code.

The type of customization which we consider in this paper can be considered *coarse-grain*, since the application task graph is an input to our methodology and tasks are atomic. Similarly, the design flow proposed in [15] allows executing applications specified as synchronous dataflow (SDF) graphs on heterogeneous systems using OpenCL. While the work in [15] focuses on optimization of task scheduling for a given mapping, we explore at the same time mapping and scheduling – and the overhead implied by the required memory transfers – to find the optimal solution by means of an analytical model.

Other works in the literature follow a similar *coarse-grain* approach, such as SOCL [5] and OmpSs [3], but they use dynamic task mapping and scheduling. SOCL provides a unified OpenCL platform, built on top of the StarPU runtime system [2], to enable access to heterogeneous accelerators from different vendor platforms as if they belonged to the same platform. With respect to these works, our approach is suitable to embedded applications since i) it improves performance predictability by means of offline workload characterization and ii) it avoids the overhead of run-time scheduling.

Several works address the problem of task scheduling on heterogeneous platforms targeting only Nvidia GPUs and focusing primarily on load balancing [11], [18]. By targeting a

This work was partially funded by the European Commission under the grant CONTREX FP7-611146.

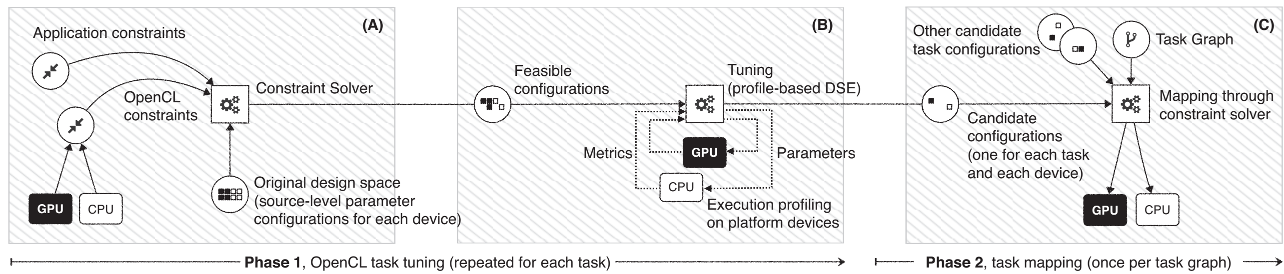


Fig. 1: Proposed Design Space Exploration (DSE) flow for tuning and mapping an OpenCL application to a heterogeneous platform.

homogeneous execution context on computing devices from the same vendor, these techniques can easily move tasks from one computing device to another and exploit specific features of the target architecture to hide memory transfers [18]. Sometimes, this hypothesis is used to completely ignore the cost of memory transfers in the timing model for GPU workload [8]. On the contrary, the framework in [11] does not support the execution of an application task graph on multiple devices in order not to deal with memory transfers, whereas our approach does.

Elastic computing [19] is another interesting approach, based on the complete separation of functionality from implementation. It uses multiple implementations of the same functionality, not limited to a specific application domain, to find the optimal mapping for a target heterogeneous platform. Differently from our methodology, the work in [19] does not start from a cross-platform programming paradigm such as OpenCL. On the one hand, this limits application portability to the set of available kernel implementations; on the other hand, kernel customization for the target platform is not needed.

III. Proposed Methodology

The methodology for the customization of OpenCL applications consists of two main phases: tuning and mapping, as illustrated in Figure 1. It applies to multi-task applications expressing task-level parallelism in the form of a *large grain data flow* (LGDF) graph [7], [1]. Each task is supposed to have a *parametric design*, i.e. the task behavior can be customized by tuning source level parameters. For an OpenCL application, the term *task* refers to the execution of an OpenCL kernel on a platform device, by means of the *clEnqueueNDRangeKernel* API [6]. Therefore, a task includes an iteration space over an *NDRange*, to parallelize the kernel execution on separate data blocks. For OpenCL kernels, the source level parameters include constants used to allocate buffers in local memory and iterate over the input data, as well as size of the global and local grids which define the *NDRange*. The choice of parameters has been inspired by the work in [16], which provides an overview of the OpenCL performance pitfalls in porting OpenCL applications from GPU to CPU devices. Since the transformations proposed in [16] do not alter the parallelization or the structure of the original OpenCL code, they could be implemented as a parameterization of the OpenCL code. In this case, code specialization becomes a generic form of parameter tuning, providing good opportunities for automation and improving inter-platform OpenCL performance portability.

The proposed methodology consists of two main phases, as illustrated in Figure 1. In Phase 1, each task is first considered independently to apply task-level optimization. We use an analytical technique to identify the subset of source level parameters that satisfy the constraints of each available platform device (Phase 1, Block A). Then the feasible configurations pass through a *tuning* phase that removes Pareto-dominated solutions (Phase 1, Block B). At the end of Phase 1, each task is associated with its best parameter configurations for each device although it has not yet been assigned to any specific device. At this point, the parametric configurations of the same task on different devices may be significantly different due to the device constraints. In Phase 2, Block C implements the mapping problem, by means of a solver based on constraint programming [12]. The input to the solver is the application task graph and the pruned set of task configurations coming from the previous phase. Phase 2 considers, as its optimization objective, the application throughput, taking into account the overhead of host-to-device and device-to-host memory transfers.

To better illustrate the steps of the proposed methodology, we introduce a synthetic OpenCL application. The final goal is to map this application to different platforms, by exploiting inter-task parallelism. The application Large Grain Data Flow (LGDF) is shown in Figure 2. It consists of OpenCL kernels with different memory access patterns and different memory requirements, as well as multiple instances of the same kernel processing different amounts of data. The input data stream is represented by blocks A and B. Data is processed by multiple tasks in a *pipelined* modality, whereas tasks are instances of a matrix multiplication (MULT), a matrix add (ADD, which adds a constant matrix K) and a matrix stencil (S2D) operator.

Considering the execution context, there are different platforms on which we might want to run the above application. Each OpenCL platform may have specific features that could affect the overall partitioning onto multiple, heterogeneous devices. To address this problem, Phase 1 of the methodology (Section III-A) focuses on each kernel of the application to prune its source-level design space, then Phase 2 (Section III-B) considers the application task graph as a whole.

A. DSE Phase 1 – Task tuning

This section describes how task tuning applies to one of the tasks of the application shown in Figure 2: the *matrix multiplication* (MULT). We consider an extended version of the original matrix multiplication sample available in the Altera OpenCL SDK[2]. OpenCL allows to run the very same code provided by Altera for an FPGA target on a CPU or GPU, while the contribution of our methodology consists of customizing source-level parameters for the target platform.

The original implementation already exploits *tiling* – by means of buffers in local memory – in order to reduce the global memory access bandwidth. However, to broaden the scope of this investigation, the application has been modified to let the *workgroup size* be set independently from the *tile size*.

[2]http://www.altera.com/products/software/opencl/opencl-index.html

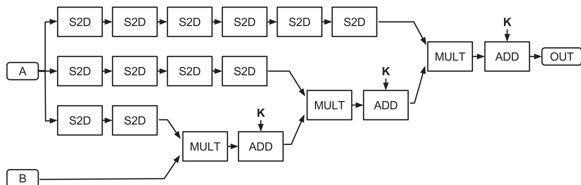


Fig. 2: Synthetic OpenCL task graph used to describe the methodology.

Besides, the workgroup shape has also been changed from a squared to a rectangular one – with dimensions `wg_size_x` and `wg_size_y`. Finally, two additional parameters allow to increase the work for a single work-item, by making it process a block of dimensions `block_size_x` × `block_size_y`.

This modification adds more opportunities for the mapping stage but, since the tile size is forced to be a square, it provides an additional set of constraints to be met:

- `tile_size_x` = `wg_size_x` × `block_size_x`
- `tile_size_y` = `wg_size_y` × `block_size_y`
- `tile_size_x` == `tile_size_y`
- The size of the tile must meet the limits of the local memory size of the device.
- The size of input matrices, in each dimension, must be a multiple of the tile size.

By solving at design time the last constraint, the methodology allows to simplify the control logic for checking the boundary conditions within the kernels themselves. This is important for GPU architectures, whose performance is affected by *thread divergence*. As described in the next section, sizing of the design parameters is done through a constraint problem formulation.

Constraint problem formulation. In constraint programming (CP), a problem is formulated using constraints to define relations between variables, then a solver generates solutions that satisfy such constraints [14]. This makes CP a form of declarative programming, since the constraints do not specify a sequence of steps to find a solution, but rather the properties of a solution to be found.

The kernel parameterization problem is formulated as a CP problem, using the formalism provided by the Minizinc software tool [12]. We define a modular composition of variables and constraints by dividing them in two sets:

A) OpenCL platform variables and constraints on the dimension of the OpenCL workgroups and local memory size.

B) Application-specific variables and constraints on the properties of the tiles used by the matrix multiplication kernel.

The problem formulation consists of a sharp separation between the definition of generic constraint rules and the specialization of these on a specific platform. This represents one of the advantages of using CP. For example, considering the platform constraints, there could be different values for the `max_wg_size` bound that are dependent on the device actually used (CPU or GPU).

The application-specific constraints represent an extension of the more general OpenCL platform constraints. Thus, another advantage of using CP is that the problem formulation can be easily extended. Moreover, the problem formulation is independent from the search strategy used by the solver, which allows to evaluate – or eventually to develop in parallel – an optimized search strategy. Finally, when dealing with minimization or maximization goals (like the task mapping problem in Section III-B), the solution is potentially optimal.

The output of the constraint solver (Block A in Figure 1) is the feasible solution set for matrix multiplication, which is a subset of the original design space. The benefit of applying this technique is a drastic reduction of the search space and, more importantly, of the exploration time, while not disregarding any feasible – and therefore possibly optimal – solution.

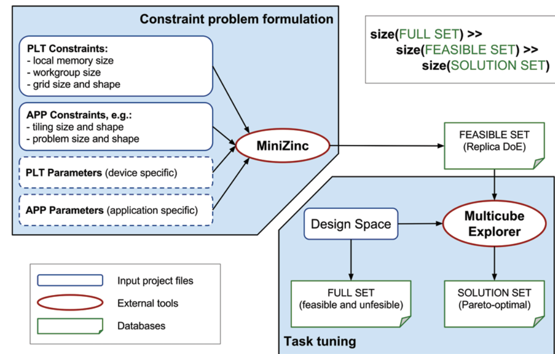


Fig. 3: The proposed DSE framework implementing Phase 1.

Task tuning. The constraint programming solution is used as a starting point for the next sub-phase — the *task tuning* (Block B in Figure 1). Here, for each task in the LGDF, we search over the feasible set in the application design space to find the optimal configuration with respect to some design objective. In our flow, the optimization objective consists of minimizing the kernel execution time, by averaging the profiled metrics over a dataset representative for the target application in order to account for data-dependent variability.

To automate this process, we use Multicube Explorer [17], an open-source profile-based design space exploration (DSE) tool for multi-objective optimization. Figure 3 shows the tools used in the proposed design flow: the MiniZinc solver, implementing the analytical technique described in the previous section, allows for fast pruning of those configurations of an OpenCL kernel unfeasible with respect to device constraints; then, Multicube Explorer adopts heuristics to navigate and prune the feasible solution set, by executing and profiling the kernel on the target device.

B. DSE Phase 2 – Task mapping

The input of Phase 2 (see Figure 1) is a set of optimal configurations for each task – one for each platform device – derived in the previous phase. Each configuration is characterized by its own *kernel execution time* (EXE), as measured by the task tuning in Phase 1. However, in order to also take into account inter-task communication when tasks are mapped to different devices, additional information is needed: i) the time required to copy input data from host to device buffers (H2D), and ii) the time to retrieve the kernel output from the device buffers (D2H). To measure the communication overhead, the application is structured as a network of *components*, where each component is an instrumented version of the original OpenCL kernels. This instrumentation allows to measure H2D and D2H average times, which are later used by the constraint solver to find the optimal task mapping, as described below.

Task graph mapping. To expose concurrency, the mapping process is based on well-known pipelining principles [7]; it corresponds to identifying *feed-forward cut-sets* in the LGDF and placing *buffers* in-between. Each cut-set of the task graph becomes thus a pipeline stage. The optimal mapping – with respect to the throughput – minimizes the critical path of the

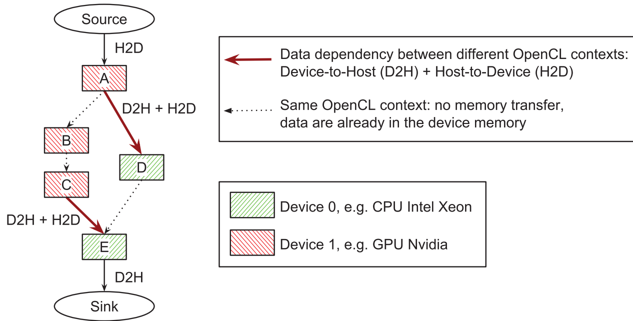
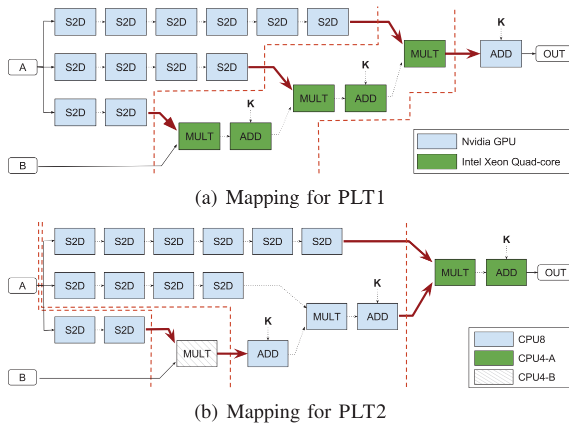


Fig. 4: Task mapping example.

decomposed graph, i.e. the slowest pipeline stage for kernel execution time and buffering overhead (H2D, D2H).

Figure 4 shows a minimal OpenCL task graph mapped to a heterogeneous platform with different OpenCL contexts, since the computing devices belong to different vendors (e.g. Nvidia for the GPU device and Intel for the CPU one). When two tasks, connected by an edge in the task graph, are mapped to computing devices belonging to different OpenCL contexts (e.g. tasks A-D or C-E), the data dependency requires a copy of data buffers from one device to another, which results in a D2H and H2D operation. Different accelerators – and even the same accelerator in different PCI slots – can experience significantly different communication times. For example, the memory transfers from GPU on our target platform PLT1 (see Section IV) are 46% slower than from the CPU. These transfer times are taken into account in the analysis and optimization.

The mapping problem is solved as a constraint optimization problem. The program formulation is applicable to any *acyclic* OpenCL task graph; it defines a set of constraints in terms of precedence rules and memory transfers that are applied only when required by a specific mapping. In fact, H2D and D2H overhead times are taken into account only when two tasks connected through an edge – data dependency – are scheduled in different OpenCL contexts; otherwise, they are ignored, since data are already in the device memory. In the final result, multiple tasks could be executed on the same device. This corresponds to instantiating a task that wraps several kernels, ensuring that internal scheduling of each kernel meets the dependencies expressed in the original task graph.



(a) Mapping for PLT1

(b) Mapping for PLT2

Fig. 5: Optimal mappings of the synthetic task graph for to two heterogeneous platforms, described in Section IV.

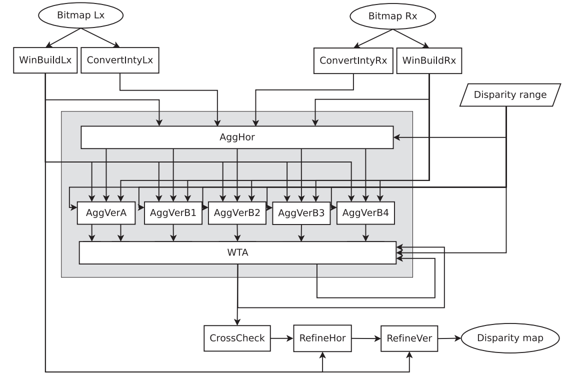


Fig. 6: Task graph of stereo-matching OpenCL kernels.

Figure 5 shows the application task graph associated with the mapping solutions found for the two platforms used in the experimental results. Kernels with the same color are mapped to the same device while the dashed lines show the *feed-forward cut-sets* in the LGDF, which define different pipeline stages.

IV. Experimental Setup

The presented flow has been automated to a large extent: the task tuning phase is fully automated and the constraint solver, both for pruning the unfeasible solutions and task mapping, directly provides the solution. The only information required as input is the list of kernel constraints and customizable parameters. Nevertheless, the application host code is not generated automatically, but we rely on template code to realize the final mapping of OpenCL kernels to platform devices. This enabled us to validate the proposed methodology on a real world use case: an OpenCL stereo-matching application targeted to two commercial heterogeneous parallel platforms. The application implements the stereo-matching (SM) algorithm described in [20]. For each input pair of stereo images, SM estimates the depth of the objects in the captured scene, by computing the *pixel disparity* between the left and the right frame of the stereo image: the higher the pixel disparity, the closer the object to the viewpoint (e.g., a camera). SM is a streaming application, since it processes an input stream of stereo frames, and therefore the *throughput* of the application (frame-rate) can benefit from the software pipelining technique proposed in Section III-B.

The SM behavior can be tuned by setting some specific parameters, to trade-off the *throughput* (frame-rate) and the *Quality-of-Service* (QoS), which measures the accuracy of the result. Our analysis is focused on the parameter which controls the step between consecutive disparity hypotheses, the `hypo_step`. Since the algorithm mainly consists of minimizing the matching-cost while iterating over the disparity range, the tuning of `hypo_step` directly affects the performance in terms of *throughput* and *QoS*, mainly the *disparity error* (the average error in the pixel disparity computation).

Figure 6 shows the OpenCL kernels invoked by SM. Some of them are instantiated several times (*WinBuild*, *ConvertInty*, *AggVer*) so that each instance can be considered a different task. The gray box represents one step of the iterative cost aggregation, which is repeated for each disparity hypothesis. The OpenCL kernels within this box have been modeled as a single task, in this experiment, for two reasons: 1) our constraint program formulation for solving the mapping problem currently supports only tasks with a single output and 2) the tasks in

this box represent steps of a cost aggregation super-task which would not benefit from execution on different devices.

In this experiment we use a stream of input frames with resolution 384×288 and we fix the maximum disparity hypothesis to 24: thus, given an anchor pixel in the left stereo frame, its corresponding pixel in the right frame should have a maximum horizontal offset of 24 pixels. We consider 4 values of `hypo_step`, from 1 to 4, corresponding to four different ranges of disparity hypotheses. Figure 7 shows the task graph flattened with `hypo_step` set to 1, i.e. the largest task graph, which results in 24 cost-matching tasks.

For the validation, we used two heterogeneous platforms:

PLT1: Workstation with Intel Xeon Quad-Core E5-1607 at 3.0 GHz and 8 GB RAM. OpenCL 1.2 for the CPU, provided by Intel OpenCL SDK 2013. Discrete GPU NVIDIA Quadro NVS 300, with OpenCL 1.1 provided by CUDA SDK v5.5.

PLT2: NUMA machine with four nodes, each a Quad-Core AMD Opteron Processor 8378 at 2.4 GHz, with 8 GB of RAM per node. OpenCL 1.2 provided by AMD OpenCL SDK v2.8.1.

The CPU nodes on PLT2 are exposed by the OpenCL runtime as a homogeneous multi-core CPU device. In order to emulate a heterogeneous platform, we used the device fission API [6] to partition the 16 cores into 3 CPU sub-devices: 1 with 8 cores and 2 with 4 cores each. Thus, both target platforms provide heterogeneity: architectural heterogeneity in PLT1 and computational heterogeneity in PLT2.

V. Experimental Results

The design flow is applied to task optimization and mapping of OpenCL stereo-matching, presented in the previous section.

DSE Phase 1 – Task tuning. The constraint solver is very efficient in finding the feasible task configurations: the feasible sets of all SM kernels on all target devices were found in $11s$, using one CPU core on PLT1. The full design space contains 2^{24} points for the GPU device on PLT1, while it is even larger on the CPU devices since they support larger workgroups. For this experimental setup, the feasible set is always smaller than 0.1% of the full design space for all kernels and target devices. These cut-sets represent the result of Phase 1, Block A: they are very precise and capture all the candidate optimal solutions. This shows a first interesting result: by adopting the *ad hoc* filtering based on constraint programming, the design space to be explored was reduced by three orders of magnitude.

The optimization phase (Phase 1, Block B) is driven by the profile-based DSE tool, Multicube Explorer, which was configured for this experiment to select up to 50% points of the feasible set and to minimize the average kernel execution time. The percentage of points can be tuned to control the trade-off between accuracy of the solution and exploration time [17]. Since we set a single design objective for the optimization of OpenCL kernels, the result is one configuration of parameters, specific for the target device, which is optimal with respect to the average throughput. In the original CUDA implementation of the stereo-matching application [20], the number of threads per block (equivalent to the workgroup size in OpenCL) is 16x16. This design choice was based on the maximum number of CUDA threads per block and on code optimizations for data fetching into local memory [20]. Thus, we consider a reference configuration of the stereo-matching kernels with 16x16 workgroup size and 1x1 block size. Figure 8 shows that the tuning phase provides optimized task configurations, with up to 60% performance improvement on the target CPUs.

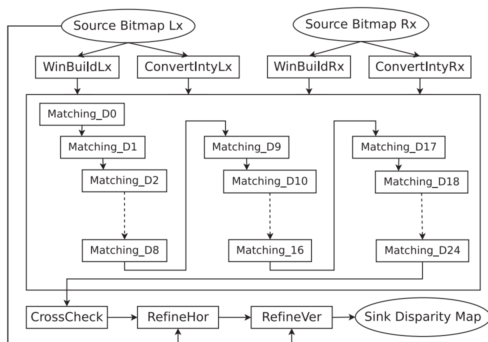


Fig. 7: Flattened task graph of stereo-matching OpenCL kernels, with parameter *hypo_step* set to 1 (24 disparity hypotheses).

DSE Phase 2 – Task mapping. We consider 4 different task graphs, one for each value of `hypo_step`, depending on the *QoS* (result accuracy) required by the user. For each task graph configuration, the mapping problem is solved to identify the optimal mapping on the target heterogeneous platforms. These points represent the solution of a multi-objective optimization, since *throughput* and *QoS* are two design objectives.

Figure 9 reports cycle time (the time between two consecutive output frames) with respect to the average disparity error (the higher the `hypo_step`, the higher this error) for PLT1 and PLT2. For each platform, four mapping configurations have been considered, by varying from a single device up to multiple heterogeneous devices. This plot shows the improvement on the throughput metric for a given level of average disparity error. The maximum observed improvement is 3x when `hypo_step` is 1, thus for the largest task graph shown in Figure 7. Not necessarily the throughput increases when using multiple devices: the fourth configuration in Figure 9 shows a sub-optimal mapping, where tasks are equally divided among all available devices. On PLT1, this configuration does not perform better than the single-device execution on CPU: the reason is that the speedup contribution from multi-device execution (CPU and GPU) does not compensate the overhead of H2D and D2H memory transfers for copying data from one device to the other. Thus, without the proposed optimization driven by a constraint solver, it might be difficult for the developer to find a mapping that balances the pipeline stages and maximizes the throughput.

Scalability. The flattened model of Figure 7 is given as input to the constraint program to identify the best mapping which maximizes the average throughput. By setting a constraint on the maximum cycle time ($150ms$ on PLT1 and $250ms$ on PLT2), the Minizinc solver identifies a solution that satisfies the mapping problem of Section III-B. The search time is less than $2s$ for all values of `hypo_step`, except for `hypo_step` set to 1 on PLT2 ($> 30min$). In this case, the constraint solver spends most time in evaluating all possible mappings (3 platform devices) of the cost-matching cascade (24 tasks). To cope with this problem, we added one constraint to the mapping formulation, based on the analysis of the task graph. When a sequence of tasks is connected like in the cost-matching cascade, i.e. the output of one task is the input to the next task, the optimal solution will be found by splitting this cascade into up to ND sub-ranges, where ND is the number of platform devices. This mapping minimizes the time overhead of H2D and D2H memory transfers between consecutive tasks. By adding the new constraint, the search time is reduced to $2.5s$.

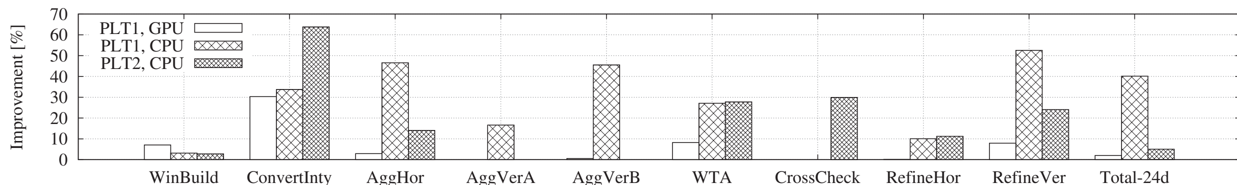


Fig. 8: Performance improvement of the stereo-matching kernels, after applying the task tuning phase, vs. the original implementation.

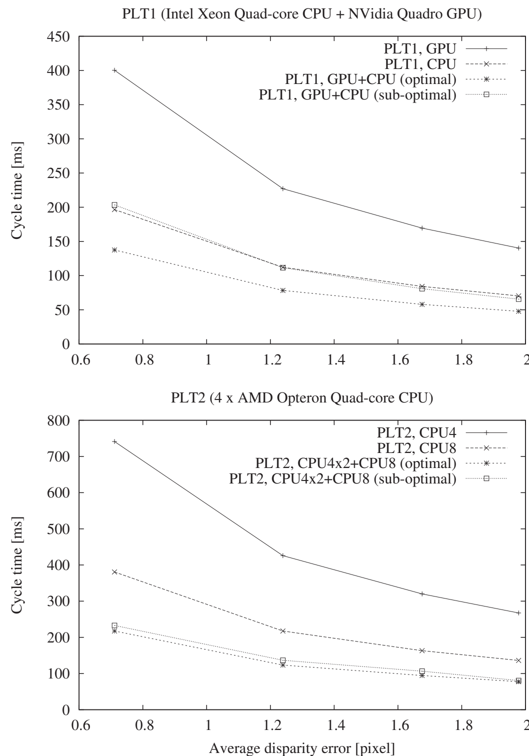


Fig. 9: Trade-off between cycle time and disparity error of the stereo-matching task graph for different mapping configurations.

VI. Conclusions

We proposed an optimization and customization design flow to efficiently map OpenCL applications to heterogeneous computing platforms. For such platforms, it is generally hard to exploit pure data-parallelism, since devices from different vendors have to be instantiated as different OpenCL contexts. The conversion to a task-parallel implementation, if possible, may suffer from unnecessary data transfers from host to device, and vice-versa, and requires manual optimization to achieve good performance for each device. Our design flow is based on a constraint solver to identify a set of feasible configurations with respect to platform-specific constraints and on a DSE framework for tuning task parameters. The constraint solver automatically pruned the search space of the stereo-matching OpenCL kernels to 0.1% of the original size. Combined with the proposed mapping methodology, this enabled the generation of optimized OpenCL pipelines, some of them even 3x faster than the single-device implementation for the selected use cases. In this work, the scheduling of tasks mapped to the same computing device has been limited to satisfy data dependencies between kernels. Future work will investigate more advanced scheduling techniques to overlap computation and communication.

References

[1] W. Ackerman. Data Flow Languages. *Computer*, 15(2):15–25, 1982.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice & Experience*, 23(2):187–198, Feb. 2011.

[3] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21:173–193, 2011.

[4] M. Flynn. *Flynn's Taxonomy*, pages 689–697. Springer US, 2011.

[5] S. Henry, A. Denis, D. Barthou, M.-C. Counilh, and R. Namyst. Toward OpenCL Automatic Multi-Device Support. In *Euro-Par 2014 Parallel Processing*, pages 776–787. Springer, 2014.

[6] Khronos Group. OpenCL Specification, v1.2. https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, 2012.

[7] E. Lee and D. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept 1987.

[8] H. Lee and M. A. Al Faruque. GPU-EvR: Run-time Event Based Real-time Scheduling Framework on GPGPU Platform. In *Proceedings of Design, Automation & Test in Europe*, DATE '14, pages 220–220, 2014.

[9] T. Lepley, P. Paulin, and E. Flamand. A Novel Compilation Approach for Image Processing Graphs on a Many-core Platform with Explicitly Managed Memory. In *Proceedings of Compilers, Architectures and Synthesis for Embedded Systems*, CASES '13, pages 1–10, Piscataway, NJ, USA, 2013. IEEE.

[10] R. Membarth, F. Hannig, J. Teich, M. Korner, and W. Eckert. Generating Device-specific GPU Code for Local Operators in Medical Imaging. In *Proceedings of Parallel Distributed Processing Symposium*, IPDPS'12, pages 569–581, May 2012.

[11] R. Membarth, J.-H. Lupp, F. Hannig, J. Teich, M. Körner, and W. Eckert. Dynamic Task-scheduling and Resource Management for GPU Accelerators in Medical Imaging. In *Proceedings of Architecture of Computing Systems*, ARCS'12, pages 147–159. Springer-Verlag, 2012.

[12] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543. Springer, 2007.

[13] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[14] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier Science Inc., Oct. 2006.

[15] L. Schor, A. Tretter, T. Scherer, and L. Thiele. Exploiting the Parallelism of Heterogeneous Systems using Dataflow Graphs on Top of OpenCL. In *IEEE 11th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia), 2013*, pages 41–50, Oct 2013.

[16] J. Shen, J. Fang, H. Sips, and A. Varbanescu. Performance Traps in OpenCL for CPUs. In *Proceedings of Parallel, Distributed and Network-Based Processing*, PDP'13, pages 38–45, Feb 2013.

[17] C. Silvano et al. MULTICUBE: Multi-objective Design Space Exploration of Multi-core Architectures. In *2010 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 488–493, July 2010.

[18] A. Tarakji, M. Marx, and S. Lankes. The development of a scheduling system GPUSched for graphics processing units. In *Proceedings of High Performance Computing and Simulation*, pages 566–575, July 2013.

[19] J. R. Wernsing and G. Stitt. Elastic Computing: A Framework for Transparent, Portable, and Adaptive Multi-core Heterogeneous Computing. *SIGPLAN Not.*, 45(4):115–124, Apr. 2010.

[20] K. Zhang, J. Lu, Q. Yang, G. Lafruit, R. Lauwereins, and L. Van Gool. Real-Time and Accurate Stereo: A Scalable Approach With Bitwise Fast Voting on CUDA. *IEEE Transactions on Circuits and Systems for Video Technology*, 21(7):867–878, July 2011.