Automated Generation of OpenCL Programs Based on Algebra-Algorithmic Approach

Anatoliy Doroshenko[1], Oleksii Beketov[1], Mykola Bondarenko[2], Olena Yatsenko[1]

[1] Institute of Software Systems of National Academy of Sciences of Ukraine,
Glushkov prosp. 40, 03187 Kyiv, Ukraine
doroshenkoanatoliy2@gmail.com, beketov.oleksii@gmail.com,
oayat@ukr.net
[2] Taras Shevchenko National University of Kyiv,
Glushkov prosp. 4d, 03680 Kyiv, Ukraine
bondarenko_mykola@yahoo.com.ua

Abstract. The paper proposes the further development of algebra-algorithmic design and synthesis tools towards the development of OpenCL programs. The method for semi-automatic parallelization of cyclic operators is proposed. The particular feature of the approach consists in using high-level algebra-algorithmic program specifications (schemes) and rewriting rules technique. The developed tools provide the construction of parallel algorithm schemes by superposition of predefined language constructs of Glushkov's system of algorithmic algebra, which are considered as reusable components. An algorithm scheme is a basis for the generation of corresponding source code in a target programming language. The approach is illustrated with an example of developing an OpenCL interpolation program used in a numerical weather forecasting. The results of the experiment consisting in executing the generated OpenCL program on a graphics processing unit are given.

Keywords. Algorithmic algebra, automated algorithm design, CUDA, heterogeneous platform, OpenCL, parallel computation, software synthesis.

1 Introduction

Further progress in improving the quality of parallel software development is associated with using heterogeneous architectures of parallel computing systems. One of the facilities for programming heterogeneous parallel systems is OpenCL (Open Computing Language) [1], a framework for developing parallel software that executes across platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators. Unlike Nvidia CUDA [2], which implements GPGPU (General-Purpose computing on GPU) only for Nvidia accelerators and requires the use of a specific compiler, OpenCL is the specification supported by various hardware developers and only requires to set the path to the OpenCL library at compilation. It should be noted that programming heterogeneous platforms is a

complex task and therefore there is a necessity of developing the tools for automated software design and parallelization of existing sequential programs for such platforms.

In the previous works, we had been developing a theory, methodology and tools for automated program design, based on the algebra of algorithmics [3, 4]. The algorithmics formalizes the knowledge about subject domains with the help of algebraic facilities and deals with problems of formalization, substantiation of correctness and transformation of algorithms. The formal facilities for development of parallel programs for multicore CPUs [4] and Nvidia GPUs (using CUDA) [3] were developed. They were based on Glushkov's system of algorithmic algebra (SAA) [3, 4] and term rewriting technique [4]. On the basis of the developed theory and methodology, the integrated toolkit for designing and synthesis of programs (IDS) was developed [3, 4].

In this paper, we propose the further development of our algebra-algorithmic methodology and tools in the direction of formalized and automated design of OpenCL programs. A method and a software tool intended for semi-automatic parallelization of cyclic operators are described. The approach is illustrated with an example of designing a parallel interpolation algorithm, which is the part of the numerical weather forecasting program. The results of the experiment consisting in executing the generated OpenCL program on a GPU are given.

2 Algebra-Algorithmic Software Design Tools

The approach to designing parallel programs being proposed is based on Glushkov's system of algorithmic algebra [3, 4], intended for formal representation of algorithmic knowledge in the form of high-level specifications. SAA is the two-sorted algebra $GA = <\{Pr, Op\}; \Omega_{GA}>$, where Pr and Op are the sets of logic conditions (predicates) and operators defined on an information set; Ω_{GA} is the signature consisting of logic operations (disjunction, conjunction, negation) and operator constructs, in particular:

- serial execution of operators: "*operator* 1"; "*operator* 2" ;
- branching: IF '*condition*' THEN "*operator* 1" ELSE "*operator* 2" END IF ;
- for loop: FOR (*counter* FROM *start* TO *fin*) "*operator*" END OF LOOP ;
- asynchronous execution of n operators: PARALLEL($i = 0,...,n-1$)("*operator i*") ;
- synchronizer, which delays the computation until the value of the specified condition is true: WAIT '*condition*' .

The algorithms, represented in SAA, are called SAA schemes. Automated design of algorithm schemes and generation of corresponding programs is provided by the developed IDS toolkit [3, 4]. The design process is represented by a tree of an algorithm. The user chooses SAA constructs from the list and adds them to an algorithm tree. On each step of the design process, the system allows a user to select only those operations, the insertion of which into a scheme does not break its syntactical correctness. The algorithm tree is then used for the automatic generation of SAA scheme text

and programming code in one of the target languages (C, C++, Java). The mapping of SAA operators to text in a programming language is defined in a form of code templates in the database of IDS. In [3], additional facilities for designing GPU programs using the CUDA platform were developed.

In this work, we introduce new SAA operations intended for high-level design of OpenCL programs and propose the method for parallelization of nested loops in sequential programs.

OpenCL [1] combines the application programming interface (API) and a variant of C language for programming and simultaneous use of various parallel computing devices (for example, CPU, GPU, and Xeon Phi) in a heterogeneous environment. The main steps of developing a common OpenCL program and corresponding new basic operators (enclosed in quotation marks) added to SAA are the following.

1. Obtaining the list of available platforms and saving it to a variable *plms*: "Get all available platforms (*plms*)".
2. Obtaining the list of devices *dvs* of the platform *plm*: "Get all devices (*dvs*) available on a platform (*plm*)".
3. Creating the computing environment for the devices: "Create a context (*cnt*) for devices (*dvs*)".
4. Creating the queue of commands for the device: "Create a command queue (*cmdqueue*) for a context (*cnt*) and a device (*dv*)".
5. Designing a kernel function — a task to execute on the devices. The example of an SAA scheme for a kernel is given in Sect. 3.
6. Compilation of a file containing an OpenCL kernel: "Create a kernel (*krnl*) from a source (*file*)", where *krnl* is the name of a kernel function, *file* is the path to the file with its source code.
7. Creating the buffer for data specified in a variable *var*: "Create a memory buffer (*buff*) for data (*var*) on devices in the context (*cnt*)".
8. Copying a buffer for a variable *var* to device memory: "Add commands to queue (*cmdqueue*) writing a buffer of data (*var*) from host to device".
9. Setting the value *arg_value* of the parameter with the index *arg_index* for a kernel: "Set the argument value (*arg_value*) for a parameter (*arg_index*) of a kernel (*krnl*)".
10. Adding a kernel to a command queue and its asynchronous execution: "Add a command to queue (*cmdqueue*) executing a kernel (*krnl*) (*globalworksize*) (*localworksize*) on a device", where *globalworksize* is the global number of dimensions [1] for executing the kernel; *localworksize* is the number of dimensions of a local subset of kernel instances (work-items) in a group.
11. Reading the data from a result buffer: "Add commands to queue (*cmdqueue*) reading from a buffer of data (*var*) from device to host".

The use of the above constructs is illustrated in Sect. 3.

In most computing problems, a large part of hardware resources are utilized by computations inside loops, therefore the use of automatic parallelization of the cyclic operators is most efficient for them. Further, we describe the developed method and software framework named LoopRipper intended for semi-automatic parallelization of programs containing loop operators for heterogeneous platforms. The framework is

based on the combined use of the rewriting rules system TermWare [4] and IDS toolkit. LoopRipper parallelizes the input compound loop of the following form:

$$
\begin{aligned}
&\text{“SEQUENTIAL LOOP”}\\
&\text{==== FOR } (i_0 \text{ FROM 0 TO } \#I_0)\\
&\qquad \text{FOR } (i_1 \text{ FROM 0 TO } \#I_1)\\
&\qquad\quad \ldots\\
&\qquad\quad \text{FOR } (i_n \text{ FROM 0 TO } \#I_N)\\
&\qquad\qquad \text{“}F(\bar{i}, \overline{p^{in}(\bar{i})}, \overline{p^{out}(\bar{i})})\text{”}\\
&\qquad \text{END OF LOOP,}
\end{aligned}
\tag{1}
$$

where $I_0, I_1, ..., I_N$ are the sets values of indices $i_0, i_1, ..., i_N$; $N+1$ is the loop nesting depth; $\bar{i} = \{i_j \mid j = 0,..., N\}$; $\overline{p^*(\bar{i})} = P^* = \{p_j^*(\bar{i}) \mid j = 0...\#P^*\}, * \in \{in, out\}$ are the sets of input and output data variables; F is the function processing the data; the iterations of the loop are independent.

Let T be the number of threads which will be used in the execution of a kernel function. The loop obtained as a result of the parallelization is the following:

```
“PARALLEL LOOP”
==== FOR (e FROM 0 TO L)
       “fill(inBuf)”;
       “push(inBuf)”;
       “kernel(e, inBuf, outBuf)”;
       “pull(outBuf)”;
       “unpack(outBuf)”
     END OF LOOP,
```

where L is the number of kernel calls which is chosen the least possible so that $L \cdot T \geq G$, G is the total number of initial loop iterations; *fill* is the function filling the buffer of initial data *inBuf*; *push* is moving the initial data buffer to the device memory; *kernel* is the call of a kernel function; the kernel function contains the call of the initial loop body “$F(\bar{i}, \overline{inbuf_{id}(\bar{i})}, \overline{outbuf_{id}(\bar{i})})$”, where *id* is the thread number; *pull* is the function moving the processed data from the device memory to the buffer of processed data *outBuf*; *unpack* is copying data from the buffer of processed data to corresponding variables.

The user of the LoopRipper framework (see Fig. 1) provides a source code or an SAA scheme of the sequential program, specifies the loop to be parallelized and also provides the list of input and output variables used in the loop. With the help of the TermWare system, the framework generates *kernel*, *fill* and *unpack* functions and additional data structures (buffers). IDS toolkit replaces the sequential loop in the

sequential program with corresponding kernel call and synthesizes the whole parallel program from the above-mentioned functions and data structures.

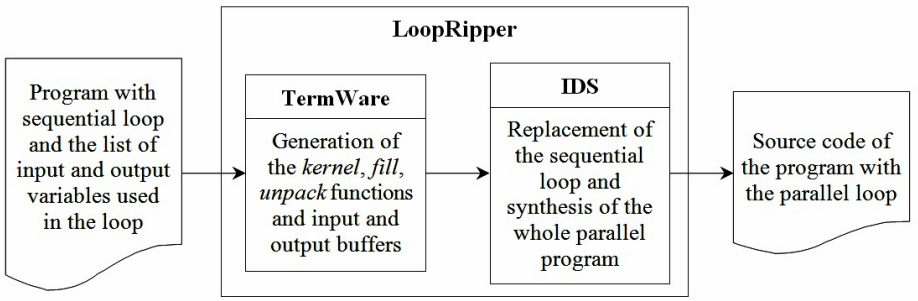


Fig. 1. Parallelization of a program with the LoopRipper framework.

3 Case Study

In this section, we illustrate the use of the considered algebra-algorithmic tools with an example of developing an OpenCL interpolation program used in numerical weather forecasting [5]. The program performs the interpolation of meteorological values defined on a macroscale grid to a mesoscale grid.

The initial sequential interpolation scheme consists of four nested loops of the form (1) with iterators $h \in [0 \dots Pk]$, $k \in [0 \dots Lmz]$, $j \in [0 \dots Mmz]$, $i \in [0 \dots Nmz]$, where Pk, Lmz, Mmz, Nmz are input integer parameters defining the number of points in finite-difference grid along altitude, longitude and latitude. The iterations of the loops are independent and can be executed in parallel. The SAA scheme of the OpenCL kernel obtained as a result of the transformation of the sequential interpolation algorithm with the help of the LoopRipper framework is given below. The computation is parallelized by indices h, k, j; the main loop of interpolation iterates through the index i. In the scheme, US, VS, TS, HS, QS are input arrays with meteorological values (wind velocity, temperature, humidity, etc.); Qc is the output array. The index for processing array elements is computed and stored in the variable ind.

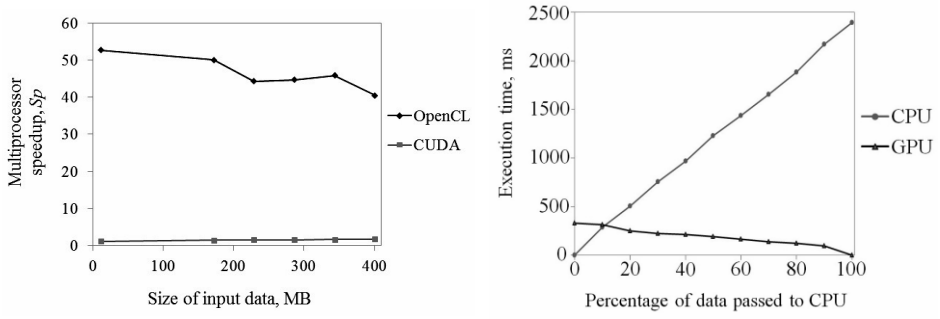
```
"interpolation_kernel(US, VS, HS, QS, TS, F_x, Zmz, Qc, Pk, Lmz, Mmz, Nmz)"
==== (h := "Get the global work-item identifier for dimension (0)");
     (k := "Get the global work-item identifier for dimension (1)");
     (j := "Get the global work-item identifier for dimension (2)");
     IF NOT((h >= Pk) OR (k >= Lmz) OR (j >= Mmz))
     THEN
       FOR (i FROM 0 TO Nmz)
       LOOP
         "(ind) := (h + Pk * k + Pk * Lmz * j + Pk * Lmz * Mmz * i)";
         "(a) := ((WZZ + US[ind] / 0.321) * Rs * VS[ind])";
```

"(*Tp*) := (*TS*[*ind*] * pow(1000.0 / *HS*[*ind*], 2.0 / 7.0))";
"(*Tv*) := (*Tp* * (1.0 + 0.6078 * *QS*[*ind*]))";
"(*Qc*[*ind*]) := (*a* – (0.5**Tv* + (1.0 – *Zmz*[*k*]) * *g* * *F_x*[*j* + *I***Mmz*]/0.321))"
END OF LOOP
END IF

The SAA scheme of the host code of the interpolation program uses the operators described in Sect. 2 and is the following:

"interpolation_host"
==== "Read input parameters (*Pk*, *Lmz*, *Mmz*, *Nmz*) from the command line";
"Get all available platforms (*plms*)";
"Get all devices (*dvs*) available on a platform (*plms*[0])";
"Create a context (*cnt*) for devices (*dvs*)";
"Create a command queue (*cmdqueue*) for a context (*cnt*)
and a device (*dvs*[0])";
"Create a kernel ("interpolation") from a source ("kernels/interpolation.cl")";
"Declare and fill continuous arrays for data (*US*,*VS*,*HS*,*QS*,*TS*,*Qc*,*F_X*,*Zmz*)";
"Create memory buffers for data (*US*, *VS*, *HS*, *QS*, *TS*, *Qc*, *F_X*, *Zmz*) on
devices in the context (*cnt*)";
"Add commands to queue (*cmdqueue*) writing buffers of data
(*US*, *VS*, *HS*, *QS*, *TS*, *F_X*, *Zmz*) from host to device";
"Set the argument values for parameters of a kernel (*krnl*)";
"Add a command to queue (*cmdqueue*) executing a kernel
(*krnl*) (3) ({*Pk*, *Lmz*, *Mmz*}) on a device";
WAIT 'All commands in (*cmdqueue*) were issued to device and completed';
"Add commands to queue (*cmdqueue*) reading from a buffer of data (*Qc*)
from device to host";

IDS toolkit automatically translated the above SAA schemes into OpenCL code. The obtained program was executed in two computing environments. The first environment consisted of Intel Core i5-4210U CPU (2 cores) and Nvidia GeForce 840M GPU (384 cores), the second one contained Intel Core i3-3110M CPU (2 cores) with integrated Intel HD Graphics 4000 GPU (128 cores). Fig. 2(a) shows the multiprocessor speedup $Sp = T_s / T_p$ obtained in the first environment, where T_s is the execution time of the sequential program on CPU, T_p is the execution time of the parallel program (OpenCL and its previous version implemented in CUDA [6]) on GPU. The previous version used multi-dimensional arrays for storing meteorological data instead of one-dimensional. As can be seen, the OpenCL program significantly outperforms the CUDA version, which is explained by the use of array linearization. Fig. 2(b) shows the dependency of CPU and GPU execution time on the percentage of data passed to CPU in the second environment. As can be seen from the diagram, the execution of all computations on CPU is approximately 7 times longer than execution on GPU only. However, if computations are performed simultaneously on CPU and GPU, the optimization by about 7–10% is achieved.



(a) (b)

Fig. 2. The results of executing the parallel interpolation program: the dependency of the multiprocessor speedup on the size of the input data (a) and the dependency of execution time on the percentage of data passed to CPU (b).

4 Related Work

The proposed approach is related to works on the synthesis of programs from specifications [7] and automated generation of OpenCL programs [8–12]. In particular, paper [8] presents an approach and a tool named Gaspard2 based on model-driven engineering to specify, design, and generate OpenCL applications. In [9], a programming tool called STEPOCL is proposed along with a new domain-specific language designed to simplify the development of an application for multiple accelerators. Paper [10] presents an automatic generator based on a C++ domain-specific embedded language for the generation of OpenCL kernels defined by high-level specifications provided by the user. In [11], an approach to the generation of OpenCL code based on high-level functional expressions and rewriting rules is proposed. Par4All [12] is an automatic parallelizing and optimizing compiler for C and Fortran which generates OpenMP, CUDA and OpenCL source codes.

The main difference of our approach from the mentioned related works is that it uses algebraic specifications, based on Glushkov algebra of algorithms. Specifications are represented in a natural linguistic form simplifying understanding of algorithms and facilitating achievement of demanded software quality. Another advantage of our tools is the method of automated design of syntactically correct algorithm specifications [3, 4], which eliminates syntax errors during construction of algorithm schemes. Unlike the above-mentioned Par4All, our parallelization framework LoopRipper allows processing the amounts of data which exceed the GPU memory size and using heterogeneous computing clusters.

5 Conclusion

This paper proposes an approach and tools for automated designing and generation of OpenCL programs based on algorithmic algebra. The method for semi-automatic

parallelization of cyclic operators is developed. The particular feature of the approach consists in using high-level algebra-algorithmic program specifications, which are represented in a natural linguistic form. The developed tools provide the construction of algorithm schemes by superposition of predefined language constructs of Glushkov's algebra, which are considered as reusable components. The developed tools automatically translate the specifications into source code in a programming language. The approach is illustrated on developing the OpenCL interpolation program used in a numerical weather forecasting.

References

1. OpenCL Overview. The open standard for parallel programming of heterogeneous systems, https://www.khronos.org/opencl, last accessed 2019/02/15.
2. Nvidia CUDA technology, http://www.nvidia.com/cuda, last accessed 2019/02/15.
3. Andon, P.I., Doroshenko, A.Yu., Zhereb, K.A., Yatsenko, O.A.: Algebra-algorithmic models and methods of parallel programming. Akademperiodyka, Kyiv (2018)
4. Doroshenko, A., Zhereb, K., Yatsenko, O.: Developing and optimizing parallel programs with algebra-algorithmic and term rewriting tools. In: Ermolayev, V., Mayr, H.C., Nikitchenko, M., Spivakovsky, A., Zholtkevych, G. (eds.) ICTERI 2013. CCIS, vol. 412, pp. 70–92. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03998-5_5
5. Prusov, V., Doroshenko, A.: Computational techniques for modeling atmospheric processes. IGI Global, Hershey (2018). https://doi.org/10.4018/978-1-5225-2636-0
6. Doroshenko, A.Yu., Yatsenko, O.A., Beketov, O.G.: Algorithm for automatic loop parallelization for graphics processing units. Problems in programming, (4), 28–36 (2017) (in Ukrainian)
7. Gulwani, S.: Dimensions in program synthesis. In: 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming, pp. 13–24. ACM, New York (2010)
8. Rodrigues, A., Guyomarc'h, F., Dekeyser, J.L.: An MDE approach for automatic code generation from UML/MARTE to OpenCL. Computing in Science and Engineering, 15(1), 46–55 (2012)
9. Li, P., Brunet, E., Trahay, F., Parrot, C., Thomas, G., Namyst, R.: Automatic OpenCL code generation for multi-device heterogeneous architectures. In: 44th International Conference on Parallel Processing (ICPP 2015), pp. 959–968. IEEE, Piscataway, NJ (2015)
10. Tillet, P., Rupp, K., Selberherr, S.: An automatic OpenCL compute kernel generator for basic linear algebra operations. In: 2012 Symposium on High Performance Computing (HPC'12), pp. 4:1–4:2. Society for Computer Simulation International, San Diego, CA (2012)
11. 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. In: 20th ACM SIGPLAN International Conference on Functional Programming (ICFP'15), ACM SIGPLAN Notices, vol. 50, pp. 205–217. ACM, New York (2015)
12. PIPS: Automatic Parallelizer and Code Transformation Framework, http://pips4u.org, last accessed 2019/02/15.