Embedded Multi-Core Code Generation with Cross-Layer Parallelization

Oliver Oey
Karlsruhe Institute of Technology, Germany
emmtrix Technologies GmbH, Karlsruhe, Germany

Michael Huebner
BTU Cottbus - Senftenberg, Germany

Timo Stripf
emmtrix Technologies GmbH, Karlsruhe, Germany

Juergen Becker
Karlsruhe Institute of Technology, Germany

Abstract

In this paper, we present a method for optimizing C code for embedded multi-core systems using cross-layer parallelization. The method has two phases. The first is to develop the algorithm without any optimization for the target platform. Then, the second step is to optimize and parallelize the code across four defined layers which are the algorithm, code, task, and data layers, for efficient execution on the target hardware. Each layer is focused on selected hardware characteristics. By using an iterative approach, individual kernels and composite algorithms can be very well adapted to execution on the hardware without further adaptation of the algorithm itself. The realization of this cross-layer parallelization consists of algorithm recognition, code transformations, task distribution, and insertion of synchronization and communication statements. The method is evaluated first on a common kernel and then on a sample image processing algorithm to showcase the benefits of the approach. Compared to other methods that only rely on two or three of these layers, 20 to 30 % of additional performance gain can be achieved.

2012 ACM Subject Classification Software and its engineering → Source code generation; Software and its engineering → Imperative languages; Software and its engineering → Very high level languages; Computer systems organization → Embedded software

Keywords and phrases Parallelization, multi-core Processors, model-based Development, Code Generation

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2024.5

1 Introduction

State-of-the-art embedded multi-core processors offer high performance with low power consumption, but programming them efficiently presents new challenges due to a high complexity e.g. in the partitioning of tasks on the specific multi-cores. Some of the main challenges when developing applications for embedded multi-core are:

- Developers tend to think sequentially. To take full advantage of multi-core systems, parallel applications need to be partitioned up front. This extra work does not occur in sequential development and distracts from the actual programming of the algorithm.
- To distribute tasks across processing units, data and control dependencies have to be considered to avoid errors like race conditions or deadlocks which cannot occur in sequential programs.
- Debugging is much more complex on multi-core systems because parallel processing with multiple threads does not ensure determinism. If execution is interrupted at any time, the current state of each processing unit cannot be predicted.

15th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 13th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2024).
Editors: João Bispo, Sotirios Xydis, Serena Curzel, and Luís Miguel Sousa; Article No. 5; pp. 5:1–5:13
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Programmers encounter challenges that divert their focus from implementing actual functionality. This paper proposes a solution that segregates application development from target platform optimization. The proposed solution involves commencing with a model-based design that remains entirely platform-agnostic, followed by iterative optimization grounded in four defined abstraction layers. Each layer progresses towards the hardware in granular stages. This approach ensures that algorithm development is completely decoupled from hardware platform optimization, as no code needs to be written after model implementation. The approach is intended to be used with applications that can be analyzed statically which means problem/data sizes are known or at least bounded so that the best performance of the application on the target platform can be achieved. There are various possibilities to implement the approach, including using multiple software tools within a tool flow for layer optimizations or employing manual optimization steps that concentrate on the defined layers only. For the remainder of this paper, we adopted a middle approach: using currently available tools for optimization at each layer, but without a fully integrated tool flow. We used tools developed by emmtrix Technologies[1] that were originally based on results from the ALMA research project [2]. While these existing tools provide the framework to apply the optimizations on different abstraction layers, the concept of this proposed cross-layer parallelization is not integrated in the usual tool flow. This novel combination of layers allows the use of specifically optimized implementations for identified algorithms, in addition to general optimizations such as code transformations and task distribution with optimized communication placement. The following four abstraction layers will be used in this paper:

1.1 Definition of Algorithm Layer

The algorithm layer in this approach refers to the abstract representation of an algorithm that has been developed independently of the target hardware platform. The choice and implementation of the algorithm has a high impact on performance. Let's take sorting algorithms as an example: different algorithms vary in terms of runtime, memory requirements, and stability. The potential for parallel execution is different with each implementation but usually comes with a slower sequential execution or increased memory requirements. The best selection can therefore only be made in combination with the information about how many processing elements and how much memory is available and how big the data set to process is. The goal of this layer is to provide a library of common algorithms or kernels with a selection of implementations to choose from, to identify these known algorithms in the source code and together with the decisions on later layers select the best performing realization for the actual hardware.

1.2 Definition of Code Layer

The code layer refers to the actual representation of the algorithm as source code. How the individual computations are represented determines how many (independent) tasks can be generated and thus how well parallelization can work. Code transformations can be used to change the code of the application without affecting the result of its calculations. They can be used to take advantage of the intrinsic parallelism that is already part of the code, for example, when processing large amounts of data. Two example transformation that are used in the evaluation:

[1] `https://www.emmtrix.com`

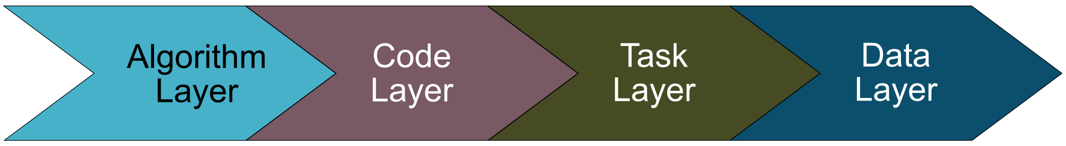

Figure 1 Abstraction layers used in this approach.

- Loop fission: If there are multiple statements in the body of a loop that can be executed independently, it is possible to split the statements into two or more loops over the same index range. Since the resulting loops have no dependencies on each other, they can be executed in parallel.
- Variable splitting: Instead of having a single array variable, splitting it into multiple ones allows parallel calculations on different regions of the original data without affecting each other.

The most beneficial transformations are applied to loops as that has the greatest potential for improving parallelism. But besides the loops and their number of iterations, other properties play a role at this layer as well, e.g. data access or locality, the actual number of individual code blocks and how well the code can be statically analyzed.

1.3 Definition of Task Layer

The task layer refers to the part of this approach at which tasks are assigned to individual execution units of the target platform. During parallelization, both mapping and scheduling are performed to select which core or processor a task is executed on and the order in which the tasks are executed there. All dependencies between tasks must be taken into account, mainly from the data flow, but also from the control flow. Depending on the target architecture, these dependencies lead to synchronization or communication overhead. In addition to pure data dependencies, anti and output dependencies must be considered to ensure the correct order of execution. Also important for parallelization decisions is the execution time of each task relative to each other and relative to the synchronization overhead. If the granularity of the tasks is not taken into account, the overhead of communication and synchronization may outweigh the benefit of parallel execution, resulting in a slowdown compared to the sequential program.

1.4 Definition of Data Layer

The data layer in this paper refers to synchronization and data exchange between cores. Without loss of generality, this work assumes hardware models with distributed memory. Any shared memory system can also be considered and treated as a distributed memory system by hard allocating the available memory to the cores. The goal of optimization on this layer is to ensure data availability on each core while achieving the best runtime on the hardware and ensuring that no errors are introduced due to false synchronization. Important aspects are the placement of synchronization instructions and keeping the overhead of transfers to a minimum.

Figure 1 shows all used layers and their typical usage from the input source code to the code for the platform. The remainder of this paper is organized as follows: Section 2 discusses the current state of the art and how the four layers are usually employed, Section 3 elaborates on the actual cross-layer optimization, Section 4 evaluates the approach with an experimental case study, and Section 5 concludes this paper with an outlook.

2 State of the Art

Foster[7] formulated already in 1995 a workflow to design and build parallel programs. The main steps were partitioning, communication, agglomeration and finally mapping of tasks and are taught at universities as a standard approach for parallelization. The approach is comparable with the last three layers of our approach but lacks the potential of special optimizations of known algorithms which allow more specialized implementations of known kernels that usually make up large parts of typical embedded applications.

In [5], methods for programming parallel platforms based on algorithmic skeletons and parallel design patterns are evaluated. While they prove useful for parallel programming, they don't ease the required hardware knowledge from the programmer. This kind of library-based approach mostly focuses on the algorithm layer to provide special parallel implementations with known functionality. Without any considerations from the task and data layers, this kind of parallelization is limited to these functions and resources respectively an efficient load balancing throughout the execution of the whole program cannot be achieved.

Automatic parallelization by the compiler is still a research topic in works like [9] and [8]. They focus on pattern recognition of common programming patterns with more recent work also on the usage of machine learning for the detection. This pattern recognition covers the optimizations on the algorithm layer and should achieve good results for programs that make great use of these patterns. However, programs usually also have parts that don't fit into patterns and might need some other optimization steps which could e.g. be applied on the code layer. [1] uses source-to-source compilation to optimize the source code for automatic parallelization using OpenMP. Together, this covers the code, task and data layers as defined in this work but does not use the benefits that optimizations on the algorithm layer could bring to known algorithms.

The Daedalus framework [14] is a more recent approach for the design of multi-processor system-on-chips. By approaching the issue together with system level synthesis, a synchronous development of software and hardware is performed. To optimize the software for parallel execution, the tool PNgen is used to apply polyhedral optimization techniques which result in loop transformations according to mathematical equations. This approach together with the design space exploration covers the two middle layers of our approach: the code and the task layer. By adding the algorithm layer, specific optimizations can improve the results while the data layer allows for more efficient execution of the parallel program on the target platforms.

A summary of popular parallelization techniques can be found in [13]. Comparing it to this approach it shows that they mostly focus on the code and data layers by optimizing loops and communication.

[12] shows how concepts from high performance computing (HPC) can be applied to embedded systems. HPC is originally more focused on achieving the best performance with the available hardware without taking the potential worst case into account. While this is not the case for typical embedded applications, the need to get the best performance out of multi or many-core processors is getting higher. The work described relies on OpenMP with task scheduling and does not take optimizations on algorithm or data layer into account.

Looking at heterogeneous systems, [11] compares different programming frameworks like OpenMP, OpenCL and CUDA regarding programming productivity, performance and energy. One major result here is that the human factor significantly impacts the fraction of lines of code used to parallelize the code. Reducing this human impact should therefore be the goal. The methodology proposed in this paper covers this aspect by only letting the programmer develop the model and no further changes to source code.

[16] introduces a compiler-based optimization specialized for machine learning. The compilation applies various parallelization techniques controlled by the user through source code annotations. This approach is sophisticated in the sense that the user does not necessarily need in-depth hardware knowledge. All four layers are addressed, with reference to the algorithm layer done through the limited applications that mainly relies on library calls. The code and task layers, along with the data layer, are automatically processed, utilizing user annotations as guidance for specific components.

Examining techniques that utilize multiple abstraction layers, studies such as [10] expand the LLVM intermediate representation using domain-specific languages (DSL) that can reuse the same compiler passes across numerous levels of abstraction. Although the initial aim was to use transformations to lower abstraction, research such as [4] can be utilized to raise the abstraction level and back-propagate information to higher levels. While this approach optimizes all abstraction layers used in this work iteratively, it lacks flexibility compared to the model-based approach used here due to its focus on DSLs.

Functional development is decoupled from optimization for the target platform in studies such as [15]. Functional development is conducted in the high-level general-purpose programming language Python 3 and the mapping process is carried out in Artisan meta-programs. This decoupling allows different experts to handle these developments, eliminating the need for understanding both the algorithm and the optimal implementation for the heterogeneous hardware. Compared to our approach, the presented work relies on downstream processing by OpenMP or high-level synthesis for optimal performance. Our work, on the other hand, optimizes program scheduling and data transfer directly on the task and data layers, eliminating the need for downstream tools or operating systems.

3 Cross-layer Optimization

While independent execution of the layers is possible, the combined approach provides distinct advantages. Progressing through the layers reduces the level of abstraction from the hardware at each stage, necessitating more information with each subsequent layer. When switching to a different hardware platform that has at least some similarity with the previous one, e.g. the same numer of processing elements, some optimizations can be re-used. To demonstrate the optimizations achieved at each layer and their potential interactions, we will employ the Fast Fourier Transform (FFT) as an exemplar kernel. It was selected as it is a well-known algorithm and shows good optimization potential on each defined layer.

3.1 Realization of Algorithm Layer

Recognition of known algorithms cannot simply be achieved by extracting them from source code, as illustrated by the halting problem, where determining a specific behavior for generic programs is impossible. Our solution is to use MATLAB® as the programming language for a model-based approach. It offers several built-in versions of common algorithms, with clear descriptions of the results. So instead of attempting to identify established algorithms, the specific algorithm can be identified through a straightforward function call and precise behavior specification within the MATLAB® description. To transfer the algorithm to the embedded device, it was converted from MATLAB® to C code, as this enables more targeted optimizations customized for the designated hardware. The code generation tool developed includes support for basic arithmetic operations. For example, when adding two arrays, the tool will convert them to arrays in C, which are then summed up with for-loops. Library functions are utilized to customize function conversions, implementing the requisite functionality in MATLAB® scripts, subsequently generating the pertinent C code. Data type and range analysis ensure the employment of only the most efficient data types in the C code. Furthermore, this approach can offer unique realizations of established functions that can be chosen with custom pragmas within the scripts. This permits additional enhancements of the produced C code by returning to the code generation phase from later stages. To achieve efficient implementation on the embedded target platform, a conversion of code must be undertaken due to the inefficiency of executing MATLAB® scripts. Utilizing C code on embedded platforms provides greater convenience and enables precise control of hardware resources. The conversion step also facilitates the selection of an algorithm realization and the generation of C code optimized for later layer simplifications.

The algorithm layer only uses the number of available cores as information about the target platform. This information can be used to select an implementation that is optimized for that number of cores.

Now, looking at the FFT, we can see that it is an in-built function of MATLAB® that has a clear specification about its behavior and accuracy. Knowledge about the algorithm allows us to provide an option for later layers of the flow: a N-point FFT may be substituted with two $N/2$-point FFTs, which can be computed independently, as illustrated in Figure 2. In other words, a simple option for the code generation can be used to specify the number of times the FFT should be split up into. With the additional information about the number of processing cores available, this can also be used as an upper boundary as splitting the FFT up into more parts is usually not useful.

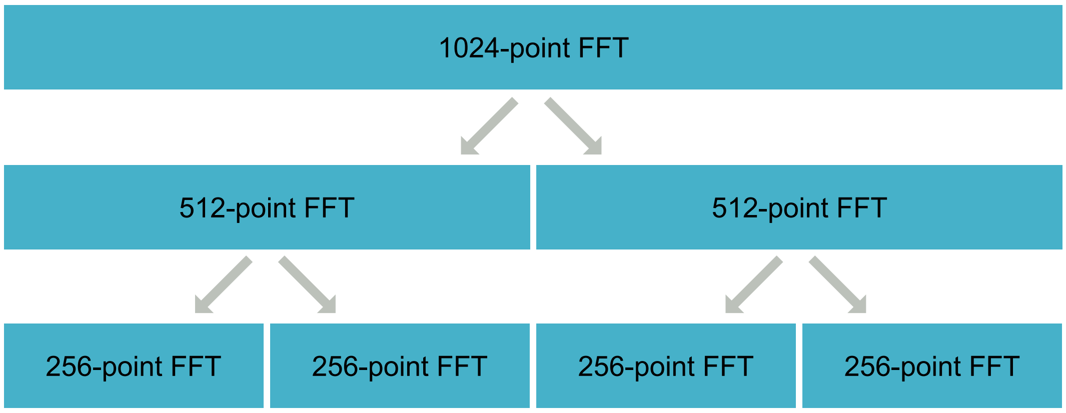

Figure 2 Splitting FFTs.

3.2 Realization of Code Layer

The code layer focuses on transforming the C code for task level parallelization. It is implemented as a source-to-source compiler that reads the input C code, applies selectable code or loop transformations to the desired parts of the code, and then outputs the transformed C code. While these steps are not new in themselves, the importance lies in their interaction with the other layers: Transformations in the code layer rely heavily on input from the algorithm layer. E.g. an increased number of loops increases the potential for beneficial loop transformations. The granularity of the transformations ensures that independent tasks can be generated very well at this point for parallelization at the task layer. However, the number of tasks should be determined together, since too many interdependent tasks only increase the complexity of the scheduling algorithm without offering more optimization potential. All transformations that can change the access to the data also have the potential to reduce the overhead at the data layer. At this layer, more information about the hardware can be used for all decisions. Besides the actual number of processing elements, the type is relevant to address available accelerators. Optimizations that improve data locality require information about memory layout and cache availability to be useful.

For the FFT example, code transformations may be applied to the main loops of the kernel by using variable splitting and loop fission to generate independent loops. Considering the splitting option at the algorithm layer, this provides several options for dividing the code execution. However, the best options will be selected with the assistance of the next layers.

3.3 Realization of Task Layer

After the enabling optimizations on the previous layers, the main part of the coarse-grain parallelization takes place on this layer. As a pure optimization algorithm on a graph representation, it takes into account all dependencies that are crucial for the allocation to the individual cores. This is important for maintaining correctness, but sometimes overestimates the actual performance. Since the placement and optimization of the communication takes place later at the data layer and is based on a different representation, the synchronization and duplication of the data cannot yet be fully considered and can only be passed as hints to the next layer.

The most important function of this layer is to identify parts of the code that can be executed independently by analyzing the data and control dependencies, and then to assign these extracted tasks to the available cores so that the overall runtime is minimized. This requires information about the actual runtime on each core to model the optimization problem and the cost of the overhead of parallelization. The actual parallelization at this layer of abstraction can be done with many different optimization algorithms. In this work, the well-known Heterogeneous Earliest Finish Time (HEFT)[8] is used. As a greedy heuristic algorithm, it usually does not reach the optimal runtime, but it has been extended to respect user constraints like fixing tasks on certain cores in order to guide the optimization process in the right direction. The parallelization on the task layer distributes all tasks onto the available cores taking into account the communication overhead required to synchronize the data. To minimize this overhead and ensure the correct execution, the actual placement and optimization of the communication is handled on the next layer.

For the FFT example, partitioning of the code on both the algorithm and code layers is decided at the task layer. It is only through the utilization of a more precise task runtime cost model on the actual hardware and overhead costs for parallel execution that the overhead of the two approaches can be accurately calculated. A performed test on an ARM Cortex A-76 processor with four cores demonstrated that dividing the FFT by four in the algorithm layer gets the best use of the available cores. Code transformation are useful on the parts where the data is partitioned, but as more involved also increases the required data transfers, an optimization for two cores on the code layer achieves the best runtime.

3.4 Realization of Data Layer

The main function of this layer is to resolve all data dependencies that cross core boundaries by inserting send and receive instructions for data transfers. Done correctly, this ensures that the two most common errors in parallelization cannot occur: race conditions and deadlocks. Race conditions occur when the result of one operation depends on the timing of another operation. This can often happen when shared resources are accessed by multiple cores at the same time. By modeling these accesses through data dependencies, synchronization instructions can be inserted whenever the resource is accessed, enforcing a predetermined order across all cores that need to access the resource. This ensures the absence of race conditions with the added overhead of synchronization whenever resources are shared. A deadlock is an error state that occurs when the program waits indefinitely for data or signals that never arrive. The program is stuck and cannot continue its execution. One way to avoid this is to duplicate the control flow for all cores involved. That is, if a signal is sent within an if-block only when a certain condition is met, the receiving core executes a copy of the if-block that evaluates the same condition. This guarantees that send and receive are always executed under the same conditions and that a receive is never called without an associated send. The data layer requires the most details about the target platform. It needs information about the communication infrastructure with all the data buffers, access to the interface code, and information about what the C code should ideally look like for the target platform's compiler.

For the FFT example, the data layer is crucial for achieving the expected performance as calculated at the task layer. To minimize core wait times, it is necessary to have a better understanding of the actual interconnections between the cores and to manage data transfers efficiently. This depends heavily on the volume of data being processed. As FFT is typically utilized for larger amounts of data and requires minimal synchronization between the cores, achieving high throughput is critical for optimizing the runtime on the hardware.

3.5 Interactions Between the Layers

With each layer, the optimization takes into account additional information about the hardware specification. Figure 3 depicts which main features are utilized by each layer. The arrows at the bottom of the figure show the typical process of cross-layer optimization: the order of the layers is the algorithm, code, task, and data layer. From the code layer to the algorithm layer, transitions occur when there is a need to implement a modified algorithm or prepare for a specific number of cores. Likewise, the task and data layers transition to the code layer when the code's representation makes it difficult to efficiently allocate tasks among the cores, either due to the volume of tasks or the resulting communication overhead. From the data layer back to the task, reevaluation is necessary when scheduling proposes parallelization that would create excessive communication overhead in the data layer. Considering a change made on a previous layer would be advantageous, going back more than one step is also an option.

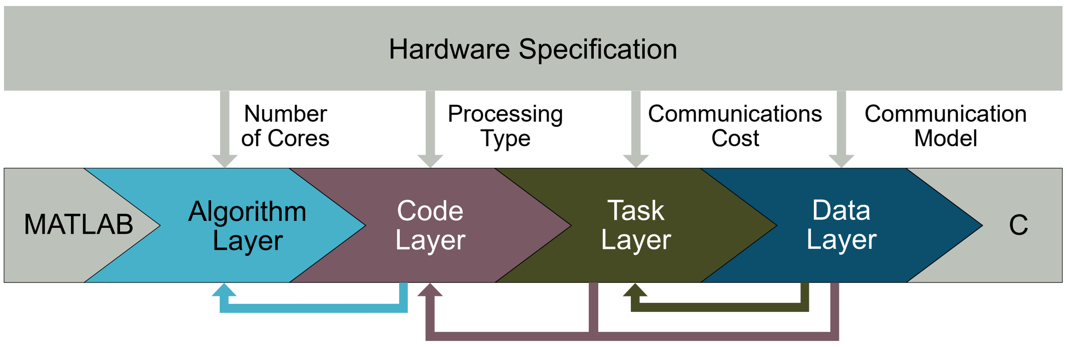

Figure 3 Interactions between the different layers.

4 Evaluation of the Method

For the evaluation, a streak detection algorithm as utilized for space debris detection is used. It was selected as it best shows the potential of the approach on all the layers. The algorithm consists of two main steps:

1. The input image is converted to grayscale and a Canny edge detector is used to extract all edges in the image. The Canny edge detector is a well known algorithm [3] that is still quite popular due to its low error rate and high accuracy. It is applied on an image that is usually denoised with a Gaussian blur filter to find the edges identified with the highest gradient change.
2. A Hough transform is applied to the resulting image, which is used to detect the peaks with the highest contrast and to mark the found lines in the image. The Hough transform is a formerly patented method [6] that is used to find geometric shapes such as straight lines in binary black and white representations.

The source code of the algorithm is shown in Listing 1. The most relevant parts of the program are directly available in MATLAB® so that calling the inbuilt functions is sufficient. Only for the function to extract a line, a new custom function needed to be developed.

Listing 1 MATLAB® code of streak detection.

```
function [point1Arrays,point2Arrays] = streak_detection(img)
  %convert to grayscale, available in MATLAB
  gray = rgb2gray(img);
  %extract edges from image, available in MATLAB
  [E,thresh] = edge(double(gray),'Canny');
  %apply Hough transform, available in MATLAB
  [H,T,R] = hough(E);
  %extract peaks in Hough representaiton, available in MATLAB
  P = houghpeaks(H, 50);
  %draw a line to connect points, implemented manually
  [point1Arrays,point2Arrays] = custom_houghlines(E,T,R,P);
end
```

The target system is an NXP P4080 DS with 8 PowerPC e500mc cores.

4.1 Evaluation of Algorithm Layer

Since the Canny edge detector algorithm is directly available in MATLAB® a call to the function can be used as an entry point for the optimization at the algorithm layer. This offers the potential for a few different optimizations: fixing the threshold for the edge detection maintains the same results while reducing the runtime by 6%. Switching to single precision data types with 32-bit floats still has enough accuracy, but reduces the runtime by another 28%. Finally, optimizing the memory allocation allows a further optimization of 19%, for a total gain of 45% on the algorithm layer alone. The optimizations were applied without any specific hardware knowledge, and the reduced complexity provides more options in later layers of the flow. The performance gains were determined on the development PC using an Intel Core i5-4570 with 16 GB DDR3 RAM with 1600 MHz. Measured was the execution time of the C program generated from the MATLAB® code and the initial 259 ms were reduced to 140 ms.

Most functions in the source code 1 are part of the MATLAB® language scope, so that specially optimized versions can be used during code generation. By using parameters to guide the code generation, the generated C code can be prepared for later transformations. In order to be able to apply the beneficial variable splitting and loop fission transformations on the functions `rgb2gray`, `edge` and `hough`, the number of generated loops and variables can be changed according to the input from the code layer.

4.2 Evaluation of Code Layer

The main task of the code layer is to provide the task layer with as many independently computable tasks as possible. This is best achieved from the combination of variable splitting and loop fission. The C code prepared at the algorithm layer now allows easy exploration of the various options. The best results are achieved when the load between cores is balanced over time with as little data exchange as possible. For the `hypot`, `hough`, and `rgb2gray` functions, which each perform only a few calculations on a single matrix, the best solution is to split them among eight, i.e., all available cores of the target system. For the Canny Edge algorithm, which is executed independently in the X and Y directions, splitting for four cores at a time has been shown to give the best results. By executing the two directions in parallel, all eight cores can be utilized and limiting them to four cores each reduces the communication overhead. While the actual splitting of the variables and loops is done on this code layer, the decision about the splitting factors is decided by the later task and data layers.

4.3 Evaluation of Task Layer

The previously generated independent tasks ensure that parallelization at the task layer can be solved largely automatically. Only for tasks that split or collect data is there room for optimization in the allocation by selecting cores in a way that minimizes communication. This is done in conjunction with the data layer, since it is there that the final placement of the communication instructions is made.

The parallelization on task layer uses a simple cost model for the actual timing: each time data is accessed by multiple cores, a cost is added to reflect the overhead. The algorithm then attempts to reduce the overall runtime while minimizing the communication overhead. This model can be used to determine the best split factors for the code layer. The results of the scheduling on the task layer can be seen in Fig. 4. It shows the load on the eight cores of the target platform over time. During the first half of the execution, the edge detection in both directions can very well utilize all available cores. While there is still some parallel processing left afterwards at the 350 ms mark, the end is dominated by functions that don't benefit much from the parallelization on either layer. For the actual Canny edge algorithm, a speedup of 6.28 is estimated while for the complete application, a speedup of 3.02 is estimated. A more accurate estimate of performance, including core wait times for data, is handled on the data layer.

4.4 Evaluation of Data Layer

Due to the nature of the algorithm, placement at the data layer is very straightforward: data is always sent after it has been split or computed in the previous step, and it is received immediately before further processing. It is important to transfer data between the cores as quickly as possible to minimize waiting times. To determine the best positions for the

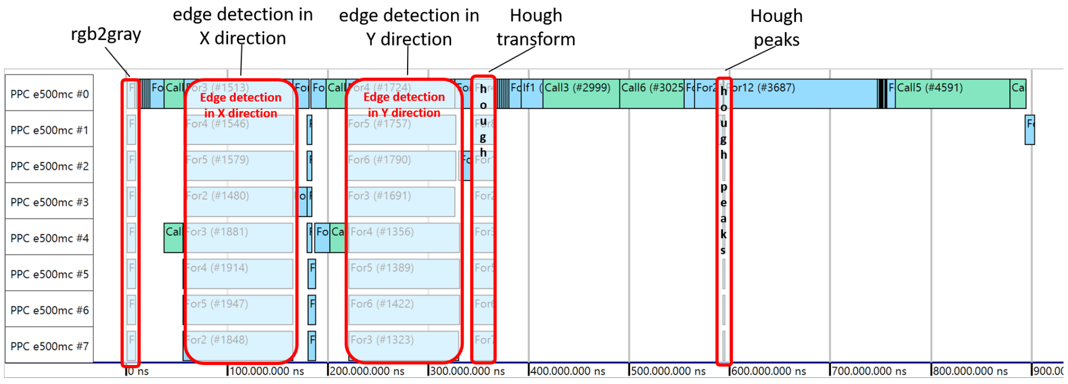

Figure 4 Scheduling result of the task layer.

synchronization between the cores, the control flow of the application is used to find the blocks that have the lowest number of executions while also minimizing any latency on the receiving cores. The communication overhead is relatively large so that some of the gains from the task layer cannot be implemented in the actual code for the hardware. Without any further changes on the data layer, a speedup of 2.4 was achieved. Further optimizations to reduce the communication overhead by implementing functions that directly access the shared memory regions gained an additional 19 % for a final speedup of 2.86.

5 Conclusion and Outlook

The runtime of the streak detection algorithm was reduced by 65 %, from an initial runtime of 2720 ms to 952 ms. Most of the optimization was done at the algorithm and code layer, but parallelization at the task layer was critical to map the prepared tasks to the available cores. Finally, the optimizations at the data layer were necessary to get most of the performance gains from the task layer to the actual hardware. This short evaluation shows the potential of this cross-layer approach and how the interaction between optimizations on different hierarchies can achieve better results compared to focusing only on two or three layers. However, more evaluations will be necessary to further analyze the benefits of this approach.

While the evaluation presented here was performed with a data-driven algorithm, the approach is not limited to it: further tests with more control-flow-driven algorithms also showed promising performance gains. Differences could be observed in the actual optimizations to be applied on the different layers. The examples used were of low complexity on the algorithmic side, so the task layer had little impact on the selection of the most efficient versions. The code layer still proved to be very valuable, but instead of ensuring that enough independent tasks were available for the task layer, smart clustering of the many tasks to reduce the actual number for the scheduling was the way to go. The task layer then had to find the most efficient distribution over the available cores, while the focus of the data layer is to minimize the amount of communication, since the cost per transfer is much higher when only few bytes need to be transferred.

It is also possible to extend the approach to support heterogeneous systems: the most relevant layers are then the algorithm layer and the code layer. The algorithm layer allows the use of optimized functions from existing libraries, while the code layer and its code transformations can be used to generate source code for accelerators such as CUDA or OpenCL.

References

1 Hamid Arabnejad, João Bispo, João M. P. Cardoso, and Jorge G. Barbosa. Source-to-source compilation targeting OpenMP-based automatic parallelization of C applications. *The Journal of Supercomputing*, 76(9):6753–6785, December 2019. doi:10.1007/s11227-019-03109-9.

2 Jürgen Becker, Thomas Bruckschloegl, Oliver Oey, Timo Stripf, George Goulas, Nick Raptis, Christos Valouxis, Panayiotis Alefragis, Nikolaos Voros, and Christos Gogos. Profile-Guided Compilation of Scilab Algorithms for Multiprocessor Systems. In *Reconfigurable Computing: Architectures, Tools, and Applications: 10th International Symposium, ARC 2014, Vilamoura, Portugal, April 14-16, 2014. Proceedings 10*, pages 330–336. Springer, 2014.

3 John Canny. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, November 1986. doi:10.1109/tpami.1986.4767851.

4 Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. Progressive Raising in Multi-level IR. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, February 2021. doi:10.1109/cgo51591.2021.9370332.

5 Marco Danelutto, Gabriele Mencagli, Massimo Torquati, Horacio González–Vélez, and Peter Kilpatrick. Algorithmic Skeletons and Parallel Design Patterns in Mainstream Parallel Programming. *International Journal of Parallel Programming*, 49(2):177–198, November 2020. doi:10.1007/s10766-020-00684-w.

6 Richard O. Duda and Peter E. Hart. Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972.

7 Ian Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995.

8 Saiyedul Islam, Sundar Balasubramaniam, Shruti Gupta, Shikhar Brajesh, Rohan Badlani, Nitin Labhishetty, Abhinav Baid, Poonam Goyal, and Navneet Goyal. Pattern-Based Automatic Parallelization of Representative-Based Clustering Algorithms. In *2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, October 2018. doi:10.1109/dsaa.2018.00020.

9 Nikita Kataev. Interactive Parallelization of C Programs in SAPFOR. In *SSI*, pages 139–148, 2020.

10 Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, February 2021. doi:10.1109/cgo51591.2021.9370308.

11 Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*. ACM, July 2017. doi:10.1145/3110355.3110356.

12 Luís Miguel Pinho, Eduardo Quinones, and Andrea Marongiu. *High-performance and time-predictable embedded computing*. River Publishers, 2018.

13 Sabri Pllana and Fatos Xhafa, editors. *Programming multi-core and many-core computing systems*. John Wiley & Sons, Inc., January 2017. doi:10.1002/9781119332015.

14 Todor Stefanov, Hristo Nikolov, Lubomir Bogdanov, and Angel Popov. DAEDALUS framework for high-level synthesis: Past, present and future. In *2021 25th International Conference Electronics*. IEEE, June 2021. doi:10.1109/ieeeconf52705.2021.9467445.

15 Jessica Vandebon, Jose G. F. Coutinho, Wayne Luk, Eriko Nurvitadhi, and Tim Todman. Artisan: a Meta-Programming Approach For Codifying Optimisation Strategies. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 177–185, 2020. doi:10.1109/FCCM48280.2020.00032.

16 Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. GSPMD: General and Scalable Parallelization for ML Computation Graphs, 2021. doi:10.48550/arXiv.2105.04663.