Automatic OpenCL Task Adaptation for Heterogeneous Architectures

Pierre Huchant(✉), Marie-Christine Counilh, and Denis Barthou

Inria/LaBRI, University of Bordeaux, Bordeaux INP, Bordeaux, France
{pierre.huchant,denis.barthou}@inria.fr, counilh@labri.fr

Abstract. OpenCL defines a common parallel programming language for all devices, although writing tasks adapted to the devices, managing communication and load-balancing issues are left to the programmer.

In this work, we propose a novel automatic compiler and runtime technique to execute single OpenCL kernels on heterogeneous multi-device architectures. The technique proposed is completely transparent to the user, does not require off-line training or a performance model. It handles communications and load-balancing issues, resulting from hardware heterogeneity, load imbalance within the kernel itself and load variations between repeated executions of the kernel, in an iterative computation. We present our results on benchmarks and on an N-body application over two platforms, a 12-core CPU with two different GPUs and a 16-core CPU with three homogeneous GPUs.

1 Introduction

Heterogeneous parallel architectures are ubiquitous, from supercomputers to cell phones. Developing an application for a heterogeneous, multi-devices system, taking advantage of all available devices is extremely challenging. OpenCL is a standard language for the development of code on heterogeneous architectures. It leverages part of this difficulty by defining one language for all platforms, and by structuring parallelism into a task graph, where tasks are parallel computations to be mapped onto one device. However, this implies that the developer has to design as many tasks as there are devices, with tasks adapted in terms of parallelism and memory granularity: There should be enough parallelism for all devices, and communications between devices have to be explicit.

OpenCL kernels describe tasks as parallel work-groups. To transform one kernel into as many kernels as devices, these work-groups have to be partitioned among devices. This raises load balancing issues, stemming from device heterogeneity and from workload variation between work-groups. As many kernels are executed in iterative computation, the workload may change from one iteration to the other, requiring a constant adaptation of the work-group partitioning. Moreover, data has to be split among partitioned work-groups in order to reduce communication time and written data has to be merged upon kernel completion. Achieving adaptive work-group partitioning, with no training, handling load balancing and data movements has never been conducted before.

P.-F. Dutot and D. Trystram (Eds.): Euro-Par 2016, LNCS 9833, pp. 684–696, 2016.
DOI: 10.1007/978-3-319-43659-3_50

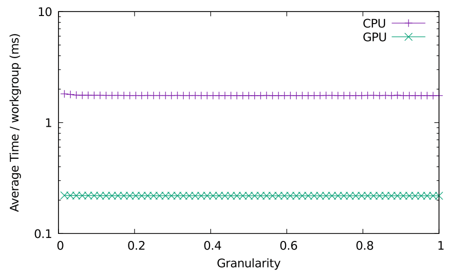
We propose in this paper a static/dynamic approach for the execution of any given OpenCL kernel on a multi-device heterogeneous architecture. The method tackles, without training, load balancing issues coming from device heterogeneity and from varying computational intensity inside the kernel, when the kernel is called multiple times. The load-time analysis computes how to partition data for the execution of work-groups as a function of the work-group partitioning. The dynamic method evaluates from previous runs how to partition work-groups and splits/merges data accordingly. We show with two different runtime methods that only a few iterations are required to reach the optimal granularity. Finally, when the kernel computational intensity changes with each execution, the method dynamically adapts the load and stays close to the optimal load.

Section 2 shows causes for load balancing issues when splitting a kernel into subkernels. Section 4 presents our method, generating partition-ready kernels and instantiating with the appropriate granularity, for each device. Computation of granularities is described in Sect. 5. Related works and experimental results are given in Sects. 6 and 7.

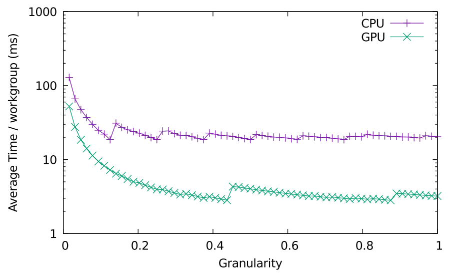
2 Motivating Example

Given an OpenCL kernel, we define a subkernel as a code executing only part of the kernel computation. As OpenCL kernel executions are characterized by the number of parallel work-groups, the ratio of work-groups of a subkernel over the total number of work-groups is called *granularity*, a granularity of 1 meaning the whole kernel is executed. We study the performance variation of a kernel on one device, decreasing manually its granularity. The granularity is indicated as a percentage of the total number of work-groups, and performance is indicated as the mean time per work-group (lower is better). Figure 1 shows performance of AESEncrypt and EP from SNU NPB Suite [13] for different granularities on a 16-core Intel Xeon E5-2650 2.00 GHz with 64GB (CPU) and on an Nvidia Tesla M2075 (GPU). For AESEncrypt, the average time per work-group is nearly constant for all granularities, and very different on CPU and on GPU. For EP, we observe large performance drops (higher average time/work-group) at regular intervals of granularities on both CPU and GPU. This may come from compiler optimizations (such as unrolling), cache effects, and inefficient occupancy of the parallel resources due to a low number of work-groups within subkernels.

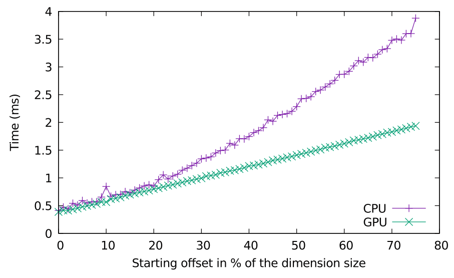
Work-groups are indexed in OpenCL by a vector of indices among a rectangular space (from 1D to 3D) called the `NDRange`. Selecting a granularity boils down to defining a subvolume of indices. In this paper, the subvolumes we consider are obtained by selecting one smaller interval in one dimension of the `NDRange`. The *offset* is the first index of this interval of indices, the granularity defining the size of this interval. Figure 1c shows for a Sparse Matrix Vector Multiply (SpMV) the influence of the offset on performance when, for a kernel of 1/4 granularity, the offset is changed. In the chosen sparse matrix, rows with a high index have more non-0 elements than those with a low index. This accounts for the execution time increase for large offsets, more than 7× the time of a 0-offset. When splitting a kernel into subkernels, this is a possible source of load-imbalance.



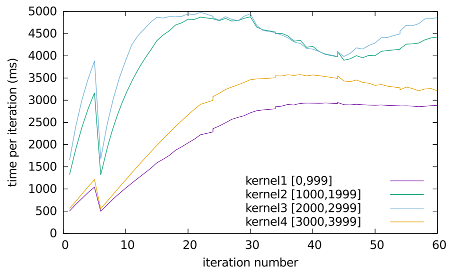
(a) AESEncrypt



(b) EP



(c) SpMV



(d) OTOO

Fig. 1. (a) and (b): Impact on performance of architectural heterogeneity and granularity on AESEncrypt and EP benchmarks. Performance is given as an average time per work-group, granularity as a percentage of the total number of work-groups. (c) Impact on performance of the offset (starting index) for SpMV kernel, with a fixed granularity of 1/4. (d) Impact of iteration count on performance for OTOO application. Granularities are set to 1/4 for all devices, and offset is fixed on all devices. (Color figure online)

Many OpenCL kernels are executed in iterative computations. For instance, OTOO [11] is an astrophysics particle N-Body simulation and the same kernel is called repeatedly to compute forces and move the different particles. Figure 1d shows how the execution time changes for different iterations, for different offsets, for each iteration of the computation. The kernel is split into 4 subkernels, each one is given a granularity of 1/4 and executed on one GPU. The input set corresponds to a non-uniform distribution of the masses in space. As this space is partitioned among the work-groups, this results in a non-homogeneous load distribution among the work-groups, changing with iteration number.

These results advocate for a method able to cope with the heterogeneity of the hardware, but also with the performance variations associated to different granularities, depending on the offset and varying with each execution of the kernel. Adapting a single OpenCL kernel to an heterogeneous architecture with any number of devices, taking into account these four sources of imbalance has never been tackled before.

3 Principle of Adaptive Granularity

The method proposed is threefold. First the kernel is analyzed and a new version, partition-ready, is generated at compile time. Then each time the original kernel has to be executed, a granularity is chosen for each device, based on previous executions if any, and the partition-ready kernel is instantiated on each device with the chosen granularity. More precisely: (i) When the kernel code is first loaded, it is analyzed. This step is more thoroughly described in Sect. 4. The objective is the generation of a parametric and partition-ready kernel, executing only a slice of the `NDRange` space. The analysis is performed once on the OpenCL code (no host code analysis) but the code generated can be instantiated at run-time for many different granularities and offsets. The slicing of the rectangular volume corresponding to the NDRange is done in any of the dimensions of the volume and does not require to flatten it. The memory region accessed by each work group is computed, parametrically w.r.t. the work-group id and the scalar parameters of the kernel; (ii) Each time the original kernel is launched, a granularity for each device is determined. This granularity determines the number of work-groups to execute on a particular device. The array regions are instantiated with the granularities and the actual parameters of the kernel. Depending on the result, all arrays are communicated to the devices or only the region they require. The same occurs for bringing back data from the devices. This kernel instantiation is described in Sect. 4.2; (iii) The execution time of each kernel execution is collected for refining the granularity in the possible following runs. This iterative granularity optimization is described in Sect. 5.

4 Automatic Adaptation of Data and Parallelism

We describe in this section how a kernel is analyzed and transformed into a parametric partition-ready kernel, function of the granularity.

4.1 Static Analysis and Transformation

The analysis determines for each array passed to the kernel how this array can be split among the different devices. For arrays that are read-only, a safe over-approximation is to broadcast the whole array to all devices. A more precise analysis can determine a finer partition, allowing shorter communication times and for some extreme case, may be the only possible way to execute the kernel if the initial array is too large for any of the devices. Finding how to partition arrays written by the kernel is essential: When the written region is precisely known and there is no overlap with other device regions, bringing back this data to the host can be done in parallel. On the contrary, if the analysis is not able to precisely determine which region has been written, a merge operation is necessary to build the output array [8,12]. The analysis only handles arrays (buffer objects) but could be extended to OpenCL images. We describe in this section how to determine precisely the array regions accessed by each work-item. The case where the analysis fails is discussed in the next section.

We first identify in the kernel all statements accessing arrays passed as a parameter. In OpenCL, arrays can be cast into other types (from 1D to 3D for instance), with possible offsets. All accesses through the cast arrays are also accesses to the initial array. Likewise, array accesses may occur inside functions called by the kernel. We therefore resort to an inter-procedural alias analysis, following assignments and use-def chains on arrays. In the following example,

```
void KERNEL(float *A) {
    ...
S1: double(* B)[3][5][5] =(double (*) [3][5][5])&A[ offset ];
    ...
S2: B[1][0][0]=..
}
```

the analysis detects that statement `S1` defines the 3D array `B`, aliasing `A`. `S2` accesses `B[1][0][0]`, corresponding to `A[25+offset]`. Only constant offsets are handled so far. It generates a list of statements accessing input arrays, with their mapping function turning the index into an index of the input array.

For each array that is a parameter of the kernel and for each statement, we compute the array region accessed by this statement. The idea is to consider the index expression and to replace the variables in it by their values, repeatedly. Assuming the code is in SSA-form, this repeated substitution may lead to several cases: The resulting expression only uses scalar parameters of the kernel and work-item ids (local, group or global). In such case, the substitution process stops and the exact region will be evaluated dynamically, when these ids and parameter values are known. When a variable is defined by a Φ-function, if this is an induction variable and its interval of values can be determined, then the variable is replaced by its interval, the index expression becoming an interval expression. For other cases of Φ-functions or variables that are defined by loads, the region accessed is assumed to be *unknown*. Therefore, the array regions computed are interval expressions of the scalar inputs of the kernel and of the work-item ids, or *unknown*. We compute additionally the conditions on the ids for which this region is accessed. The abstraction we use for these conditions is an interval on ids. A similar analysis is therefore conducted on the conditionals (or loop bounds) governing the execution of the statement considered. Out of simplicity, only uniform expressions on ids are kept, *i.e.* inequalities of the form: $\pm id \leq expr$ where $expr$ is an expression independent of the work-item ids. Conditionals that are not uniform expressions are assumed to be true. The conjunction of such conditionals define intervals of ids. To wrap-up, the array region accessed by a statement is either *unknown*, or represented by a guarded region of the form:

$$\textbf{if}\ \ id \in [lb(s), ub(s)] : [expr_1(id, s), expr_2(id, s)]$$

with s the scalar parameters of the kernel and id a work-item id. Note that the expressions $expr_1$, $expr_2$, lb and ub have no restriction and can use any operator allowed by the language. For a given array, the region accessed by the kernel is defined by the union of array regions accessed by all statements.

Kernel Modification. When executing a kernel with a fraction of the original `NDRange`, some syntactic modifications are needed in order to keep the correct

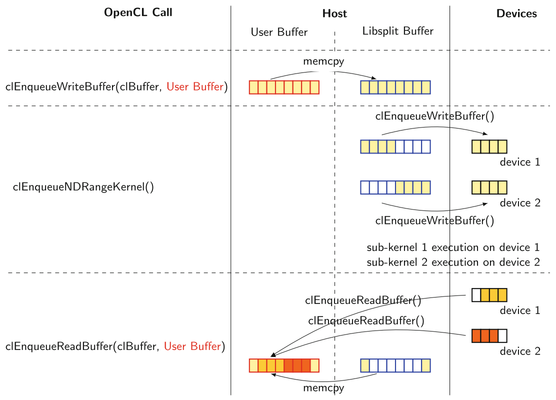


Fig. 2. Buffer management

semantics. Indeed, the global size is different from the original kernel, the number of work-groups has changed and their id has changed too. Two additional parameters are added to the partition-ready kernel: `splitdim` (1, 2 or 3) accounts for the dimension of the `NDRange` that is split, and `numgroups` is the number of work-groups in this dimension. Then the following function calls are changed:

Expression	Rewritten into
`get_global_size(expr)`	`(expr == splitdim?numgroups*get_local_size(expr):get_global_size(expr))`
`get_num_groups(expr)`	`(expr == splitdim?numgroups:get_num_groups(expr))`
`get_group_id(expr)`	`(get_global_id(expr)/get_local_size(expr))`

All analyses and transformations are performed once at compile-time within the LLVM compiler [6]. The granularities are determined later at runtime.

4.2 Kernel Instantiation and Communication Generation

OpenCL function calls from the host are intercepted by our runtime library `Libsplit`. When the kernel is called in the OpenCL code, the values of the scalar parameters and the size of the `NDRange` are known. The array regions can then be evaluated and the range of work-item ids is determined according to the chosen granularity and the size of work-groups. The runtime kernel instantiation evaluates the dimension of the `NDRange` that allows to distribute written data among devices, with no need for a merge operation, if possible.

Array regions are defined as union of guarded intervals. In order to reduce the number of communications, we evaluate for a given interval of ids and for each array an interval including the array region.

Figure 2 shows the different steps, assuming a copy is performed with a `WriteBuffer` command. Calls to `WriteBuffer` are deferred communications:

Our library registers the commands, write protects the buffers to prevent any modification until the kernel execution. If one buffer is modified before kernel execution, the modification access is trapped, so that the copies occur first and then the modification occurs. If the buffers are not modified, this additional copy is not done. When the kernel is called, the granularity for each device is computed and the devices execute a subkernel, with its associated data. The runtime analysis keeps information related to data distribution. When multiple kernel executions are performed, communications are only performed for data not already present on the device.

Limits of the Analysis. The previous analysis is not always able to precisely compute the regions accessed by a work-item, in particular when indirections occur or when the region accessed depends on control flow too complex for the analysis. When this happens, the array is not split between devices. If the array is written, a merge operation is required after the kernel execution in order to fuse the different contributions computed by each device. The operation we propose is based on a diff, similarly to [12]. Such operation degrades the overall performance for communication/memory bound kernels.

5 Adapting Granularity

This section proposes a method to dynamically adapt granularity to the devices and to the kernel, assuming the same kernel is executed multiple times.

5.1 Formalization

Given a kernel and n devices, the problem consists in determining how to split the computation among the devices so as to minimize the execution time. Each device executes the same kernel, but possibly with a different number of work-groups and different data. We formally define the granularity as a value x_i in $[0, 1]$ corresponding to the ratio between the number of work-groups allocated to the device i and the total number of work-groups (`numgroups`). `numgroups` is known when the kernel is called. We define $f_i(x_i, \text{offset}_i, t)$ as the mean time to execute one work-group on device i, when a subkernel of granularity x_i is executed at time step t, with an offset offset_i. The total execution time of this subkernel is therefore $f_i(x_i, \text{offset}_i, t) * x_i * \texttt{numgroups}$.

The solution to the problem consists in finding the time T and the granularities x_i and the offsets offset_i such that the system in Fig. 3a is fulfilled.

The functions f_i are not known precisely but they can be measured for a given x_i, offset_i and t. We arbitrarily order the offsets by increasing id of device. Thus, with offsets defined by values in $[0, 1]$, $\text{offset}_1 = 0, \ldots, \text{offset}_n = \sum_{k<n} x_k$ and f_i no longer depends on offsets in Fig. 3b but on x_i.

We generalize this formulation by introducing a new set of variables, $y_i \in [0, 1]$, as shown in Fig. 3b. Now it is possible to define a function F such that

$$\begin{array}{ll} \min T & \\ f_1(x_1, \text{offset}_1, t) * x_1 * \texttt{numgroups} & \leq T \\ \dots & \\ f_n(x_n, \text{offset}_n, t) * x_n \texttt{numgroups} & \leq T \\ \sum_i x_i = 1 & \end{array}$$

(a) Initial formulation

$$\begin{array}{ll} \min T & \\ f_1(x_1, t) * y_1 * \texttt{numgroups} & \leq T \\ \dots & \\ f_n(x_1 \dots, x_n, t) * y_n * \texttt{numgroups} & \leq T \\ \sum_i x_i = 1, & \sum_i y_i = 1 \end{array}$$

(b) Generalized formulation

Fig. 3. Formulations of the granularity problem

$F_t(\mathbf{x}) = (\mathbf{y})$, with $\mathbf{x} = (x_i)_i$ the vector of all x_i and $\mathbf{y} = (y_i)_i$ the vector of all y_i, satisfying conditions from Fig. 3b:

$$F_t(\mathbf{x}) = \left(\frac{T}{f_i(x_1, \dots, x_i, t) * \texttt{numgroups}} \right)_i ,$$

with $T = \frac{\texttt{numgroups}}{\sum_i 1/f_i(x_1,\dots,x_i,t)}$. The evaluation of $F_t(\mathbf{x})$ requires $O(n)$ basic arithmetic operations with n the number of devices. A solution to the problem of Fig. 3a can be found by computing a fixed point of the function F: $F_t(\mathbf{x}) = \mathbf{x}$ or similarly, by finding the 0 of the function G_t: $G_t(\mathbf{x}) = \mathbf{x} - F_t(\mathbf{x})$.

5.2 Resolution Method

First assume the function F_t does not depend on t, the iteration count. Several methods have been proposed in the literature for solving such problem, when the function is not known analytically: The fixed point method consists in computing the suite of granularity vectors $\mathbf{x}_k = F(\mathbf{x}_{k-1}), k \geq 1$ from some initial value $\mathbf{x}_0$. The evaluation of $F(\mathbf{x}_{k-1})$ requires to execute the kernel with the granularities $\mathbf{x}_{k-1}$. When the suite converges, it converges linearly towards a vector of optimal granularities satisfying the initial problem and achieving perfect load balance. The convergence depends on F and on the initial value $\mathbf{x}_0$ but is in general linear. The secant method, or its generalization for n-D space the Broyden's method, uses an approximate gradient to converge to the 0 of a function with a near quadratic convergence rate. We implemented both methods to refine the granularity assigned to each device. F is evaluated at each kernel instantiation and provides the input necessary to instantiate the partition-ready kernel.

Finally, when the functions f_i also depend on the iteration count t, the fixed point equation becomes $F_{t-1}(\mathbf{x}_{k-1,t-1}) = \mathbf{x}_{k,t}$ with F changing for each term of the suite. For real applications, a good approximation of the solution at step t remains a good approximation at step $t+1$. As the fixed point method converges quickly when the approximation is close to the solution, we believe this approach can be used for many real cases. We demonstrate for a N-Body application that the fixed point method is able to stay close to the optimal, even when the optimal granularity is varying with the iteration count (see Sect. 7).

6 Related Works

Several works focus on kernel splitting. Kim *et al.* [3] are making one OpenCL device unifying multiple uniform GPUs. The OpenCL `NDRange` and array regions are split according to the values recorded by sampling. They do not handle conditionals as we do and assume that array indices are all linear in the kernel parameters. Moreover, they assume subkernels have the same load. Luk *et al.* [10] propose an heterogeneous programming system that provides an adaptive mapping technique based on execution-time projections stored in a database during training runs. The technique we propose does not require training runs. Li *et al.* [9] present STEPOCL, a tool which takes as input kernels along with a configuration file and generates automatically an OpenCL multi-devices application. The configuration file describes how to split data, the control flow of the program, and allow to have specialized kernels for different architectures. The work partitioning between devices is based on offline profiling. Grewe and O'Boyle [2] propose a pure static task partitioning method, based on predictive modeling and program features. They do not collect data dynamically however and cannot adapt to differences in terms of computing efficiency, depending on granularity, as shown in the motivating example. Kim *et al.* [4] propose to solve the load imbalance problem on CPUs by dynamically assigning sets of work-groups with decreasing sizes to an idle compute unit thread. It manages data across different devices in a cluster, however mapping tasks and data to these devices is left to the programmer. Seo *et al.* [14] propose an automatic work-group size selection technique for OpenCL kernels on multicore CPUs. Their method uses a profiling-based algorithm. Heterogeneity is not handled however and kernels are assumed to be work-group size independent. Kofler *et al.* [5] present a method for OpenCL task partitioning relying on offline generated model. This model is based on artificial neural networks, relying on the features of the kernels, including their input sizes. In [1], the authors propose a dynamic method to partition OpenCL tasks and perform load balancing. Their approach generates chunks of work-groups with increasing size to execute on different devices and selects the best partition of these chunks on the devices. The chunks are manually generated and there is no automatic scheme to partition data for the OpenCL kernels.

In [12], the authors propose an OpenCL runtime that takes a single device kernel and executes it on CPU and GPU. Load balancing is managed at runtime. While their approach dynamically balance work between one CPU and one GPU, it cannot be easily generalized for any number of devices. Besides, data is not split between subkernels, all arrays are transferred to all devices. Finally, the kernel transformation is achieved by hand, and not with an automatic compiler optimization. In [15], Shen *et al.* present a method for heterogeneous platforms and imbalanced applications. They propose a model integrating both the workload of the application, determined by sampling, and the architecture. When the workload has an irregular shape, it is reshaped by sorting to obtain a regular shape with a peak and a bottom part. Based on this model and after profiling, a predictor determines the optimal partitioning between CPUs and GPUs. The work described in [8] relies on complex training, resorting to linear regression techniques in order to predict the correct load balance. They do not handle dynamic load changes, such as the SpMV or OTOO case shown in the motivating example. In [7], the authors extend the previous work to complete task graphs.

It is still based on offline training and assumes performance per work-group is constant, while we have shown there can be large variations.

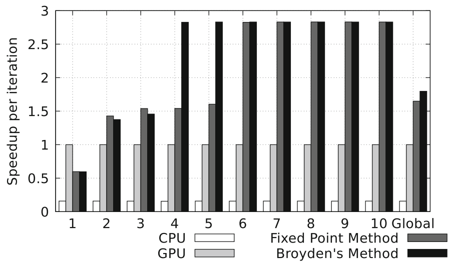
7 Performance Evaluation

Experiments are conducted on two platforms: `conan` — 16-core Intel Xeon E5-2650 2.00 GHz with 64 GB, 3 Nvidia Tesla M2075; `happyCL` — 12-core Intel Xeon E5-2680 2.80 GHz with 64 GB, Nvidia Tesla K20c, Nvidia Quadro K5000.

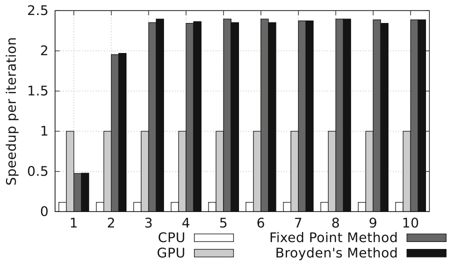
Detailed Load Balancing: Figure 4 shows the speed-up obtained on `conan` with AESEncrypt and EP compared to the best single device performance, when these kernels are repeated 10 times. The speed-up shown here are per iteration. For the first iteration, the granularity is the same for all devices (uniform hypothesis), explaining poor performance compared to the GPU performance. For EP, Fig. 4a, the fixed point method requires 6 iterations to reach a maximum speed up of 2.8, whereas the Broyden's method converges in only 4 iterations leading to a better global speedup. For AESEncrypt, Fig. 4b shows there is no such difference and both methods are similar, reaching a peak speed-up of 2.15 in 3 iterations only. The variations are due to the fact that the optimal granularity does not correspond to a round number of work-groups, hence there are granularity adjustments and communications at each step.

Figure 5 shows how our method behaves when the load changes over 60 iterations. Figure 5a illustrates the time taken by each subkernel for OTOO when the granularity is the same for all devices (`Uniform` strategy). From one device to the other, the execution time differs by more than a factor 3 (iteration 15 for instance). Figure 5b shows how the same load is shared among the four devices when it is continuously adapted by our technique (`Adaptive` strategy). As the 4 plots are close to each other, this shows the execution time is nearly optimal. We observe that convergence to the optimal only requires 2 iterations.

Overall Speedups: Figure 6 presents speed-ups compared to the best single device performance on the two target architectures, for a large number of benchmarks when they are repeated 100 times. We observe the results of our method



(a) EP



(b) AESEncrypt

Fig. 4. Speedup per iteration of EP and AESEncrypt

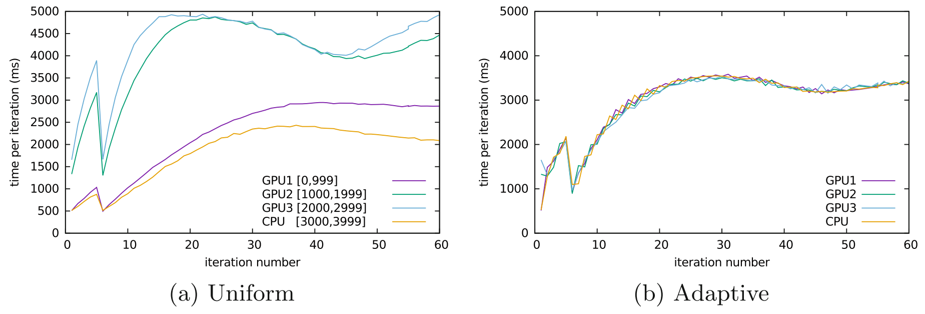


Fig. 5. Performance of OTOO executed on conan (3GPUs+CPU) for 60 iterations. (a) shows the execution time of each subkernel with the Uniform splitting, same granularity for all devices. (b) shows the execution time of each subkernel with the Adaptive splitting. (Color figure online)

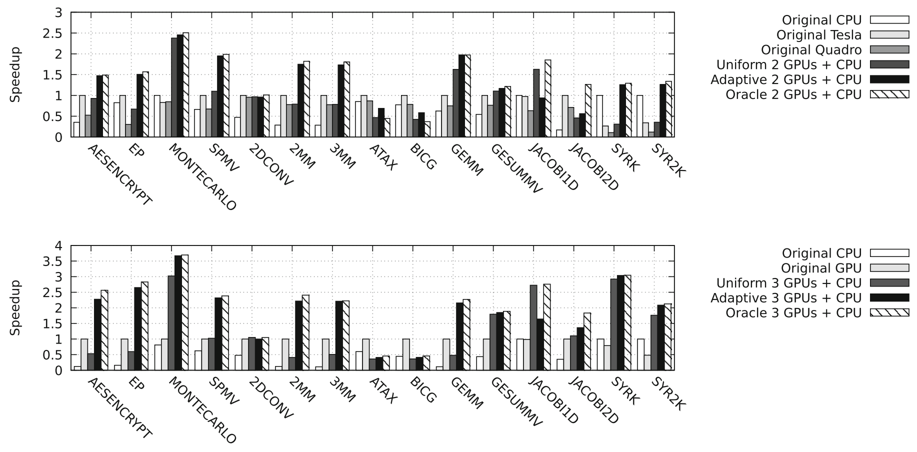


Fig. 6. Performance of AESEncrypt, EP, MonteCarlo, OTOO, SPMV and some Polybench on happyCL (top) and conan (bottom). Original codes run only on one device. Uniform and Adaptive are using subkernels automatically obtained by our method.

(`Adaptive`) are close to the optimal, obtained when launching the kernel directly with the granularity obtained after convergence (`Oracle`). For Jacobi1D and Jacobi2D, the gap is more important because these benchmarks consist in 2 kernels, one stencil and one copy. Defining the same granularity for both copy and stencil minimizes communication time. Our method handles only one kernel at a time, and does not find the same granularity for both kernels.

8 Conclusion

We proposed in this paper the design and implementation of a method that simplifies the development of OpenCL applications for heterogeneous, multi-device systems. Our technique splits computation and data automatically across the computing devices, handling all load-balancing issues, including load variations when the kernel is executed iteratively. We have shown the optimal granularity is obtained in a few iterations and the technique does not require profiling or training. The approach is completely transparent to the user, and the same code can be executed without modification on different machines.

References

1. Boyer, M., Skadron, K., Che, S., Jayasena, N.: Load balancing in a changing world: dealing with heterogeneity and performance variability. In: Computing Frontiers Conference (2013)
2. 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)
3. Kim, J., Kim, H., Lee, J.H., Lee, J.: Achieving a single compute device image in OpenCL for multiple GPUs. In: Principles and Practice of Parallel Programming, PPopp 2011, pp. 277–288. ACM, New York (2011)
4. Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., Lee, J.: SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In: ACM International Conference on Supercomputing, ICS 2012, pp. 341–352. ACM, New York (2012)
5. Kofler, K., Grasso, I., Cosenza, B., Fahringer, T.: An automatic input-sensitive approach for heterogeneous task partitioning. In: International Conference on Supercomputing, pp. 149–160. ACM, New York (2013)
6. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis and transformation, San Jose, CA, USA, pp. 75–88, March 2004
7. Lee, J., Samadi, M., Mahlke, S.: Orchestrating multiple data-parallel kernels on multiple devices. In: Parallel Architecture and Compilation Techniques. IEEE (2015)
8. Lee, J., Samadi, M., Park, Y., Mahlke, S.: SKMD: single kernel on multiple devices for transparent CPU-GPU collaboration. ACM Trans. Comput. Syst. **33**(3), 9:1–9:27 (2015)
9. Li, P., Brunet, E., Trahay, F., Parrot, C., Thomas, G., Namyst, R.: Automatic OpenCL code generation for multi-device heterogeneous architectures. In: International Conference on Parallel Processing, pp. 959–968 (2015)
10. Luk, C.K., Hong, S., Kim, H.: Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Symposium on Microarchitecture, MICRO 42, pp. 45–55. ACM, New York (2009)
11. Nakasato, N., Ogiya, G., Miki, Y., Mori, M., Nomoto, K.: Astrophysical Particle Simulations on Heterogeneous CPU-GPU Systems. CoRR abs/1206.1199 (2012)
12. Pandit, P., Govindarajan, R.: Fluidic kernels: cooperative execution of OpenCL programs on multiple heterogeneous devices. In: Code Generation and Optimization, pp. 273–283. ACM (2014)
13. Seo, S., Jo, G., Lee, J.: Performance characterization of the NAS parallel benchmarks in OpenCL. In: Workload Characterization, pp. 137–148 (2011)

14. Seo, S., Lee, J., Jo, G., Lee, J.: Automatic OpenCL work-group size selection for multicore CPUs. In: Parallel Architectures and Compilation Techniques, pp. 387–397 (2013)
15. Shen, J., Varbanescu, A.L., Sips, H., Arntzen, M., Simons, D.G.: Glinda: a framework for accelerating imbalanced applications on heterogeneous platforms. In: Computing Frontiers Conference, p. 14. ACM (2013)