DOI https://doi.org/10.24297/ijct.v21i.9004

Code Generation from Simulink Models with Task and Data Parallelism

Pin Xu[1], *Masaki Kondo*[2] *and Masato Edahiro*[1]

[1]Nagoya University
[2]NEC Solution Innovators, Ltd.

eda@ertl.jp

Abstract

In this paper, we propose a method to automatically generate parallelized code from Simulink models, while exploiting both task and data parallelism. Building on previous research, we propose a model-based parallelizer (MBP) that exploits task parallelism and assigns tasks to CPU cores using a hierarchical clustering method. We also propose a method in which data-parallel SYCL code is generated from Simulink models; computations with data parallelism are expressed in the form of S-Function Builder blocks and are executed in a heterogeneous computing environment. Most parts of the procedure can be automated with scripts, and the two methods can be applied together. In the evaluation, the data-parallel programs generated using our proposed method achieved a maximum speedup of approximately 547 times, compared to sequential programs, without observable differences in the computed results. In addition, the programs generated while exploiting both task and data parallelism were confirmed to have achieved better performance than those exploiting either one of the two.

Keywords: Simulink model, code generation, task parallelism, data parallelism

1. Introduction

The Simulink [14] graphical programming environment is widely used in the model-based design of embedded systems. Its multi-core and many-core performance has drawn great attention recently. In addition, calculations exhibiting data parallelism, for example, image processing and neural network inferences, have become increasingly common in embedded systems, to the extent that systems-on-a-chip (SoCs) containing graphics-processing units (GPUs) or other specialized hardware are being created. Thus, there is a great need to automatically generate parallel code from Simulink models.

In this paper, we propose a method to automatically generate parallelized code from Simulink models, while exploiting both task and data parallelism. Based on previous research, we propose a model-based parallelizer (MBP) that exploits task parallelism and assigns tasks to CPU cores, using the hierarchical clustering method described in [22][23]. We also propose a method; data-parallel SYCL [11] code is generated from Simulink models, in which computations with data parallelism are expressed in the form of S-Function Builder blocks and are executed in a heterogeneous computing environment. Most parts of the procedure can be automated with scripts, and the two methods can be applied together.

The contributions of this paper are as follows:

1. We developed a model-based parallelizer for automatically generating task-parallel C++ code, centered around the block-level XML (BLXML) markup language, while implementing a hierarchical clustering method.
2. We proposed a semi-automatic approach for generating data-parallel SYCL code utilizing an existing source-to-

source compiler, while targeting heterogeneous architectures containing data-parallel accelerators.

3. We proposed an approach for semi-automatically generating programs exploiting both task and data parallelism through the combined application of the above two approaches, which we believe is the first proposal of this kind.

Section 2 provides a brief review of the related work and previous studies. Section 3 describes the major tools used in our approach, and Section 4 provides an overview of the proposed method. In Section 5, we evaluate our approach with models that exhibit data parallelism and models that exhibit both task and data parallelism. Section 6 concludes the paper.

2. Previous and related work

BLXML, a markup language specifically designed for efficiently representing block-level information in Simulink models, was proposed by Yamaguchi et al. [21]. It saves information on performance profiles, caller-callee relationships, and core assignments of functions, in addition to block diagrams and source code. Our approach to task-parallel code generation centers around the BLXML file format.

Zhong et al. [22][23] proposed a hierarchical clustering method that facilitates task-parallel code generation. It is a set of algorithms for mapping functions to cores on a heterogeneous platform. It improves the efficiency of previously proposed algorithms; e.g., the critical-path algorithm proposed in [10]. We implemented the hierarchical clustering method as part of our approach to task-parallel code generation.

A large amount of research has been conducted on generating task-parallel code from Simulink models. One popular approach is to alter the Simulink block diagram to reflect the mapping of tasks to cores before generating serial code, and then making relatively minor changes to facilitate multithreading. References [8], [15], and [17] achieved varying levels of automation using such approaches, while targeting homogeneous architectures.

On the other hand, [3] proposed an operation scheme in which code written by a model designer with task parallelism in mind is incorporated into Simulink models. The scheme was implemented in [2], where code that contains compiler directives targeting the RATI Linux parallel-programming library is inserted into hand-written C MEX S-Functions to achieve automated code generation. A source-to-source compiler is then utilized to create pthread calls, based on compiler directives.

More recently, direct, automatic translation to parallelized code has been attempted, such as [4], which translates the code into SystemC, while also targeting heterogeneous architectures, and [12], which translates it into symmetric multiprocessing (SMP) code. However, most of these studies only attempted to exploit task parallelism. Reference [18] aims at exploiting both task parallelism and loop-level (i.e., data) parallelism through the application of OSCAR, a multigrain parallelizing compiler, on the serial code generated by Simulink, while targeting homogeneous architectures.

We believe that our proposed approach is the first attempt at semi-automatic parallel-code generation that exploits both task and data parallelism, while targeting heterogeneous architectures. Our approach incorporates multiple ideas from related works, such as using compiler directives and S-Functions, source-to-source compilers, and extracting data parallelism from *for* loops.

3. Tools

SYCL [11] is an abstraction layer standardized by the Khronos Group, which aims to enable cross-platform heterogeneous parallelized programming with a pure C++ grammar. Intel data-parallel C++ (DPC++) is one of the many implementations of the SYCL standard, developed for Intel CPUs and GPUs, while also providing limited support for

Nvidia GPUs and other platforms. Furthermore, Intel provides the Intel oneAPI Base Toolkit [9], which contains a number of tools centered on the compilation and deployment of Intel DPC++ programs. In this research, the following two tools were utilized to generate data-parallel programs.

1. Intel oneAPI DPC++/C++ Compiler, which is an open-source compiler for Intel DPC++, based on Clang/low-level virtual machine (LLVM).
2. Intel DPC++ Compatibility Tool (Intel DPCT), which converts compute-unified device architecture (CUDA) code to Intel DPC++ code.

Polyhedral parallel-code generation (**PPCG**) [19] is a source-to-source compiler that converts C code, where computations exhibiting data parallelism are expressed in multi-layered *for* loops, to CUDA code. It is based on the polyhedral model [6][20], a well-known optimization model for parallel programming.

4. Proposed method

4.1 Model-based parallelizer (MBP)

Building on previous research, we propose a model-based parallelizer (MBP) that exploits task parallelism and assigns tasks to CPU cores, using the hierarchical clustering method described in [22][23]. The steps by which such parallelization is performed are illustrated in Fig. 1 and are described in the following sections.

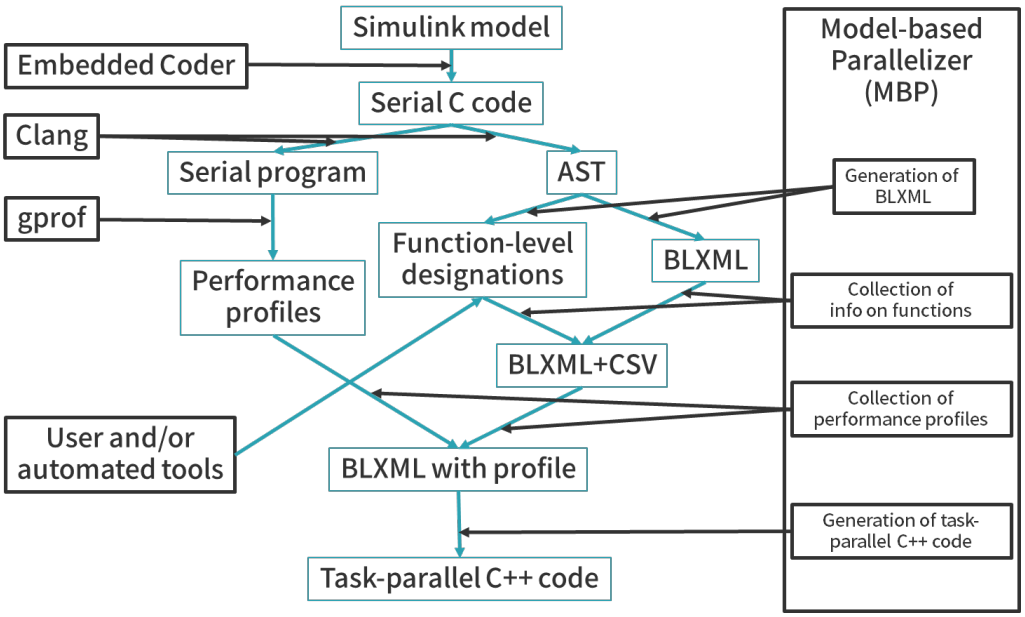


Figure 1: Steps by which MBP performs task parallelization

4.1.1 Generation of BLXML file

As the first step of MBP, serial C code is generated from the Simulink model using Embedded Coder [13], a tool provided as a part of Simulink. The serial C code is further processed by the Clang compiler to generate an abstract syntax tree, expressed in the JSON file format. MBP then generates a BLXML file, based on information from the serial C code and the JSON file.

4.1.2 Collection of information on functions

Some functions must be executed serially; especially, functions that are always generated by Embedded Coder from any Simulink model and functions specifically designated by the user. To address this, MBP generates a CSV file containing the function names, line numbers in the source file where the function is located, and information on such designations.

4.1.3 Collection of performance profiles

MBP collects a flat profile of the functions' performance using the default profiler provided by the operating system: gprof [7] in the GNU/Linux environment. Information on performance is then stored in the BLXML file, as described in Section 4.1.1.

4.1.4 Task-parallel program generation

Finally, MBP generates task-parallel C++ code using the standard thread library, based on information on function-level designations and performance, collected as described in previous sections. The functions were assigned to the CPU cores using the hierarchical clustering method described in [22][23].

4.2 Data-parallel program generation

4.2.1 Construction of Simulink models with data parallelism

Custom Simulink blocks are required to efficiently model arbitrary computations that exhibit data parallelism. The various types of custom Simulink blocks are compared in Table 1. Among the options, we propose the use of the S-Function Builder for the following reasons.

1. S-Function Builder blocks are capable of storing internal states, thereby enabling the modeling of more complicated computations.
2. Code written in S-Function Builder blocks is copied in its entirety into the serial code generated by Embedded Coder, making it possible to indicate the position of data-parallelizable code with either comments or compiler directives.
3. S-Function Builder blocks automatically generate the target-language compiler (TLC) files needed for serial-code generation, reducing the amount of code that must be written by hand.

Specifically, computation with data parallelism is expressed as nested *for* loops. Their locations are indicated by compiler directives specified by the PPCG compiler, whose use is described in the following sections.

Table 1: Comparison of custom Simulink blocks

	S-Function Builder	Level-2 MATLAB S-Function	C MEX S-Function	MATLAB Function	C Caller
Performance	Fast	Fast	Fast	Slow	Fast
Supports internal states	Yes	Yes	Yes	N/A	N/A
Language written in	C/C++	MATLAB	C MEX	MATLAB	C
Generation of TLC	Automatic	Manual	Manual	N/A	N/A

4.2.2 Parallel-programming language

We propose to use SYCL [11] as the code-generation target, and Intel DPC++ [9] to implement SYCL.

The parallel-programming languages that are probable targets of code generation are summarized in Table 2. Among them, SYCL has a great advantage in that it is designed as a general-purpose programming language, and it uses the same syntax for the code of both the host and the target. Moreover, it is a higher-level programming language than some of its counterparts; thus, enabling code portability. Additionally, the facts that it supports more target platforms and is an open-standard language are also desirable properties.

Table 2: Comparison of parallel-programming languages

	SYCL	OpenVX	OpenCL	CUDA
Application type	General-purpose	Image processing	General-purpose	General-purpose
Language based on	C++	C/C++	C/C++	C/C++
Syntax of host and kernel code	Same	Different	Different	Different
Level of abstraction	High	High	Low	Low
Target platforms	Multiple	Multiple	Multiple	Nvidia GPU
Open standard	Yes	Yes	Yes	No

A number of projects have aimed at implementing the SYCL standard. The status of some of these projects at the time of writing are summarized in Table 3 [1][5][9][16]. We propose to adopt Intel DPC++ as the SYCL implementation, for its official release, active maintenance, and greater number of supported platforms.

Table 3: Projects implementing the SYCL standard

	Intel DPC++	TriSYCL	hipSYCL	ComputeCpp
Leading organization	Intel	Xilinx	Heidelberg University	Codeplay
Release	Official	Incomplete	Beta	Official
Most recent release	December 2020	March 2018	December 2020	November 2020
Open source	Yes	Yes	Yes	No
CPU	Any	Any	Any	Any
GPU	Intel/Nvidia	Nvidia	Nvidia	Intel/AMD/Nvidia
FPGA	Intel	Xilinx	N/A	N/A

4.2.3 Generation of SYCL code

Common methods of generating code from C code usually include conversion to lower-level languages, e.g., LLVM-intermediate representation (LLVM-IR), using tools such as Clang/LLVM. However, generating code in high-level languages, e.g., SYCL, from such lower-level languages is difficult. Intel DPCT provides automatic translation from CUDA to SYCL. Therefore, SYCL code could be generated relatively simply, without needing to handle lower-level languages, by using CUDA as an intermediate representation.

In this case, a source-to-source compiler that can compile C code to CUDA code and extract data parallelism is required. However, most such compilers can process only a subset of legitimate C code. Moreover, it is in the best interest of our research, which aims to automate the code-generation process, to reduce possible errors that may occur through using such compilers. In this regard, the compiler to be used should satisfy the following requirements:

1. The subset of legitimate C code it can process shall be as large as possible.
2. It shall be possible to designate code segments to be processed with comments or compiler directives.

We propose to use PPCG as the C-to-CUDA compiler, as it satisfies the requirements to the greatest degree.

In summary, we propose a method consisting of the following steps, in which the SYCL code is generated from Simulink models constructed according to the method described in Section 4.2.1.

1. Generate serial C code from the Simulink model using Embedded Coder [13].
2. Generate CUDA code from the serial C code using PPCG [19].
3. Generate SYCL code from the CUDA code using Intel DPCT [9].

4.3 Combined application of Sections 4.1 and 4.2

We propose a method that combines the parallelizing techniques described in Sections 4.1 and 4.2, as follows. A performance profile is first generated from the data-parallel SYCL program, using the method described in Section 4.2, and executed on the target heterogeneous platform. MBP then performs task parallelization on the original serial C code, while treating the data-parallel accelerator as if it were a type of CPU core. Finally, we substitute the functions containing data-parallel computations in the generated task-parallel C++ code with their respective parallelized functions in the SYCL code.

This approach of combining MBP and the data-parallelization application is possible for the following reasons. First, MBP can be configured to target heterogeneous platforms, making it possible to treat the accelerator as a type of CPU core. Second, the task-parallel C++ code generated by MBP uses the standard thread library, making it possible to substitute all functions that would be executed in separate threads without having to alter the code that calls them. Finally, the code that conforms to the SYCL standard uses valid C++ syntax, enabling direct substitutions without further translation.

4.4 Automating the code generation

The steps by which the SYCL code is generated, as described in Section 4.2.3, are summarized in Fig. 2. Among these steps, the serial C code from the Simulink models must be generated manually using Embedded Coder; however, the other steps, where the CUDA intermediate representation and SYCL code are generated, as well as the compilation of SYCL code into data-parallel programs, were successfully automated.

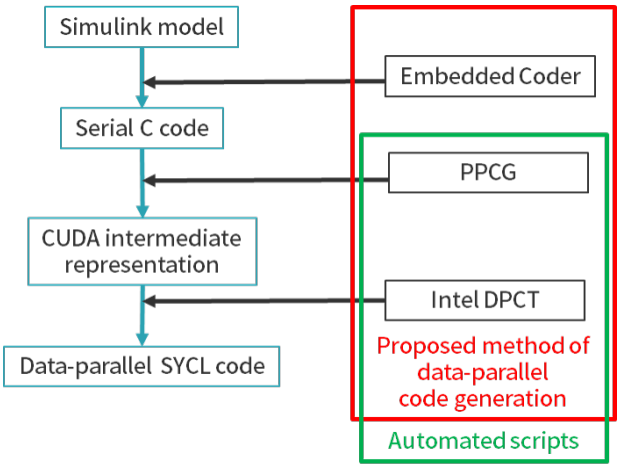


Figure 2: Steps by which SYCL code is generated

Fig. 3 summarizes the steps for generating the code that exploits both task and data parallelism, as described in Section 4.3. Among these steps, the data-parallel program generation contains manual operations, whereas all the other steps are successfully automated.

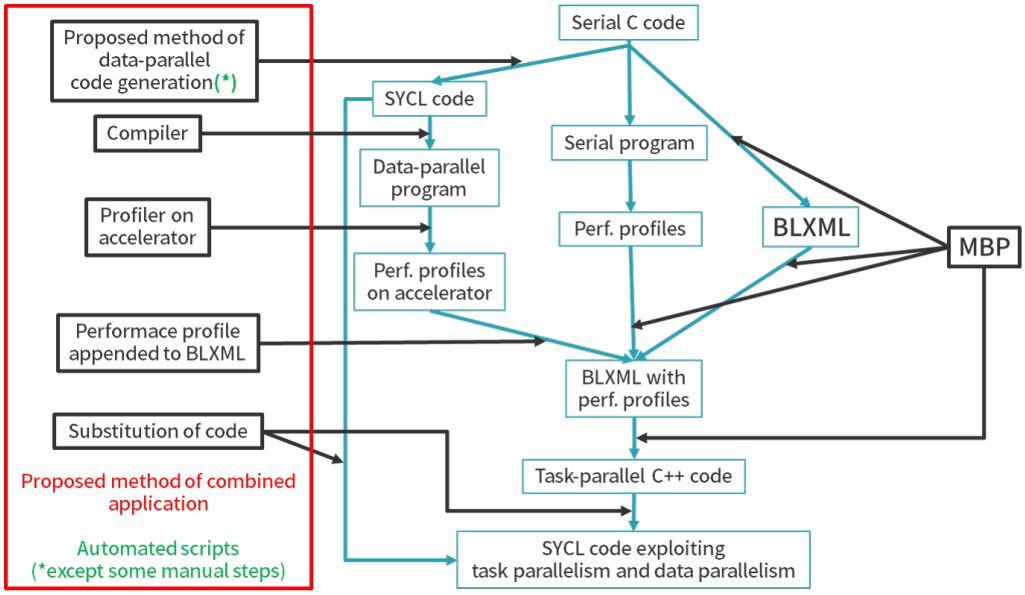


Figure 3: Steps by which the code exploiting both task parallelism and data parallelism is generated

5. Evaluation

This section describes the experiments with our proposed methods to evaluate the correctness of the generated parallel programs and their performance gains over their serial counterparts.

5.1 Methodology

We evaluate the correctness of the generated parallel programs by comparing the computation results from the parallel and serial programs. We also evaluated the performance gain by measuring the time taken to execute both programs 100 times each. The performance gain is defined as the ratio of the time taken by the serial programs to that by the parallel programs.

5.2 Environment

The serial programs are executed in a single-core CPU environment, as described in Table 4. The data-parallel programs are executed in the multicore CPU environment described in Table 5, as well as in the CPU + GPU environment described in Table 6. The programs exploiting both task and data parallelism are executed in the CPU + GPU environment, as described in Table 6.

Table 4: Single-core CPU environment

Item	Content
CPU	Intel Core i9-7900X (3.30 GHz)
RAM	16 GB DDR4 DIMM×8
C standard	C11
C compiler	Clang 12.0.0 (trunk)

Table 5: Multi-core CPU environment

Item	Content
CPU	Intel Core i9-7900X (3.30 GHz) (10 cores, 20 threads)
RAM	16 GB DDR4 DIMM×8
SYCL compiler	Intel oneAPI DPC++ Compiler 2021.1
SYCL backend	Intel OpenCL 2.1

Table 6: CPU + GPU environment

Item	Content
CPU	Intel Core i9-7900X (3.30 GHz)
RAM	16 GB DDR4 DIMM×8
GPU	Nvidia Titan V (1.20 GHz)
SYCL compiler	Intel oneAPI DPC++ Compiler 2021.1
SYCL backend	Nvidia CUDA 10.2.89

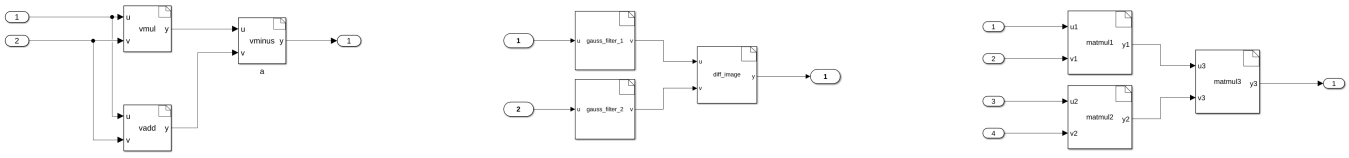
5.3 Evaluation 1: Correctness and performance gains of data-parallel programs

In this evaluation, we compare the data-parallel programs, generated through the method described in Section 4.2, with the serial programs generated by Embedded Coder from a number of Simulink models.

5.3.1 Simulink models used

In this evaluation, we use three types of Simulink models, each designed to carry out one of the following types of calculations that exhibit data parallelism, with adjustable calculation scales.

- Simple vector calculations, with vector lengths between 10,000 and 50,000,000 (Fig. 4a).
- Application of a Gaussian filter on an image, and the difference between two images, with the number of pixels in each image between 76,800 and 64,000,000 (Fig. 4b).
- Matrix multiplications, with matrix sizes between 250×250 and 5000×5000 (Fig. 4c).



(a) Simple vector calculations (b) Gaussian filter and difference (c) Matrix multiplications

Figure 4: Simulink models used in evaluation 1

5.3.2 Results

For all Simulink models, there were no differences between the calculation results from the data-parallel programs and those from the serial programs, indicating the correctness of the data-parallel programs generated according to our proposed approach, within the limits of this evaluation.

For each type of Simulink model, the performance gains through data-parallelization, as measured in the multicore CPU environment and in the CPU + GPU environment, are summarized in Figs. 5 through 8.

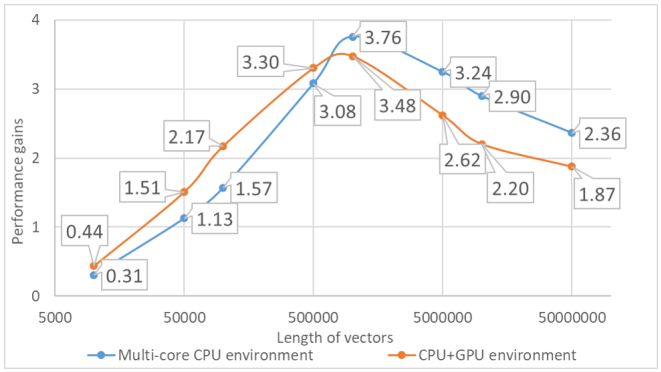


Figure 5: Simple vector calculations

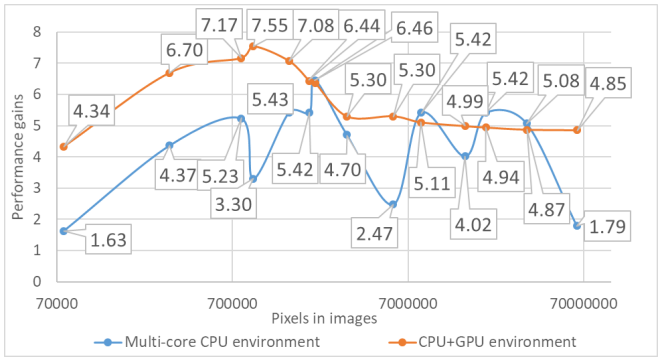


Figure 6: Gaussian filter and difference

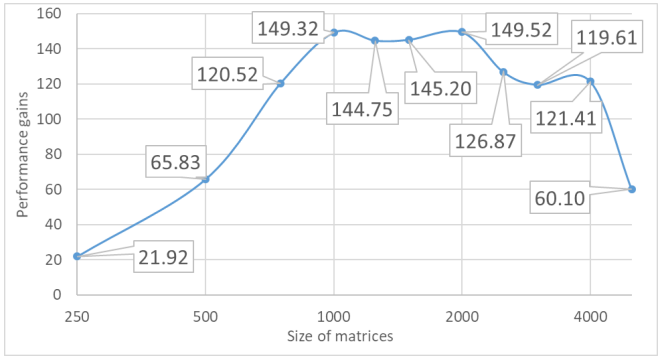


Figure 7: Matrix multiplication (multi-core CPU)

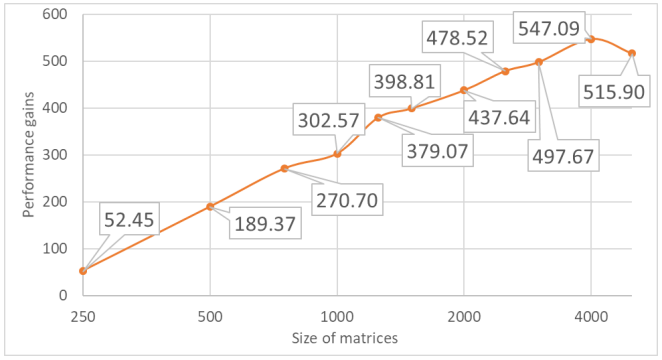


Figure 8: Matrix multiplication (CPU + GPU)

5.3.3 Discussion

5.3.3.1 Difference in performance gains through data parallelization between different models

The maximum performance gains attained from each type of Simulink model and in each target environment are summarized in Table 7. It is shown that, although the correctness of the data-parallel programs and their performance gains over the serial programs were confirmed across three types of Simulink models and two types of target environments, there were significant differences between the maximum performance gains attained.

Table 7: Maximum performance gains compared with the serial programs, attained on different Simulink models and environments

	Multi-core CPU environment	CPU + GPU environment
Simple vector calculations	3.76	3.48
Gaussian filter and difference	6.46	7.55
Matrix multiplication	149.52	547.09

It is assumed that this difference is due to the difference in the proportion of data-parallel operations between the models.

5.3.3.2 Performance gains in multicore CPU environments

With respect to the matrix-multiplication model, the maximum performance gain attained in the multicore CPU environment was 149.52, which significantly exceeds the core count (10) and thread count (20). The reason for this discrepancy is assumed to be the automatic optimizations, such as coalescing and loop unrolling, performed by PPCG and Intel DPCT on the code they generate, in addition to the parallelization.

5.4 Evaluation 2: Correctness and performance gains of programs exploiting both task and data parallelism

In this evaluation, we compare programs exploiting both task and data parallelism, generated through the method described in Section 4.3, with task-parallel programs generated by MBP, data-parallel programs, and serial programs.

5.4.1 Simulink models used

In this evaluation, we used the model depicted in Fig. 9, where matrix multiplications were performed. The candidate for data parallelization is limited to one of the blocks, that is, the *matmul* block, whose calculation scale is adjustable between 500×500 and 2500×2500.

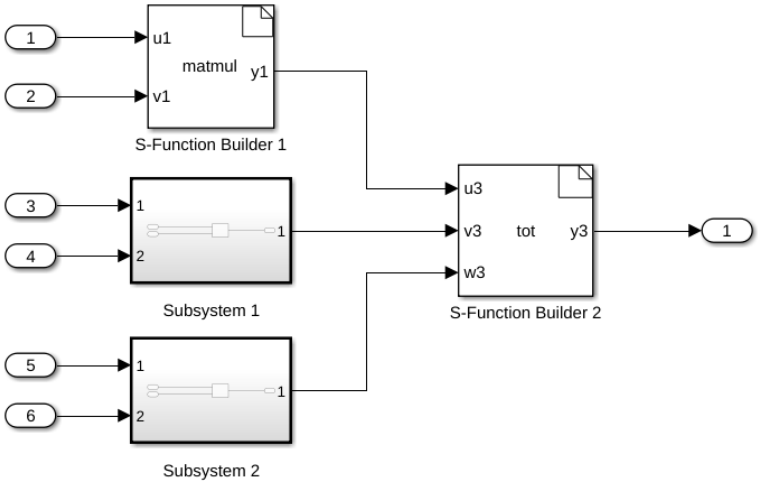


Figure 9: Simulink models used in evaluation 2

5.4.2 Results

Across the different calculation scales, there were no differences between the calculation results from the programs exploiting both task and data parallelism, those from task-parallel programs, those from data-parallel programs, and those from serial programs. This indicates the correctness of the programs exploiting both task and data parallelism generated according to our proposed approach, within the limits of this evaluation.

The performance gains through the different parallelization techniques in the CPU + GPU environment are summarized in Fig. 10.

5.4.3 Discussion

The performance gains attained by the programs exploiting both task and data parallelism were consistently greater than those attained by task-parallel programs and data-parallel programs, indicating the superiority of our proposed combined-parallelization approach.

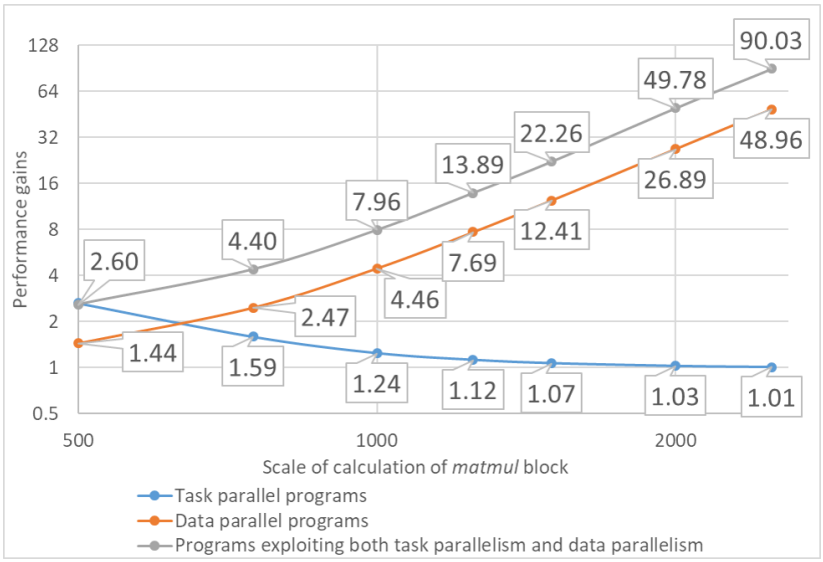


Figure 10: Performance gains through different parallelization techniques

The performance gains attained by the task-parallel programs decreased, while those attained by the data-parallel programs, as well as by programs exploiting both task and data parallelism, increased with the calculation scale of the candidate block for data parallelization. This is assumed to be due to the great effectiveness of parallelization on the candidate block, to the extent that the block is not on the critical path of the entire program. This can be confirmed by the extremely small variations in the time taken by data-parallel programs, as well as by programs exploiting both task and data parallelism, with respect to different calculation scales.

6. Conclusion

We proposed methods in which task-parallelism and data-parallelism were exploited to generate parallel programs from Simulink models. Specifically, we proposed the following approaches.

1. A model-based parallelizer (MBP) for automatically generating task-parallel C++ code, centered around the BLXML markup language proposed in [21], and using the hierarchical clustering method described in [22][23] for core assignment.

2. A semi-automatic approach for generating data-parallel SYCL code, which combines the PPCG and the Intel DPCT compilers.

3. An approach for semi-automatically generating programs exploiting both task and data parallelism through the combined application of the above two approaches.

Evaluation 1 compared data-parallel programs generated through our proposed approach to serial programs generated by Embedded Coder from a number of Simulink models. The data-parallel programs performed correct calculations, while attaining a maximum performance of approximately 547 times that of the serial programs.

Evaluation 2 compared programs exploiting both task and data parallelism, generated through our proposed approach, to task-parallel programs, data-parallel programs, and serial programs. The parallel programs performed correct calculations, and the programs exploiting both task and data parallelism achieved better performance than those generated while exploiting either one of the two.

Further research is needed with regard to the performance of parallel programs generated through our proposed approach on target platforms containing more than one type of accelerator, as well as an accurate estimation of the cost of communication between the host CPU and the accelerators in MBP.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] Alpay, A. (2020). *Hipsycl*. Heidelberg University. Retrieved January 24, 2021, from https://github.com/illuhad/hipSYCL.

[2] Cha, M., Kim, K. H., Lee, C. J., Ha, D., & Kim, B. S. (2011). Deriving high-performance real-time multicore systems based on Simulink applications, In *2011 IEEE 9th International Conference on Dependable, Autonomic and Secure Computing*. IEEE. https://doi.org/10.1109/DASC.2011.64

[3] Cha, M., Kim, S., & Kim, K. (2012). An automatic parallelization scheme for Simulink-based real-time multicore systems. *Science & Engineering Reasearch Support Society, 5*, 215-217.

[4] Cilardo, A., Socci, D., & Mazzocca, N. (2014). ASP-based optimized mapping in a Simulink-to-MPSoC design flow. *Journal of Systems Architecture, 60* (1), 108-118. https://doi.org/10.1016/j.sysarc.2013.10.004

[5] Codeplay. (2020). *Computecpp*. Codeplay Software Ltd. Retrieved January 24, 2021, from https://developer.codeplay.com/products/computecpp/ce/home/.

[6] Feautrier, P. (1996). Automatic parallelization in the polytope model. In *The Data Parallel Programming Model* (pp. 79-103). Springer. https://doi.org/10.1007/3-540-61736-1_44

[7] GNU. (2020). *Gnu gprof*. Free Software Foundation, Inc. Retrieved January 24, 2021, from http://sourceware.org/binutils/docs/gprof/.

[8] Han, G., Di Natale, M., Zeng, H., Liu, X., & Dou, W. (2013). Optimizing the implementation of real-time Simulink models onto distributed automotive architectures. *Journal of Systems Architecture, 59* (10), 1115-1127. https://doi.org/10.1016/j.sysarc.2013.08.009

[9] Intel. (2020). *Intel oneAPI*. Intel. Retrieved January 24, 2021, from https://software.intel.com/content/www/us/en/develop/tools/oneapi.html.

[10] Kasahara, H., & Narita, S. (1984). Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Computer Architecture Letters, 33* (11), 1023-1029. https://doi.org/10.1109/TC.1984.1676376

[11] Khronos. (2020). *Sycl*. The Khronos Group Inc. Retrieved January 24, 2021, from https://www.khronos.org/sycl/.

[12] Kumura, T., Nakamura, Y., Ishiura, N., Takeuchi, Y., & Imai, M. (2012). Model based parallelization from the Simulink models and their sequential C code, In *17th Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI 2012)*. http://sasimi.jp/new/sasimi2012/files/program/archive/pdf/p186_R2-8.pdf

[13] Mathworks. (2020). *Embedded Coder*. The MathWorks, Inc. Retrieved January 24, 2021, from https://www.mathworks.com/products/embedded-coder.html.

[14] Mathworks. (2020). *Simulink*. The MathWorks, Inc. Retrieved January 24, 2021, from https://www.mathworks.com/products/simulink.html.

[15] Tran, Q. M., Wilmes, B., & Dziobek, C. (2013). Refactoring of Simulink diagrams via composition of transformation steps, In *International Conference on Software Engineering Advances*, 140-145.

[16] triSYCL. (2020). *Trisycl*. Github, Inc. Retrieved January 24, 2021, from https://github.com/triSYCL/triSYCL.

[17] Tuncali, C. E., Fainekos, G., & Lee, Y.-H. (2015). Automatic parallelization of Simulink models for multi-core architectures, In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. IEEE. https://doi.org/10.1109/HPCC-CSS-ICESS.2015.232

[18] Umeda, D., Suzuki, T., Mikami, H., Kimura, K., & Kasahara, H. (2015). Multigrain parallelization for model-based design applications using the OSCAR compiler, In *Languages and Compilers for Parallel Computing*. Springer. https://doi.org/10.1007/978-3-319-29778-1_8

[19] Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gomez, J., Tenllado, C., & Catthoor, F. (2013). Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO), 9* (4), 1-23. https://doi.org/10.1145/2400682.2400713

[20] Verdoolaege, S., & Grosser, T. (2012). Polyhedral extraction tool, In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT '12), paris, france*. https://doi.org/10.13140/RG.2.1.4213.4562

[21] Yamaguchi, K., Takematsu, S., Ikeda, Y., Li, R., Zhong, Z., Kondo, M., & Edahiro, M. (2015). Block-level parallelization of simulink models [In Japanese], In *2015 Embedded Systems Symposium*.

[22] Zhong, Z., & Edahiro, M. (2019). Model-based parallelizer for embedded control systems on single-ISA heterogeneous multicore processors, *INTERNATIONAL JOURNAL OF COMPUTERS & TECHNOLOGY (IJCT), 19*, 7470-7484. https://doi.org/10.24297/ijct.v19i0.8123

[23] Zhong, Z., & Edahiro, M. (2020). Model-based parallelization for Simulink models on multicore CPUs and GPUs., *INTERNATIONAL JOURNAL OF COMPUTERS & TECHNOLOGY (IJCT), 20*, 1-13. https://doi.org/10.24297/ijct.v20i.8533