Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Efficient Execution of Microscopy Image Analysis on Distributed Memory Hybrid Machines

Willian de Oliveira Barreiros Júnior

Tese apresentada como requisito parcial para
conclusão do Doutorado em Informática

Orientador
Prof. Dr. George Luiz Medeiros Teodoro

Brasília
2023

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Efficient Execution of Microscopy Image Analysis on Distributed Memory Hybrid Machines

Willian de Oliveira Barreiros Júnior

Tese apresentada como requisito parcial para
conclusão do Doutorado em Informática

Brasília, 8 de março de 2023

Dedicatória

Este trabalho só pôde ser feito com o apoio da minha família, a pedra fundamental sob a qual me suporto. Em especial, dedico este trabalho à minha mãe Rosângela e meu irmão Lucas.

Agradecimentos

Primeiramente, agradeço ao meu orientador Prof. Dr. George Luiz Medeiros Teodoro, com quem tenho tido o prazer e a honra de trabalhar por mais de 6 anos. Sou muito grato por sua influência, já me auxiliando no meio de HPC desde minha monografia, me orientando no mestrado, e agora em meu doutorado. Não somente agradeço por todo conhecimento específico do nosso domínio de pesquisa, mas também por me guiar no meio acadêmico e pessoal.

Gostaria de agradecer aos colegas pesquisadores com quem tive contato nesse período de doutorado. Agradeço aos pesquisadores Prof. Dr. Joel Saltz e Dr. Tashin Kurc pelos recursos de supercomputação disponibilizados para minha pesquisa. Agradeço também à Profa. Dra. Alba Cristina Magalhães Alves de Melo por sua contribuição nos trabalhos publicados neste período, assim como sua orientação nesses últimos meses antes da entrega dessa tese.

Por fim, agradeço aos meu amigos, em especial Luís e Igor, que não apenas me ajudaram a permanecer são nesse período, mas pelas nossas diversas conversas e saraus de discursões técnicas do campo de expertise de cada um.

Resumo

A análise de imagens de *whole slide tissue image* (WSIs) é uma tarefa computacionalmente cara, impactando negativamente no uso de dados de patologia em imagens em larga escala para pesquisa. Diversas soluções paralelas para otimizar tais aplicações já foram propostas, mirando no uso de dispositivos e ambientes, como CPUs, GPUs e/ou sistemas distribuídos. Porém, a execução eficiente de código paralelo em máquinas híbridas e/ou distribuídas permanece um problema em aberto para histopatologia digital. Desenvolvedores de aplicações podem precisar implementar múltiplas versões de código para diferentes dispositivos de hardware. Desenvolvedores também precisam lidar com os desafios de distribuição eficiente de carga para nós computacionais de máquinas de memória distribuída, assim como para os dispositivos de execução de cada nó. Essa tarefa pode ser particularmente difícil para análises de imagens de alta resolução com custo computacional dependente de conteúdo. Esta tese tem como objetivo propor uma solução para a simplificação do desenvolvimento de aplicações de análise de WSI, assegurando o uso eficiente de recursos distribuídos híbridos (CPU-GPU). Para esse fim foi proposto um modelo de execução de alto nível de abstração, em conjunto com um método de particionamento automático de carga. A fim de validar os métodos e algoritmos propostos, uma linguagem de processamento de imagem de alto nível de abstração (*Halide*) foi utilizada como solução de paralelismo local (CPU/GPU), junto com o *Region Templates* (RT), um sistema de gestão de coordenação de dados e tarefas entre nós distribuídos. Também foi desenvolvida uma nova estratégia *cost-aware* de particionamento de dados (CADP) que considera a irregularidade de custo de tarefas a fim de minimizar o desbalanceamento de carga. Para tal, dois algoritmos de particionamento foram propostos, o *Expected Cost Bisection* e o *Background Removal Bisection.* Resultados experimentais mostram melhorias significativas na performance de execução com recursos híbridos CPU-GPU, comparada com o uso de recursos homogêneos (CPU ou GPU apenas). Os algorithmos de particionamento foram comparados com uma abordagem *baseline* hierarquica usando *KD-Trees* (KDT), em ambientes multi-GPU, multi-GPU híbrido e distribuído de larga escala. Os resultados mostraram ganhos de até 2.72× para o ECB e de 4.52× para o BRB, ambos em comparação ao KDT. Em adição ao modelo simplificado de desenvolvimento de *workflows* por experts de domínio, a performance vista em ambos ambientes híbridos e de larga escala demonstra a eficácia do sistema proposto para uso em estudos WSI de larga escala. Ambas melhorias na performance dos algoritmos do CADP como no modelo de estimação de custo de execução são esperadas como trabalhos futuros para o sistema aqui proposto.

Palavras-chave: HPC, Histopatologia, Halide, *Region Templates Framework*, Computação Heterogênea, Computação Distribuida, Processamento de Imagens, Particionamento Irregular de Dados

Abstract

The analysis of high resolution whole slide tissue images (WSIs) is a computationally expensive task, which cost adversely impacts large scale usage of pathology imaging data in research. Parallel solutions to optimize such applications have been proposed targeting multiple devices and environments, such as CPUs, GPUs, hybrid compute nodes and distributed systems. However, the generalization of efficiently executing parallel code on hybrid and/or distributed machines remains an open challenge for digital histopathology. An application developer may have to implement multiple versions of data processing codes targeted for different compute devices. The developer also has to tackle the challenges of efficiently distributing computational load among the nodes of a distributed memory machine and among computing devices within a node. This can be particularly difficult for analysis of high-resolution images with content-dependent computing costs. This thesis aims to provide a solution for simplifying the development of WSI analysis workflows while also enabling efficient use of distributed and hybrid (CPU-GPU) resources. For this end, a high-level execution model, coupled with an automatic workload partitioning method was proposed. In order to validate the proposed methods and algorithms, a high-level image processing language (Halide) was used as a local resource (CPU/GPU) parallel solution, together with Region Templates (RT), a system for managing data/tasks coordination among distributed nodes. A novel cost-aware data partitioning strategy that considers the workload irregularity to minimize load imbalance was also developed. For it, two partitioning algorithm were proposed, the Expected Cost Bisection (ECB) and the Background Removal Bisection (BRB). Experimental results show significant performance improvements on hybrid CPU-GPU machines, as compared with using a single compute device (CPU or GPU), as well as with multi-GPU systems. The partitioning algorithms were compared with a baseline hierarchical KD-Tree (KDT) approach, on multi-GPU-only, hybrid CPU-GPU and large-scale distributed CPU nodes environments. Results show speedups of up to 2.72× for ECB and 4.52× for BRB, both compared to KDT. In addition to the simpler development model for domain experts, the attained performance for both hybrid and large-scale distributed computing environments demonstrates the efficacy of the proposed system for large-scale WSI studies. Both improvements on the CADP algorithms performance and the accuracy of the execution cost estimation model are expected as future works for the proposed system.

Keywords: HPC, Histopathology, Halide, Region Templates Framework, Heterogeneous Computing, Distributed Computing, Image Processing, Irregular Data Partitioning

Contents

1 Introduction **1**
1.1 Problem Definition . . . 3
1.2 Research Hypothesis . . . 3
1.3 Goals . . . 3
1.4 Contributions . . . 5
1.5 Thesis Organization . . . 5

2 Background **7**
2.1 Taxonomy for Distributed and Parallel Infrastructures and Applications . . . 7
2.2 Parallel Programming Tools and Languages . . . 10
2.2.1 The Message Passing Interface (MPI) . . . 11
2.2.2 OpenMP . . . 12
2.2.3 CUDA . . . 12
2.3 Frameworks for High-Performance Image Processing . . . 14
2.3.1 OpenCV . . . 14
2.3.2 Halide . . . 15
2.3.3 The Region Templates Framework (RT) . . . 23
2.4 Motivating Field of Research . . . 25
2.5 Motivating Application . . . 27

3 Related Works **29**
3.1 Spatial Data Partitioning . . . 29
3.1.1 Fixed-Grid Spatial Partitioning . . . 30
3.1.2 Recursive Spatial Partitioning . . . 30
3.2 Related Languages, Frameworks and Tools . . . 32

4 Supporting Pathology Image Analysis Applications on Distributed Memory Hybrid Systems **35**
4.1 System Overview . . . 36
4.2 Interface for Implementing the Applications . . . 38

4.3 Internal Data Management for Hybrid Execution 40
4.4 Halide Integration and Implementation Details 42

5 Cost-Aware Data Partitioning for Irregular-Cost Applications 45
5.1 The Expected Cost Bisection (ECB) CADP Algorithm 46
5.1.1 Background Separation/Partitioning 46
5.1.2 Data Partitioning for Homogeneous Environments 50
5.1.3 Data Partitioning for Hybrid Environments 52
5.2 The Background Removal Bisection (BRB) CADP Algorithm 53
5.2.1 A Method for Hierarchical Partitioning with Background Removal . . . 56
5.3 Data Region Cost-Functions 61
5.4 Time Complexity Analysis of CADP 62
5.4.1 Time Complexity Analysis of ECB 62
5.4.2 Time Complexity Analysis of BRB 63

6 Experimental Results 64
6.1 Single Node Evaluation 65
6.1.1 Comparison to handwritten and Multi-core scalability 67
6.1.2 Multi-GPU Execution 68
6.1.3 Cooperative CPU/GPU Execution 71
6.2 Distributed Memory Execution 75
6.2.1 Execution Time of the Partitioning Algorithms 79
6.3 Impact of Background Removal 80
6.4 Impact of Inaccurate Cost-Function Estimates 83

7 Conclusion 85

References 89

List of Figures

2.1 Duncan's taxonomy for parallel hardware architectures. Image extracted from [1]. 9
2.2 CUDA memory hierarchy and hardware organization. Image from [2]. . . . 13
2.3 CUDA organization of execution units, or threads. Image from [2]. 13
2.4 Illustration of possible scheduling tradeoffs for the Distributed Halide schedules example of the blurring algorithm. Each line of blocks is processed on a distributed setting, with the first column being the *blurx* function and the second *blury*. 22
2.5 RTF architecture and workflow execution steps: (1) Worker queries Manager for stage instances created by the application workflow, (2) Worker consumes instances for processing, (3) Fine-grain tasks created in each Worker are assigned for execution with CPU and/or GPU, (4) Data is read, tasks processed, and results written to storage, and (5) Worker is notified of the end of a stage instance execution. 24
2.6 Segmentation and Feature Computation phases of the motivating application. 28

3.1 Partitioning examples of multidimensional point-based data and image-based data. 30
3.2 Fixed-grid partitioning examples. 31
3.3 Recursive spatial data partitioning examples. 31

4.1 Execution model of the proposed system. A set of WSIs and a application workflow are used as inputs. The system then partitions the input WSIs and executes one stage of the workflow at a time. Partitions are executed in parallel. Borders are automatically resolved at the end of the execution of a given stage, before the next stage can be executed. 37
4.2 Process of executing a stage. After partitioning and task submission any required data partitions are loaded for the required resource. After execution and border resolution a new stage can be requested. 38

4.3 An illustration of the processes involved into using multiple GPU devices. Direct communication between CUDA and user code is just a simplification. In reality all CUDA-related calls are performed only through Halide. 44

5.1 Overlapping BB resolution. For the horizontal/vertical cases, A is reduced. For the Diagonal cases, B is broken into 4 tiles, being the fully overlapped tile removed. 47
5.2 Background partition generation from initial dense partitions using the scan line algorithm. Partitions are numbered according to the order they were generated. 49
5.3 An example partitioning with at upper limit of 10% error on a tile with expected cost of 37. Initial bounds $begin_1$ and end_1 image limits are defined and tightened after each split attempt until a partition with imbalance smaller than the target is found. 52
5.4 Two cases of ECB partitioning. Although the partitions cost was similar, the shapes for generating 32 partitions are mostly long and thin strips. . . 54
5.5 Although the coarse-grain background removal algorithm can remove most of the background, some of it still remains inside the dense rectangular regions. 55
5.6 Viability study for fine-grain background removal using 60 images. 55
5.7 Hierarchical partitioning for 6 expected partitions. First, 2 partitions are generated (red) from the single initial dense partition (blue). Then, 3 more partitions (green) are required by each of the previous 2, resulting in 6 partitions. 56
5.8 First approach for background removal. The contour of the dense region is delineated in white, being the background on the lower left corner of the image. Two background tiles are expected to be generated. Background tiles generated are expected to maximize their area. 57
5.9 Generation of partitions which result in removing previously generated background tiles. The partition highlighted in red is considered too small, and as such should be merged with another partition. 57
5.10 Extraction of partitions with a background removal version of BTS. It is expected to generate 5 partitions. Red lines mark the current partition generated. The remaining partition is shown in green. Previously generated partitions are shown in blue. The black area is the background. 60

5.11 BRB partitioning algorithm for 4 and 32 expected partitions. ECB partitioning results for the same image are also shown as a comparison to BRB. 61

6.1 All WSIs images used for the experimental evaluations. Each image has its ID and their resolution. 66
6.2 Scalability of the motivating application developed using Halide on a single CPU with 28 cores. 68
6.3 Full data for single node experiments executed on a single GPU-only compute node. For each image, the values are related to the number of compute elements used, from 1-4 GPUs for the used node. 69
6.4 Application makespan of images BR-8682 and B6-A0X1 with related scaling efficiency for 1 to 4 GPUs used. 70
6.5 Application makespan of images BR-8682 and B6-A0X1 with related scaling efficiency for 1 to 4 GPUs used. 70
6.6 Full data for hybrid execution experiments on a single CPU-GPU compute node. For each image, the values are related to the number of compute GPU devices used, from 1-4 GPUs. 73
6.7 Application makespan of images BR-8682 and B6-A0X1 with related scaling efficiency for 1 to 4 GPUs used for hybrid execution. 74
6.8 Application makespan of images BR-8682 and B6-A0X1 with related scaling efficiency for 1 to 4 GPUs used for hybrid execution. 74
6.9 Application makespan of images BR-8682 and B6-A0X1 with related scaling efficiency for 1 to 32 nodes used. Image BR-8682 represents the cases of low background images while B6-A0X1 represents sparse images. 75
6.10 Application makespan of images BR-8682 and B6-A0X1 with related scaling efficiency for 1 to 32 nodes used. Image BR-8682 represents the cases of low background images while B6-A0X1 represents sparse images. 76
6.11 Full data for distributed execution experiments on CPU-only compute nodes. For each image, the values are related to the number of compute nodes used, from 1-32 nodes. 77
6.12 Partitioning results for image BR-8682. 32 partitions were generated for the execution with 8 nodes. 78
6.13 Partitioning results for image B6-A0X1. 32 partitions were generated for the execution with 8 nodes. 78
6.14 Workers workload for 16 CPU nodes configuration with BRB, ECB and KDT respectively, images BR-8682 and B6-A0X1. Execution times are sorted for better visualization. 79

6.15 Execution times in seconds for the partitioning algorithms on CPU-only compute nodes. For each image, the values are related to the number of compute nodes used, from 1-32 nodes. 80
6.16 Speedups for ECB vs. KDT. ECB with and without coarse-grain background removal shown. 81
6.17 Comparative data of ECB with vs. without coarse-grain background removal. 81
6.18 Partitioning results for images B6-A0X1 and BR-8285 using algorithms ECB and KDT. 16 partitions were generated for the execution with 4 nodes. 82
6.19 Workers workload for 32 CPU nodes configuration with ECB and KDT for image B6-A0X1. Execution times are sorted for better visualization. 82
6.20 Partitioning results for image B6-A0X1 using algorithm BRB with and without coarse-grain background removal. 16 partitions were generated for the execution with 4 nodes. 83
6.21 Normalized execution time of CADP using an error-prone cost-function against regular CADP. KD-Tree performance added as a baseline. 84

List of Tables

3.1 Related work comparative analysis. Each work is classified considering code availability, whether it is a fully fledged DSL or a set of compilation definitions [3], scheduling decoupling, target processor, distributed execution, and partitioning characteristics. 34

5.1 All possible cases of overlapping between two bounding boxes A and B with how they are resolved. By definition B is larger in area than A. 47

6.1 Information on all images used for the experiments. 67

6.2 Profiling of the pipeline's tasks on a serialized environment for Halide and Handwritten code. I/O times not considered. 67

6.3 Speedups of the GPU-only and hybrid CPU-GPU execution vs. CPU-only (48 cores) on a single node. A single V100 GPU was used for the GPU-only and Hybrid cases. 72

6.4 Speedups of BRB with vs. without coarse-grain background removal. 82

Chapter 1

Introduction

The ability to quickly analyze large datasets is critical to enable scientific studies in several application domains. Through modern digital microscopy technology it is possible to quickly obtain high-resolution Whole Slide Tissue Images (WSIs). Such images may now be captured at around 100K$\times$100K pixels with multiple channels from tissue specimens rapidly. These images are used, for instance, to assist in the analysis of several cancer types as they contain morphological information at cellular/sub-cellular levels that are known to correlate well with molecular and clinical data. These analyses can provide a better understating of underlying biological mechanisms, optimizing the selection of them as focused as therapeutic targets, and improve survival estimation [4].

There are several applications and studies that have used large imaging datasets available in the literature. On [5], a group of highly expressive regulators of tumor microenvironments for glioblastoma, a type of brain/spinal-cord cancer, were investigated. The correlation between them and a diminished survival rate indicated that treatments for such master regulators should be focused, as therapeutic targets. This analysis was performed using a total of 177 WSIs with 20$\times$ magnification, which are images in the order of 100K$\times$100K pixels [6]. Although the segmentation of regions of interest were manually done, removal of luminal areas and subsequent total tissue area calculations were performed using computer-based analysis. Another similar study also investigated glioblastoma, but with a more automated process [7]. There, 117 WSIs were classified on an oligodendroglioma-astrocytoma spectrum, which complements human-based pathologic review. Although the WSIs were annotated by domain experts, the goal of the work was to conceive an end-to-end automatic system, which returns valuable information based on the input WSI and simple annotations. This was achieved by segmenting the nuclei of 4K$\times$4K WSIs' partitions and extracting multiple features from them. These features underwent a selection process to filter the most impactful ones and were combined in a nuclear score (NS) on the oligodendroglioma-astrocytoma spectrum. The NS equation was trained through machine-learning methods. The work on [7] manages to improve end-user usability by using high-performance computing (HPC) to enable the use of such compute-demanding methods. Finally, at [8] an even larger experiment is done, with over 2400 40$\times$ magnification WSIs analyzed through 9879 quantitative extracted features. The main goal of this latest work was to improve prognosis prediction accuracy for adenocarcinoma and squamous cell carcinoma (lung cancer). Again, machine-learning algorithms were used to find the best features to distinguish short-term from long-term survivors.

As such, executing complex image analysis workflows with such imaging datasets is a costly task. The processing of a single WSI on a regular commodity computer may take hours. Consequently, studies that use datasets with hundreds to thousands of images would take several days to execute [9, 10]. This high computational cost is one of the obstacles for a broader use of large microscopy imaging datasets in clinical and research settings. As seen, the use of HPC can empower domain experts, enabling large-scale tasks which are unfeasible for humans, thus making them less laborious. However, as the scale of available hardware resources grows, so does the complexity of efficiently using such resources [9, 11].

The use of multi-core CPUs requires knowledge of thread-safeness and data-access patterns. GPUs, which have seen growing popularity [6, 12, 13, 14, 15, 16, 17], require even more complex understanding of the underlying hardware organization for its proper use. The cooperative use of both CPU and GPU can be even harder since the code for one compute resource may not be compatible with the other. This means that the same algorithm would need to be implemented twice, with one version for CPU execution and another for GPU. At last, there is a larger availability of clusters with multiple computing nodes at large-scale. Coordinating execution on distributed memory environments also has specific challenges, such as maintaining coordination of tasks and data locality. Thus, the development of such applications is a time-consuming and error-prone task with increasing complexity due to the fast hardware evolution.

The problem of efficiently developing applications for HPC systems can be particularly difficult if the application processing cost vary across different inputs, i.e., being content-dependent. For instance, on a WSI feature extraction process there may be more objects of interest (e.g., nuclei) in certain image regions, making those areas more compute-demanding. Not considering these characteristics during data-partitioning for workload distribution may lead to significant load imbalance and inefficient utilization of parallel resources. These problems are more difficult to solve for application domain scientists who do not necessarily have programming expertise on these conditions.

1.1 Problem Definition

Digital histopathology whole slide imaging analysis is computationally expensive by itself. This issue is worsen by large scale studies, which are both expected and common on the medical field. HPC solutions are a natural choice in attempting to reduce the execution time of these studies, making them more manageable. However, the efficient use of such resources is a complex task, requiring the understanding of the underlying devices, code generation for each device, and managing workload partitioning. The later, can be adversely impacted by histopathology applications with irregular processing cost, which can vary depending on the input data contents. Given these difficulties, widespread utilization of distributed and/or hybrid computing environments by experts in this application domain is still limited.

1.2 Research Hypothesis

It is the hypothesis of this work that high-level abstract programming coupled with automatic cost-aware workload partitioning methods can simplify the development effort of digital histopathology applications while enabling the efficient use of hybrid machines and distributed compute resources by medical domain experts.

1.3 Goals

The main goal of this work is to propose and develop methods for rapid and simple implementation of efficient WSI analysis applications targeting high-performance hybrid computing machines on a distributed environment. For such, existing algorithms and methods for leveraging high-performance hybrid computing resources are evaluated. From them, a WSI workflow processing solution with an automatic cost-aware workload partitioner is proposed. These methods should allow easy usage by domain experts of such HPC resources through high-level programming structures and semantics for parallel environments, automating most of the distributed environment management. In particular, the system here implemented using the proposed methods should (i) simplify the application development targeting hybrid machines with CPU and GPU compute devices and (ii) enable efficient cost-aware workload partitioning for distributed memory environments. The specific goals of this work are as follows:

Design and implementation of a WSI execution system on distributed memory hybrid machines

Efficient general use of hybrid resources on a distributed setting remains an open problem for digital histopathology. As such, these workflows are optimized on a per-application basis. Experts of the medical motivating domain should be able to easily access this cooperative resource pool through a unified framework. It should be opaque to the domain expert whether a compute node is composed by homogeneous elements (e.g., a single CPU) or heterogeneous elements (e.g., multiple CPUs and GPUs). Data management should also be automated, either across distributed elements or between hybrid resources on a compute node.

Proposal and development of irregular cost-aware data partitioning techniques for parallel execution of content-dependent applications

For some digital pathology applications the computational cost can vary according to the contents of the input data. This can pose as a problem for distributed execution of such applications since poor data partitioning can lead to workload imbalance and inefficient usage of distributed compute resources. To avoid workload imbalance on a distribution level across compute nodes, or on a node level across compute elements (CPUs/GPUs) irregular data partitioning should be supported. An initial analysis of current solutions shows that traditional image-partitioning algorithms (e.g., Fixed-Grid [18], KD-Trees and Quad-Trees [19]) are not suited for data with processing times dependent on the content. After the evaluation of more solutions already proposed, new strategies that take into account the content-dependent computations costs will be developed in order to improve the quality of the partitioning.

Evaluation of the proposed solutions with a real-world histopathology application on a large-scale compute environment

In order to validate the proposed techniques, these are experimentally evaluated with a well known real-world application, which has been used on past research [7, 9, 20, 21, 22, 23, 24]. The proposed algorithms are evaluated on a large-scale distributed environment with hybrid compute nodes containing CPU and GPU devices. The evaluation should include overall speedups of the proposed solution when compared with baseline approaches, their scaling efficiency and the state of workload imbalance.

1.4 Contributions

This work resulted in the following concrete contributions for large-scale WSI applications:

- An automated runtime system for simplifying the implementation and deployment of WSI workflows on hybrid large-scale compute environments.
- Cost-aware partitioning algorithms which are able to work with cost-dependent workflows by generating cost-wise balanced partitions.
- Solutions for both coarse-grain and fine-grain background removal, which results in lower execution times without impacting the given workflow output quality.

This work is a continuous effort on efficient execution of WSI analysis applications. Previously, the use of sensitivity analysis methods for identifying the most relevant image segmentation features were optimized [10]. There, the application workflow used was implemented, which lead to a better understanding of the problem approached here. Further, methods for reducing the compute cost of large-scale studies with sensitivity analysis through reuse were proposed, using the distributed execution system implemented on this work [24]. Finally, a previous state of this ongoing work describing implemented system with the first irregular data partition algorithm has been published [25].

1.5 Thesis Organization

The remainder of this work is organized as follows:

- **Chapter 2 [Background]:** introduces the background concepts of this work, the taxonomy used for HPC execution environments, the used tools and libraries. The motivating domain is explored, showing trends for current works.
- **Chapter 3 [Related Works]:** elaborates on spatial data partitioning algorithms. Related works and the current state of the art solutions for the specific goal of distributed execution of WSI applications are also explored.
- **Chapter 4 [Supporting Pathology Image Analysis Applications on Distributed Memory Hybrid Systems]:** describes the overall runtime system used for the execution of WSI workflows. The method for easing the implementation effort of domain experts for hybrid compute nodes is also approached.
- **Chapter 5 [Cost-Aware Data Partitioning for Irregular-Cost Applications]:** details the cost-aware partitioning algorithm proposed on this work. Both

workload imbalance and background removal are resolved by the proposed algorithms.

- **Chapter 6 [Experimental Results]:** describes the experimental analysis of the proposed solution with the partitioning algorithms on hybrid and large-scale distributed environments.
- **Chapter 7 [Conclusion]:** overviews the achieved results of this work, also approaching open questions and avenues for future works.

Chapter 2

Background

This chapter presents a taxonomy for defining distributed applications and the underlying hardware infrastructure. The main tools and libraries used or improved upon are then enumerated and described. Among them, two tools are highlighted, the Halide DSL and the Region Templates Framework. Finally, this chapter closes with an introduction to the wider research field of Whole Slide Imaging and histopathology and presents the motivating application used on this work.

2.1 Taxonomy for Distributed and Parallel Infrastructures and Applications

First, it is important to define the architectures for computer hardware (low-level) and systems (high-level) to provide consistent naming conventions. This distinction relates to the level of abstraction on which the parallel solution is implemented. Low-level architectures describe only hardware solutions, independently from any program, compiler or software on which it is executed. The naming conventions most commonly used for such systems are Flynn's [26] and Duncan's [1] taxonomies for parallel architectures. Flynn's taxonomy provides a model based on instructions and data streams, classified as either single or multiple, focusing only on hardware-level (or instruction-level) features. For Flynn, there are three main hardware architectures, Single Instruction-Stream Single Data-Stream (SISD), Single Instruction-Stream Multiple Data-Stream (SIMD) and Multiple Instruction-Stream Multiple Data-Stream (MIMD). SISD refers to serialized single-core computing elements with no parallelism. SIMD refers to any type of data-parallel architecture which runs the same instructions on a partitioned data set. Examples of SIMD are array vectorized and some types of pipelined architectures. MIMD is the most common parallel architecture, on which a set of independent computing elements can execute different instruction-streams on individual data segments. Both SIMD and MIMD classes are too inclusive, making it difficult to differentiate between different types of parallel architectures. This was pointed by Duncan [1], whose taxonomy is more detailed.

Duncan's taxonomy can be mainly divided into synchronous and asynchronous (MIMD) architectures, based on whether instructions are executed using a global lock-step, coordinated by a central control unit. The organization of architecture classes are shown in Figure 2.1, from his original publication. Unlike with Flynn's taxonomy, Duncan's better classifies SIMD sub-types. Pipelined Vector architectures refer to single-instruction specialized processing units. The parallelism is achieved through a pipelined execution of a single data-stream. For Duncan, SIMD architectures can be either Processor array of Associative memory. Processor array is the most common architecture, on which a single instruction is given to a group of processing elements, being executed on individual portions of an input data-stream. Examples of such architecture includes the SSE/AVX instructions sets [27, 28] and CUDA streaming processors, which execute on a locked-step fashion in each streaming multiprocessor [29]. Associative memory architectures inverts the von Neumann model, with operations executed by the associative memory unit [30]. Through the use of bit-masks, query operations are performed on large words on memory, which then execute bit-parallel comparison and logic operations. This type of architecture is mostly used on programmable hardware, e.g., FPGAs [31]. Systolic architectures can be defined as an abstract type of pipelined vector processors. This is composed by an array of interconnected specialized processing units which propagate computed results to the next unit on a locked-step synchronized manner [32, 33].

Duncan's MIMD architectures represent the most common types of parallel hardware architectures, with independent computing elements which can perform asynchronous tasks. Regardless of how such processing units are implemented, they can be classified by their relation to memory. Shared memory architectures provide direct hardware access to the same memory device. Modern multi-core CPUs are an example of such architecture, on which each core, independently from the other cores, can perform individual tasks on a shared memory device. Also, some GPU architectures can be described in a similar way. For instance, CUDA streaming multiprocessors also execute independently on the unique GPU's memory. For distributed memory architectures there are multiple memory devices, accessible only by the processing elements directly connected to them. Each of these shared memory elements (or groups) are interconnected externally and can only communicate explicitly through memory-passing operations. Distributed memory architectures are more sensitive to inter-process communication latency. However, they are far more extensible through the organization of clusters of processing elements.

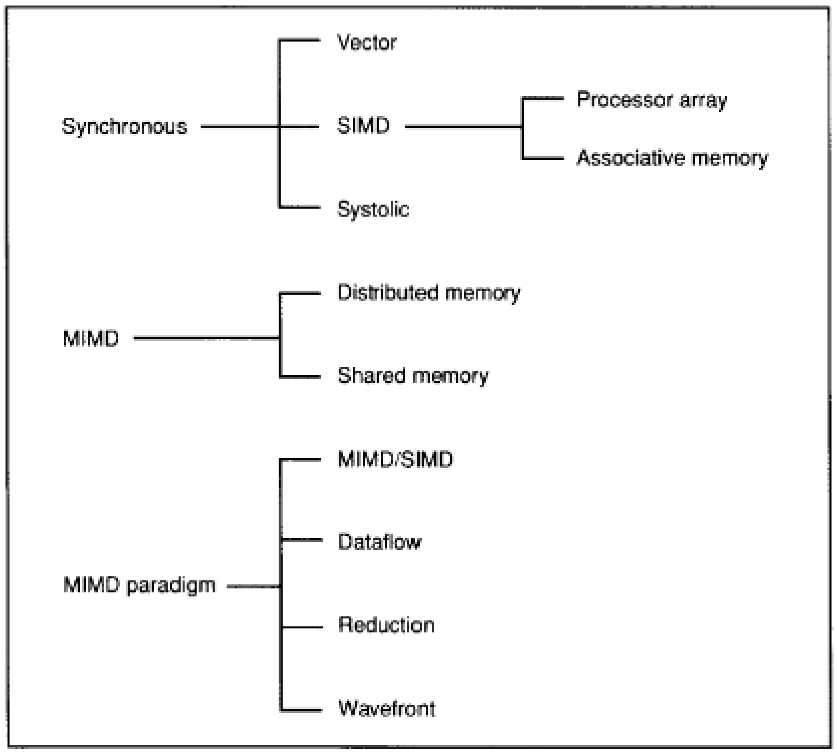


Figure 2.1: Duncan's taxonomy for parallel hardware architectures. Image extracted from [1].

At last, Duncan also describes a group of hybrid architectures which do not properly fit on the previous classifications, all of which being MIMD-based. One type of architecture which is widely present nowadays is the MIMD/SIMD hybrid architecture. Originally, this type of architecture was comprised of a primary MIMD organization of secondary (old slave definition) set of SIMD processors. Currently, processor array SIMD (SSE/AVX) processing units, or cores, are packed together on a single shared memory MIMD multi-core die. The hybrid organization of MIMD/SIMD raises the question of hierarchical classification of architectures. For instance, multiple distributed processors are a distributed memory MIMD organization of shared memory MIMD processing cores, which are in turn SIMD-processor-array-capable. This can become even more convoluted for hybrid CPU/GPU compute nodes. As such, either Flynn's and Duncan's taxonomy terms should always be accompanied by the layer on which said terminology is being applied to.

Other hybrid architectures are specific MIMD cases of the SIMD and Systolic types. Dataflow architectures are composed of a hardware and a software layer, specifically designed for such applications. The hardware layer is composed by a group of MIMD processing and instruction/memory elements, connected to a routing network unit, which enable dynamic definition of data paths between elements. From the software-level an application is defined as a dataflow of asynchronous task nodes. Each node is expanded to generate a data-path, similar to SIMD pipelined vector execution, with each end-node (task) being assigned to a processing unit. The execution of the dataflow application is data-driven, on which each finished node fires the subsequent nodes based on data dependency [34]. An alternative is to implement the processing unit allocation lazily based on the required values. This demand-drive model is defined as Reduction architectures [35]. Finally, the Wavefront architecture model is equivalent to a Systolic architecture for MIMD dataflow computing [36]. This means that different/irregular operations can be loaded for each processing element. Although interesting solutions, the Dataflow, Reduction and Wavefront hardware models are proposals of non-von-Neumann architectures not widely used.

According to Blank et.al [37], there are 3 modeling levels for designing parallel applications: (i) Algorithm Model, (ii) Programming Model and (iii) Machine Model. At the algorithmic level, parallelism can be achieved either through Data Parallelism (DP) or Control Parallelism (CP). For instance, synchronized execution of a partitioned data set, and an abstract FCFS thread-pool are respective examples of DP and CP. Algorithms are then reduced into a form closer to metal at the Programming Model level. These can either be Single Program Multiple Data (SPMD) or Multiple Program Multiple Data (MPMD) [38]. Since DP algorithms require implicit synchronization, only SPMD models can be applied (DP-SPMD). For CP algorithms, it is possible for it to be translated into CP-SPMD with multiple threads on the same program, or CP-MPMD with multiple programs. Previous definitions from Flynn's and Duncan's taxonomies cover the Machine Model. However, Blank's notation also refers to memory access for the following cases:

- Shared memory, Single Instruction, Multiple Data Streams (SIMD)
- Shared memory, Multiple Instruction, Multiple Data Streams (sMIMD)
- Distributed memory, Multiple Instruction, Multiple Data Streams (dMIMD)

As such, the naming conventions extracted from Blank's taxonomy are used on this work. This is mainly for simplicity, also noting that common modern machine architectures [39], such as the ones used on this work, can be fully described by his notation.

2.2 Parallel Programming Tools and Languages

Specific tools and frameworks are required to implement algorithms which can fully utilize the distinct resources available on modern distributed environments. For this work there are three main considerations which should be addressed: executing on (i) distributed environments, (ii) multi-core CPUs, and (iii) GPUs. The MPI standard was thus chosen, with OpenMP and CUDA used for parallel CPU and GPU execution, respectively. These tools and languages are detailed on the following sections.

2.2.1 The Message Passing Interface (MPI)

The Message Passing Interface (MPI) can be defined as a tightly constrained set of simple goals for Inter-Process Communication (IPC) on non-shared memory spaces. Such memory spaces can be either on the same memory hardware, or on distributed hardware. Communication between these can then be respectively performed by IPC or network sockets. MPI enables parallel programming through the use of multiple processes, either locally on the same compute node, or distributed across a network of compute nodes. The semantic for either case is the same. By definition, MPI (i) is an open interface which could be implemented by many vendors, without impact to the underlying communication and system software, (ii) allows thread-safeness, (iii) abstracts the communication layer, and (iv) allows convenient C and Fortran bindings while having language-independent semantics [40]. This resulted in MPI being a system and architecture agnostic interface, responsible for standardizing inter-process communication for programming on dSPMD and MPMD machines. From a given space of processes, each and any MPI process can send and receive point-to-point messages from other processes of the same space. Communication calls (send/receive) can be thread-blocking, with further support for thread-blocking barrier primitives.

Regardless of the implementation, execution of parallel applications on MPI can be configured, with the spawned processes scattered locally or across a network. It supports both SPMD and MPMD executions. For MPMD it enables the creation of a shared message space for more than one executable instance. Another important feature is the ability to specify thread-affinity and processes distribution. This is possible due to an abstract definition of processing units (PUs), which can be specified by the user. For instance, a dual-socket 8-core CPU compute node can configured as a single PU on *node* level, two PUs at *socket* level, 16 PUs with one per core, or an user-defined number of cores per PU. This configuration is important since each PU have an enclosed space for the process' threads. This means that a thread of a given space (and thus process) cannot utilize resources (i.e., cores) belonging to another space. It is also possible to remove all bindings, thus enabling shared use of all local resources by any thread on the given node. Finally, each created process can be distributed (mapped) across PUs on a user-defined policy, being possible to oversubscribe a PU (i.e., more than one process per PU).

2.2.2 OpenMP

As MPI became the standard for parallel high-performance scalable code, multi-core architectures began to arise, and with it the possibility for optimizing cache-coherent hardware. The message-passing model scalability relies on the developer, since it does not support high-level directives, and also cannot optimize cache locality. For this new Scalable Shared-Memory Multiprocessor architecture (SSMP) a new programming model/interface was created, the OpenMP. As with MPI, OpenMP was specified to have native Fortran/C bindings, but in contrast, defining high-level compiler directives which could be used for incremental parallelization of code [41].

OpenMP is used through compiler directives for structured blocks of code which allow them to be executed parallelly either with threads on a shared-memory space or with aSIMD instructions. These parallel blocks can be both loop iteration spaces or distinct sections of code to be executed concurrently. OpenMP also supports the specification of private and shared variables on its structured blocks, synchronization directives through barriers, critical sections of code and atomic operations, and other hierarchical task structures on its latest versions [42, 43].

2.2.3 CUDA

In the early days, GPUs were predominately used for fast image and video rendering, for instance, in games. Then, GPU cards were mostly required to process integer values (fixed-function pipelines [44]). Even with the GPU limitations at that time, initial attempts to harness its computational power were made on linear algebra applications [45]. Following the growth of the games industry better GPU technology was developed to support richer and more realistic graphic effects, eventually resulting in the introduction of floating-point arithmetics [46]. Later, the introduction of programmable shaders and floating-point operations [47] solved fixed-point overflow-related limitations. This enabled the scientific community to implement more complex linear algebra operators [48] and even manage to outperform CPUs on a general-purpose task for the first time ever [49]. Still, support for general programming was limited, requiring the use of graphics APIs such as Microsoft's DirectX or OpenGL [50]. The continued interest on using GPU devices for general computing resulted in the introduction of tools as OpenCL [51] and CUDA [52].

CUDA is a proprietary framework for developing efficient general-purpose applications on modern NVIDIA GPUs. Compared with regular CPU computing these GPGPUs distinguish themselves by having a high count of simple computing units with increased memory locality and bandwidth. CUDA devices are a hierarchy of compute units, as demonstrated on Figure 2.2. Each *core*, defined as a Streaming Processor (SP), executes

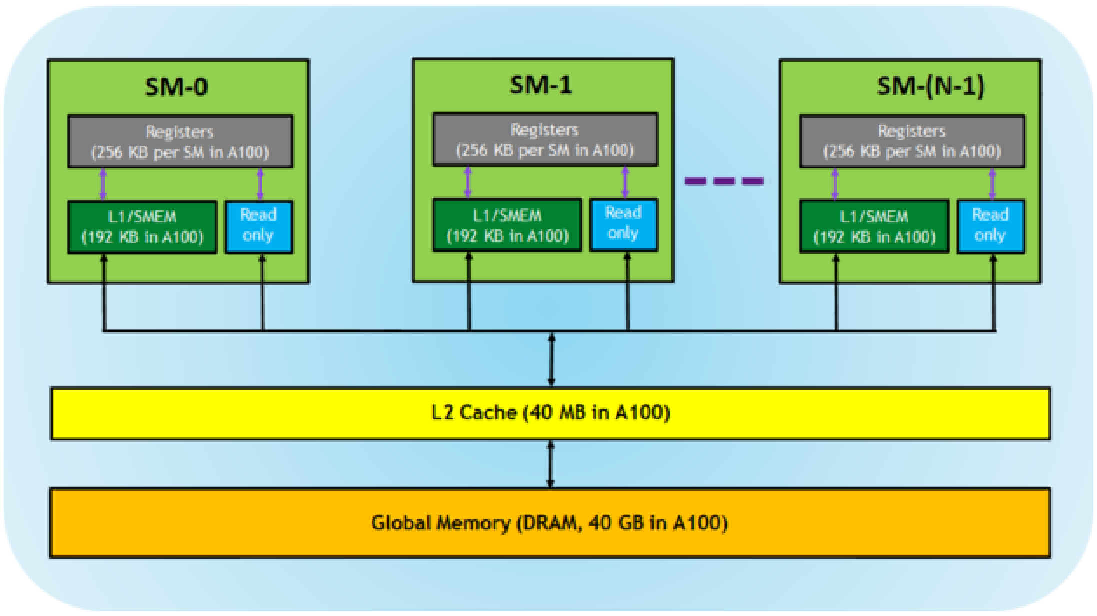


Figure 2.2: CUDA memory hierarchy and hardware organization. Image from [2].

a thread individually and has direct access to a private registers space. A group of SP is assembled in a Streaming Multiprocessor (SM). Each SM possesses a local L1 cache unit, shared among the SP which compose the SM. A GPU is composed by a set of SM, all sharing a global L2 cache.

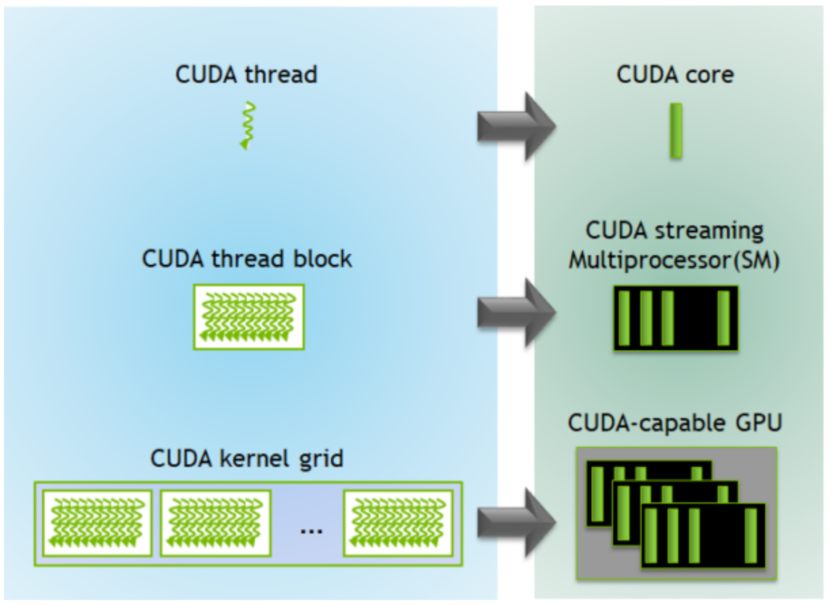


Figure 2.3: CUDA organization of execution units, or threads. Image from [2].

Figure 2.3 shows the organization of executable code on CUDA. The minimal unit of work is a thread, which is executed on a single SP. A group of threads, or thread block, is scheduled on a single SM. Since the number of SP of a SM is fixed, there is a limit of how many lock-step threads can run in parallel. Each SM groups a subset of threads from its thread block into warps, scheduled by the SM which owns them. At the top level, the programmer implements kernel grids, which are executed on a whole GPU. Although further hierarchical levels may be present on different devices, these changes only regard the organization of warp schedulers, cache and operation-specific cores, not influencing the programing interface of SMs and SPs. For multi-GPU configurations this work considers each device independent, with its own memory space, on a dMIMD environment. Also, communication is only performed with the CPU via the PCIe interface [29], although other faster interfaces are available [53].

2.3 Frameworks for High-Performance Image Processing

The parallel programming tools presented on previous sections can be combined into higher-level frameworks for image processing. One tool developed to reduce implementation complexity for high-performance image processing applications is the Halide DSL. Halide employees OpenMP and CUDA on its back-end. Also, the Region Templates Framework, a tool for distributed execution of medical imaging pipelines, is preferable instead of just MPI, since it has a higher-level abstraction of inter-node execution. Finally, both frameworks use OpenCV as their image data objects, detailed on the following sections.

2.3.1 OpenCV

With the purpose of easing the development of computer vision applications, OpenCV was launched in 1999 by Intel. The main goals of OpenCV were to provide a common, free, basic infrastructure for vision research. This interface would be one of portable code, optimized for different platforms. Some applications which benefits from OpenCV are stereo vision, object detection, segmentation and recognition. OpenCV is regarded as the main standard for image processing in general, extensively used by C++ and python applications [54].

The base structure of OpenCV is a *cv::Mat*, which represents a lightweight n-dimensional dense numerical array of an user-defined type. The data inside a *cv::Mat* can be accessed directly, or through pointer arithmetic. Regarding memory, these matrices are handled with high-level mechanisms such as reference counting and default shallow copying, enabling easy and implicit memory management. Image pixel data is released only when all *cv::Mat* references are destroyed. Since images can be composed of multiple channels, OpenCV provides a simple syntax for color images, also allowing the use of different colorspaces. Multiple matrices can be combined with high-level operators like sum, dot-product and regular matrix product, which over-saturates the matrices' types on the resulting array. It is possible to perform efficient masked operators for applying such equations on defined regions of interest. OpenCV provides a comprehensive set of morphological operations, which ease the development process of efficient image processing applications. Some of these high-level functions used on this work are erosion/dilation with different structuring elements and retrieval of connected components' bounding boxes [55].

Although GPU processing is available to OpenCV, processing of such data can only be performed by either OpenCV's pre-defined data processing functions or a CUDA kernel. Direct pixel data access is only allowed inside such CUDA kernels, which allows the execution of user-developed operations on GPU data. As such, the only way to share data between CPU and GPU memory spaces is through explicit data transfer.

2.3.2 Halide

On a higher level of abstraction, Halide [56] is a DSL aimed at enabling transparent implementation of image processing pipelines through different hardware environments, or targets. Its main contributions were to provide an environment on which the implementation and optimization of algorithms were done separately, thus enabling the use of the same algorithm for multiple hardware targets. This feature eases the implementation effort, on which the developer needs only to focus on correctness to later tackle the performance through scheduling directives. Given the vast combinations of available hardware resources, the Halide model encourages the programmers to find the best schedules empirically through few (and simple) directives. It is also worth noting that, although Halide was developed for manual scheduling its semantics are representative enough that there is extensive work on automatic scheduling systems [57, 58, 59].

The API and Syntax

Halide code is comprised mainly of the algorithm definition and its schedule. The algorithm is thus a set of functional definitions, parameterized by its input domains, which operate on buffers, constant values and other functional definitions. The schedule is a set of primitives which parallelizes, orders, or defines memory patterns for each functional definition on a given domain. For images, each domain can be seen as a dimension of that image (e.g., height or width), and a functional definition can be defined as a stencil to be performed once at the image, being possible to compose multiple of those in a workflow.

Internally, Halide uses interval-based domains to represent iteration spaces, i.e., every domain is defined as a triple of a label, its initial value and its end value. This means that creating non-rectangular iteration spaces can be somewhat difficult, if not impossible for some applications. The alternative would be to use the polyhedral model for describing them. However, using interval-based domains is significantly easier for entry-level developers to use while also being able to better evaluate implementation correctness at compile-time, moving many errors away from execution time [60].

Halide algorithms' implementations mainly use 5 types of directives:

- *Var*: represents an abstract iterative domain
- *RDom*: represents a static iterative domain
- *Func*: represents a stage of a pipeline parameterized by its domain coordinates
- *Expr*: represents a pure abstract function of *Func*'s or domain variables
- *Buffer*: a concrete in-memory representation of data, which can be used as input and/or output

Var's are the base elements for iteration. They represent an abstract domain, in which its concrete interval is inferred at run-time. This allows the flexibility of implementing pipelines for any-sized input images. It is worth noting that although the domain sizes are abstract its structure is not. Thus, the image must have the correct dimensionality for execution. It is also possible to define static sub-domains, called Reduction Domains. For statically-defined access patterns, which would require exhaustive definition, Halide provides *RDom* variables. For instance, the structuring elements for image processing algorithms like erosion/dilation [61] with large static values (e.g., 50×50) would either require the runtime iteration of a *Var* or the handwritten definition of all coordinates. *RDom*'s allow the creation of a static domain variable which will be resolved (unrolled) at compilation time, thus not impacting negatively the execution time.

A Halide *Func* is the base building block for creating pipelines. They can operate on an abstract input domain, representing a pixel or any type of data on a k-dimensional domain, assuming there are k input domain variables. They are defined as a pure function of other *Func*'s, domain variables, constants, input Buffers and *Expr*'s. It is also possible to re-define a *Func*, defined by Halide as a function update. For these, each update is fully executed to completion individually and in the order of their definition. It is also possible to schedule each update function individually.

As a tool to improve code legibility, Halide provides *Expr*'s, which are *Func*'s without any input domain. Whenever an *Expr* is referenced on a *Func*, Halide fully replaces its content on the target *Func*, acting like a label on a purely functional language [62]. Finally, concrete user inputs and outputs are represented by Halide's *Buffer*'s. These are strongly-typed, memory-resident data which size must be known before executing on them. From this input/output structure Halide infers *Func*'s typing and domain size. It is worth noting that Halide uses a strong type system.

An extended example of the *blur* algorithm is presented on Algorithms 1 , 2 and 3. First its behavior is shaped by the pure algorithm, to later be scheduled for both CPU and GPU though CUDA. The *blur* algorithm implemented have a structuring element of size 3 $\times$ 3, as seen on line 3 of Algorithm 1, on which each point of the input image (*in*) is blurred around the intervals $[x-1, x+1]$ and $[y-1, y+1]$. Each dimension (or domain) is blurred individually to ease the optimization process. The final result is stored at buffer *out* on line 9, as the realization of function *blury*, which in turns depends on *blurx*. By using the structuring element *se* we are able to parameterize it, if necessary, and with the use of the *sum* aggregator function, avoid having to expand the terms of lines 6-7 manually. Otherwise, for a structuring element of size 11 $\times$ 11 we would be required to change the algorithm (lines 6-7) to *blurx(x,y) = (in(x-5,y) + in(x-4,y) + ... + in(x+5,y))/11*.

Algorithm 1 Example of a blur algorithm implementation.

```
1: Halide::Buffer $in$, $out$
2: Halide::Var $x$, $y$
3: Halide::RDom $se(-1, -1, 1, 1)$
4: Halide::Func $blurx$, $blury$
5:
6: $blurx(x, y) = sum(in(x + se.x, y))/3$
7: $blury(x, y) = sum(blurx(x, y + se.y))/3$
8:
9: $out = blury.realize()$
```

This algorithm can then be scheduled for either CPU or GPU execution. Algorithm 2 presents a simple CPU parallel execution schedule. The scheduling process can be divided into three steps: (i) internal tiling/reordering, (ii) inter-functions ordering and memory allocation, and (iii) parallelization. Each function is initially bounded by its original domains (x and y for both functions). The ordering is from left to right, meaning that x is the innermost level of iteration. Each initial domain can be split and tiled, as is y for *blury* (see lines 1-2 of Algorithm 2). This results in the concrete iteration spaces for *blury* of *yo*, *yi* and x, from outermost to innermost levels. The next step regards execution order of points between functions with producer-consumer relationships. Still on Algorithm 2, line 4 states that *blurx* will be produced on-demand, for each point of *yi*, i.e., immediately after *yi* iterates, all required points of *blurx*, by *blury* at a *yi* value, will be calculated beforehand. Then, on line 5, we attempt to reduce redundancy of computation, storing the calculated values of *blurx* on the scope of each *yo* value. This means that all values of *blurx* which are required by *blury* on the full iteration space of *yi* $\times x$ are stored temporarily, and later discarded for the next value of *yo*. Finally, *blury* is set to be calculated fully and stored locally before any consumer can access it (line 6). Since *blury* is the last function which is realized, this last *compute_root* declaration is implicit, placed there only for illustration purpose. Regarding parallelization, *blurx* is vectorized by a factor of $VECT\_SIZE$ on coordinate x (line 8), and *blury* is executed parallelly on coordinate *yo*, meaning that there is a single thread for every *yi* $\times x$ iteration space. It is worth noting than, in the absence of scheduling directives, every function is scheduled as a serialized execution, with *compute_root* stages on every function.

Algorithm 2 CPU Scheduling of the blur algorithm example.

```
1: Halide::Var $yi$, $yo$
2: $blury.split(y, yo, yi, PARALLEL\_BATCH\_SIZE)$
3:
4: $blurx.compute\_at(blury, yi)$
5: $blurx.store\_at(blury, yo)$
6: $blury.compute\_root()$
7:
8: $blurx.vectorize(x, VECT\_SIZE)$
9: $blury.parallel(yo)$
```

For the GPU schedule, Algorithm 3 is a similar but slightly modified version of the CPU schedule due to the differences between CPU and CUDA execution environments. For CPU execution each operation can execute parallelly on sMIMD or through vectorization on each SIMD core of the CPU. For CUDA GPUs, aSIMD execution is implicit through tasks inside warps (*gpu_threads*) and sMIMD execution is performed by SMs (*gpu_blocks* for Halide). Also, given the memory hierarchy of GPUs, it is recommended to perform computation in rectangular regions. For such, Algorithm 3 changes are, (i) the *blury* domains are fully tiled on both x and y according to the configuration of the used GPU (lines 2-3), (ii) *blurx* is computed at the innermost level for each *blury* value (lines 5-7), and (iii) both *blurx* and *blury* are parallelized using the GPU directives (lines 9-10). With these changes the GPU schedule works similarly to the CPU schedule, attempting to reach a middle-ground on parallelization, redundancy reduction and memory footprint reduction.

Algorithm 3 GPU Scheduling of the blur algorithm example.

```
Halide::Var $xi, xo, yi, yo$
$blury.split(y, yo, yi, height/NB\_CUDA\_SM)$
$blury.split(x, xo, xi, width/NB\_CUDA\_SP\_PER\_SM)$

$blurx.compute\_at(blury, xi)$
$blurx.store\_at(blury, yo)$
$blury.compute\_root()$

$blury.gpu\_blocks(xo, yo)$
$blury.gpu\_threads(xi, yi)$
```

Halide in Depth

The Halide DSL uses the LLVM toolkit for creating its own internal representation language. Originally, LLVM was designed as a framework of tools for transparent, life-long, code analysis and transformations [63]. Currently it has evolved beyond its initial framework to include a suite of sub-projects, being the most popular, Clang [64]. The main goal for LLVM is to provide an Intermediary Representation (IR) language, which can undergo its internal analysis and transformations. This IR is language-independent and have strongly-typed Static Single Assignment (SSA) definitions [65], which further improves its analysis capacity. Further, LLVM provides plenty of target-specific back-ends for executing its architecture-independent IR objects (e.g., x86, CUDA, OpenCL). These features make LLVM an interesting choice for creating compilers and DSLs.

On pipelines' compilation process of Halide, these are converted into Halide's IR language, based on LLVM's IR. In the compilation process, being it Just-in-Time (JIT) or Ahead-of-Time (AOT), the IR is consecutively lowered, from its initial high-level representation to a more hardware-specific (or by Halide's terms, target-specific) code, initializing on the top-level loop. Each lowering passing includes runtime sanity checks which ensures that the compiled pipeline is executable. This is required for Halide's correctness and stability guarantees. Thus, any schedule which may violate the assertion that the output must be the same, independently of the execution ordering, fails to compile before actually running. An example of this would be parallelizing a function on a domain which depends on the previous value, e.g., $sum(x, y) = sum(x - 1, y) + input(x, y)$ with a $parallel(x)$ schedule. Since every value $sum(x, y)$ can only be calculated after $sum(x-1, y)$, it cannot be parallelized on domain x. This could be fixed by changing the parallelization domain from x to y. Further, the runtime sanity checks prevents the execution of inputs which cannot be properly executed. For instance, It is impossible to execute a *blur* filter with a large structuring element (e.g., 20×20) on an image smaller than the structuring element itself (e.g., image is 15×18 in size). Attempting to perform such executions would result in a runtime error, with Halide stating the problem.

Currently, Halide supports a number of architectures, or targets, including but not limited to x86, ARM [66], MIPS [67], and CUDA [29]. Code generation to multiple hardware devices is facilitated by the use of LLVM [63]. Halide employs a Visitor Pattern [68] on IR elements as a way to decouple the high-level abstraction from hardware specific code. This improves Halide's extensibility for adding new targets. On the compilation of the Halide library, each target implementation of these high-level objects (e.g., *Func*'s and *Var*'s) is compiled as LLVM IR fragments with the resulting bitcode put together as constant strings. When compiling a Halide pipeline, either JIT or AOT, Halide deserializes the required components for a given target and combine them with the lowered pipeline on a single LLVM module. This module is then compiled to the target-specific machine code to either be used at once for JIT compilation, or outputted as a runtime static library. It is worth noting that for JIT compilation, the compiled module is flexible in a way that it may be updated dynamically after the initial compilation without requiring a whole new runtime to be created. This is possible through the decoupled use of individual bitcode strings.

Halide Limitations

Although Halide was created with the separation between algorithm and schedule in mind, some more sophisticated optimizations may require the algorithm to change. This happens for problems with more complex inter-stage dependencies and data access patterns. For instance, the *blur* algorithm was implemented in a two-step architecture to facilitate scheduling, while a single-step implementation was possible. Still, being with schedules independent from the algorithm the debugging process remains an easy task. Only later the pipeline can be optimized through the same trial-and-error process Halide advocates. Further, Halide suffers from having runtime-only errors on schedules, as opposed to compilation-time errors. This is due to the runtime soundness checks which cannot be performed on schedules at compilation time, and results in scheduling code which is harder to debug.

Halide is used mainly for image processing problems, but can also be used for stencil operations. Regarding complex neural networks, Halide is not the most recommended tool since it does not support cyclical pipelines. This is required for Long Short-Term Memory architectures in which a number of steps, unknown at compilation time, is required [69].

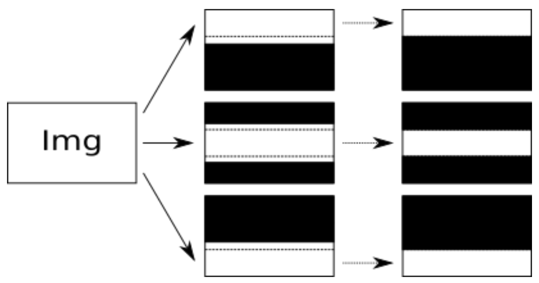
Given that Halide was implemented for shared memory targets only, it does not natively support any sort of CPU-GPU cooperative or multi-GPU execution, nor it supports distributed environments. It is possible however to use Halide as a sSPMD execution tool and add support for distributed memory manually through MPI. This strategy can however be cumbersome since efficient MPI/CUDA programming is not a trivial problem. Although some work has been done to improve Halide's usability for hybrid or distributed computing [60, 70, 71], cooperative CPU-GPU execution is, to this work publication date, still unavailable for Halide.

Finally, Halide is also limited by its iteration model, which does not support more complex irregular spaces. Also, Halide's static analysis model requires that pipelines must be defined as DAGs, not allowing dynamic loops which size are only known at runtime. By known at runtime it is meant the Halide pipeline internal execution. For instance, it is legal to have a pipeline with domains' sizes defined only at the pipeline execution, however, it is impossible to iterate a given function a number of time which is unknown or content-dependent. This limitation can be surpassed with polyhedral DSLs as seen next [60].

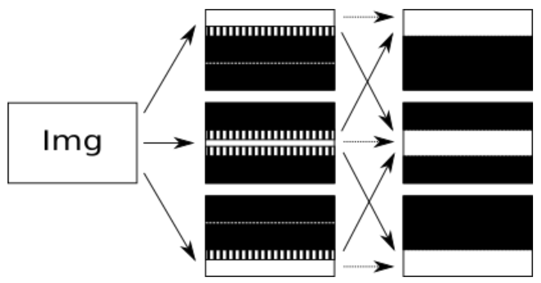
Halide Extensions

With regards to Halide's lack of cooperative CPU-GPU execution capability, Liao et al. [71] propose an extension which partitions the input domain with a user-defined size parameter to later enable cooperative execution. Boundaries of the two partitions are resolved through redundant computation of borders with regards to existent data dependencies. In contrast to a naive approach of manually partitioning the input image for later concurrent Halide execution on both targets, the proposed work avoids unnecessary memory transfer operations by joining the output buffers into a single output by performing in-place operations on CPU targets and copying the GPU output data directly to the final output buffer. In order to reduce load imbalance the input domain can be partitioned in more than two parts. The execution model is based on a one-dimension array of to-be-executed partitions. This array is consumed from both ends towards the middle by both a CPU thread and a GPU thread on each opposing end. This approach results in a flexible model for cooperative CPU-GPU execution, however, not enough to enable the use of multiple CPU or GPU devices (or threads).

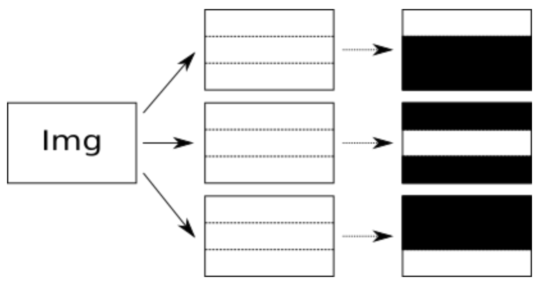
The work of Denniston et al. [70] focused on adding distribution support for Halide by adding a new scheduling level for distributed computing with a tradeoff of redundancy vs communication. This is done through the introduction of two new scheduling directives: *distribute()* and *compute_rank()*. Using a simple producer-consumer pipeline as an example, e.g., *blurx* produces for *blury*, it is possible to represent four points on the redundancy/communication tradeoff: (i) local and global redundancy, (ii) local redundancy and no global redundancy (through communication), (iii) no local redundancy and massive global redundancy, and (iv) no local redundancy and border global redundancy, without communication. These cases are depicted on Figure 2.4. For all cases *blury* is scheduled for distributed execution across the y domain. The first case can be achieved through *blurx.compute_at(blurx, y)*, meaning that for every y value of *blury* an entire *widith* $\times 3$ area of *blurx* is calculated beforehand. This results in local redundancy between every value of y and global redundancy on the intersections between computed regions of *blurx*, as seen on Figure 2.4a. By doing *blurx.compute_root().distribute(y)* the global redundancy is replaced by communication of the bordering regions between partitions on y (see Figure 2.4b). Local redundancy can be removed completely by scheduling *blurx.compute_root()*, which also results in massive global redundancy, since *blurx* executes on the whole image on each distributed space, as seen on Figure 2.4c.



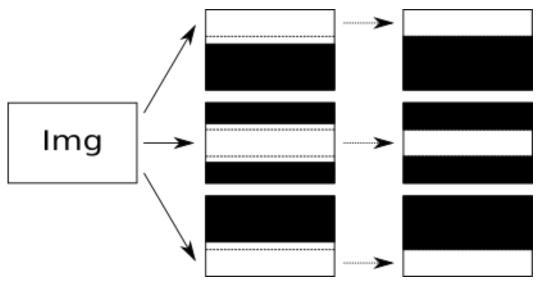
(a) Scheduling case (i): local and global redundancy.



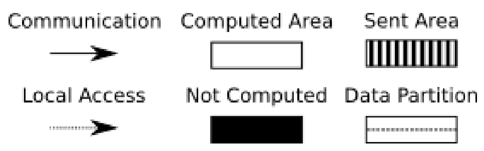
(b) Scheduling case (ii): local redundancy without global redundancy. Dashed areas are communicated between distributed spaces.



(c) Scheduling case (iii): No local redundancy and massive global redundancy.



(d) Scheduling case (iv): No local redundancy with small global redundancy. Overlapping regions are defined as *ghost zones*.



(e) Caption of symbols used to represent computation, communication and access to local memory.

Figure 2.4: Illustration of possible scheduling tradeoffs for the Distributed Halide schedules example of the blurring algorithm. Each line of blocks is processed on a distributed setting, with the first column being the *blurx* function and the second *blury*.

For this particular example the best schedule is achieved by (iv) with *blurx.compute_rank()*, which removes any local redundancy and minimizes global redundancy without requiring communication between distributed spaces. This is possible through the *compute_rank()* directive, which infers a *ghost zone* [72] required by *blury* and executes this extended partition locally, simulating a *compute_root()* schedule on a distributed environment.

The Tiramisu compiler was recently introduced as an extension/specialization of Halide, with novel commands for distributed environments [60]. It extends Halide with a polyhedral syntax with directives to support distributed execution and explicit management for data storage and movement. However, Tiramisu does not support JIT compilation and parametric tiling of images, i.e., the input size must be known at compilation time. As such, a compilation for each data input size is required and must be known AOT. The scheduling directives are compiled into a Tiramisu-specific IR, which wraps Halide's IR for local scheduling, and MPI for distribution. Thus, Tiramisu supports all of Halide's directives. Regarding actual coding, Tiramisu represents its polyhedral domains as text strings instead of having actual C++ direct bindings (as with Halide for instance). This further exacerbates the already present problems of runtime/compilation-time errors by also having text strings as high-level entities, which are not validated on compilation-time.

There has been extensive work on automation using the Halide DSL. Most predominantly, there has been a focus on automatic schedulers (auto-schedulers) [57, 58, 59], which are available for a diverse set of targets. This particular problem is considered rather hard by itself given the large search domain for these algorithms on deeper and more realistic pipelines. Also, automatic low-level code to high-level Halide translation has been applied to over 260 image processing functions on Adobe Photoshop while also achieving speedups with auto-schedulers on generated code [73].

2.3.3 The Region Templates Framework (RT)

The Region Templates (RT) [74] is a runtime system designed to execute large-scale dMPMD image analysis applications. It allows for applications to be described as a hierarchical workflow, where coarse-grain workflow stages may be implemented as a workflow of fine-grain operations. The overall system architecture is presented in Figure 2.5. The system uses the Manager-Worker model, where a single Manager process (system-wide) maintains a queue of stage instances to be executed. Stage instances are assigned for processing on Workers in a demand-driven basis as their workflow dependencies are resolved. Each Worker is responsible for internal scheduling of fine-grain tasks on available local resources.

RT also abstracts the data storage layer from the application programmer. In RT the communication between coarse-gain stage instances is carried out by writing/read-

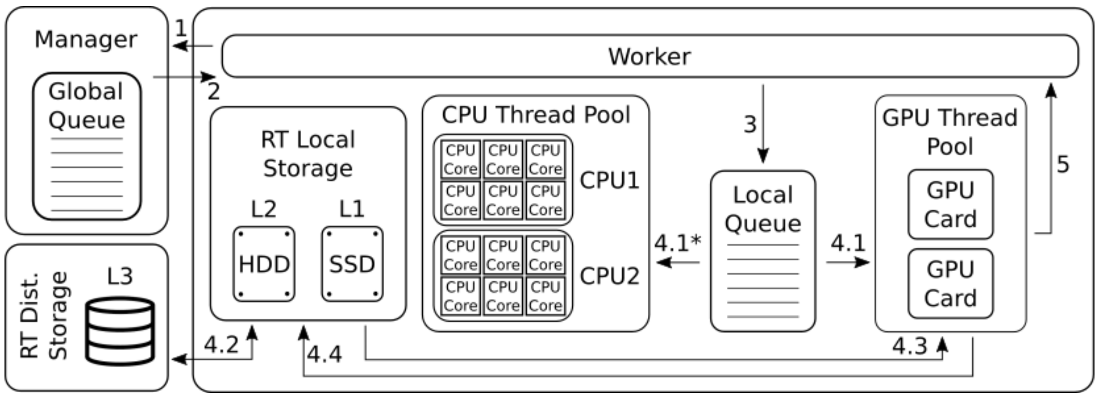


Figure 2.5: RTF architecture and workflow execution steps: (1) Worker queries Manager for stage instances created by the application workflow, (2) Worker consumes instances for processing, (3) Fine-grain tasks created in each Worker are assigned for execution with CPU and/or GPU, (4) Data is read, tasks processed, and results written to storage, and (5) Worker is notified of the end of a stage instance execution.

ing to/from region templates data elements, instead of performing explicit inter-process communication. This simplifies the application development and also enriches the system with data placement awareness, which may be used to improve data locality during stage instance scheduling.

Globally, RT uses MPI for inter-node (or inter-Worker) communication and thread affinity configurations on a per-node scope. Workers are composed of thread pools to manage available devices as CPU and GPUs and a have access to the storage layer. A single Worker may be created per computing node as it is able to use of all available computing devices. On a given distributed system, each computing node has a single Worker process, with a single Manager process globally. For heterogeneous computing nodes, each distinct resource on a shared memory space represents a single schedulable target on a Worker. For instance, a computing node with dual-socket CPUs and two GPUs have a single CPU and two GPU schedulable resources.

The execution steps when using RT are detailed in Figure 2.5. The Manager creates a local queue of stage instances to be executed, and Workers will consume and process them. This request is performed through a message (1) sent by the Worker, which is replied (2) by the Manager with metadata describing the instance(s) it should process. The Worker then (3) creates the fine-grain tasks that represent the processing of the received stage instances and inserts them in the local queue. Once (4.1) a task is dispatched for execution with a CPU or a GPU, (4.2) the input (or intermediary) data are read, (4.3) the processing takes place, and (4.4) output data is written to the RT storage hierarchy after execution. It is worth noting that data movement is performed dynamically on-demand when a Worker process requests data that may be on another node. After each task is completed, (5) a callback function is executed to notify the Worker about the task's completion. Further, when all of the fine-grain tasks for a given stage instance (created in 3) have finished, the Manager is notified along with a request for new stage instances (1). This process continues until there are no stage instances left to be processed.

The distributed storage hierarchy enables exchanging data among stage instances executed in different Workers or computing nodes. Local and distributed storage are part of a single hierarchical infrastructure, and search for the requested data starts in the first layer of the storage (local and faster) and follows until it is found, for instance, in the distributed storage. The user can configure which storage devices are used in each layer of the hierarchy, the amount of space available, and the data replacement policy. For instance, as illustrated in Figure 2.5, one could configure the system with a three level hierarchy having two node local levels L1 and L2 using, respectively, SSD and HDD, and a distributed storage in L3. All data are stored in RT data containers, which are spatial-temporal data representing 2D/3D regions with a temporal component. Each RT data unit may contain multiple Data Regions (DR), for instance, storing different measures of the same spatial location. Typical data structures that a DR may store include vectors, matrices, and polygons. The availability of such data structures improve development productivity as data are available to the application in common representations used in the domain. While RT presents all the features described above, it does not simplify the development of stages or tasks code that execute the application transformations.

2.4 Motivating Field of Research

There is a widespread adoption of Whole Slide Imaging (WSI) solutions for digital histopathology applications [75]. Through the use of powerful slide scanners it is possible to extract high-resolution images, which can be used for helping medical professionals [13]. These automatic slide loaders are able to quickly retrieve large numbers of WSIs. With these, tools and techniques for Computer-Assisted Diagnosis (CAD) are being developed, which promises to improve on the effort required by medical professionals by automating laborious tasks [76]. Another interesting application is related to Content-Based Image Retrieval (CBIR) related to histopathology [13, 77], on which large databases of WSIs can be queried for image content. Finally, there are also research efforts to recognize patterns of diseases and classify them, how patients may respond to treatment or to estimate their prognosis also benefit from WSI usage [78, 79, 80, 81, 82].

All mentioned applications and fields have one aspect in common, its use of WSIs. These WSIs are high-resolution images, sometimes reaching over $100,000 \times 100,000$ pixels of resolution on higher magnifications [6]. These images are aggressively compressed since they could reach over 30 GB of size for an uncompressed color image. Also, these are available in a pyramidal representation, on which the same image is available at different magnifications (and thus resolutions). Further, each patient may have multiple tissue slides, and data from multiple patients are required for larger-scale studies [21, 20, 13]. These characteristics compound, increasing the computational cost of performing such studies, making it necessary the use of high-performance resources. There are currently open databases for such WSIs, with the images used on this work being provided by The Cancer Gnome Atlas project (TCGA) through the Genomic Data Commons Data Portal (GDC) [83, 84].

For such large-scale studies the use of High-Performance Computing (HPC) solutions is a natural choice. The usage of such solutions have been more widely employed for large-scale WSI studies [6, 13, 14, 15, 77]. However, most related works for such studies have only employed the use of GPUs, on some cases using more than one GPU on a single node. The use of distributed environments is mostly found on works focusing HPC solutions instead of histopathology or cytology [7, 21, 22, 23, 24, 25, 74]. Usage of such resources at a larger-scale should then be a goal of histopathology research, mostly since there are evermore resources available through public access programs like the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support, previously known as the XSEDE. One reason behind the usage of mostly local resources, albeit with multiple GPUs, is the usage of Convolutional Neural Network (CNN) solutions [6, 13, 14, 15, 16, 17], which is known to have high communication demands, which can be drastically slowed down by communication overheads on distributed environments [85]. It is worth noting that one field which is gaining increasing interest regarding distributed training of CNNs is federated learning [15, 86, 87, 88]. It appeals to the usage of data locally sourced from hospital or data silos with privacy requirements. However, it is important to note that although most WSI solutions are shifting towards CNNs there is still work being done with classical image processing algorithms [16, 77]. Also, there are some limitations to the usage of CNNs. It is required for WSIs to be partitioned into rather small regions (e.g., around 512×512 pixels), being thus unable to work with larger, or even full-sized images. This can be restricting for some applications. For instance, the Camelyon16 challenge for detecting metastases in WSIs included the goal of also detecting the actual invasive cancer region, which would be challenging for CNNs since it requires a more global knowledge of the image [89].

2.5 Motivating Application

A WSI classical image processing application was chosen as the object of study and optimization for distributed execution. This is a well known watershed-based segmentation workflow, also used on previous works [7, 9, 20, 21, 22, 23, 25]. This application, also seen on Figure 2.6 focus on enabling correlative analysis, such as survival analysis and identification of significant gene expressions through pathology imaging features. Thus, the motivation of this work is to accelerate the extraction of morphological information from histological tissue structures, such as cell nuclei. This process leads to characteristics which correlate well with disease progression and clinical data [5, 7].

The original WSI image analysis applications have the following set of core analyses stages: (i) color normalization, (ii) segmentation, which detects and delineates objects of interest or cells nuclei in our use-case, (iii) feature computation to extract descriptors of the objects, and (iv) correlative analysis that classifies and/or integrates data extracted from image with other information sources according to the target analysis. Stage (i) is responsible for normalizing different images, sometimes retrieved by different equipment, enabling consistent analysis between different images. The segmentation step mostly delineates the boundaries of cell nuclei. The segmented nuclei are used to extract a large number of morphological features. These features can undergo a SA process to abbreviate the large set of features to the most important ones. From these, a correlational analysis can be performed to improve the quality of digital pathology applications.

Regarding compute costs, the normalization and correlative analysis phases are typically compute inexpensive as compared to segmentation and feature computation. The correlative analysis, for instance, works on patient signature level or a set (vector) of features per image. The segmentation, which is the most compute expensive step and target for implementation of for this work, executes complex operations on high-resolution images.

The segmentation is presented along with the feature computation on Figure 2.6. It identifies and delineates cells and nuclei using a series of transformations built on top of morphological operations. First, the background/foreground are identified (GetRGB), and an initial object candidate set is reconstructed from the cells seeds (Morphological Reconstruction) through erosion/dilation and Irregular Wavefront Propagation Pattern algorithms (IWPP). Further, holes in objects are filled (Fill Holes) with another round of dilation/erosion and objects are opened (bwOpen) to compute the cell nuclei set. Finally, objects touching each other or clumped are separated with Distance Transform and Watershed. These morphological reconstruction operations are mostly built on the concept of data propagation through the image (IWPP), which makes the computing cost dependent of data content. In other words, certain data regions, for instance, containing more

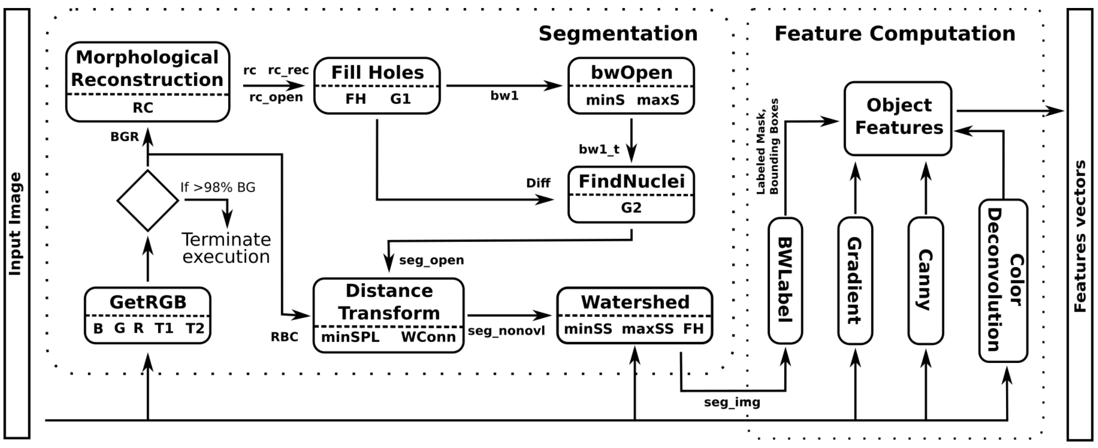


Figure 2.6: Segmentation and Feature Computation phases of the motivating application.

objects of interest tend to require more propagation, and consequently, a larger number of iterations through the image, resulting in higher execution times [90]. In contrast to regular computing pattern algorithms, as erosion/dilation, propagation algorithms cost my be rather different for same-sized inputs. Feature computation is less compute demanding than segmentation and calculates characteristics of the segmented objects, which includes color, gradient statistics, edge, and morphometry statistics. Most of the features can be computed with high parallel efficiency and are easier to implement. Note that after the GetRGB step, the execution may be terminated if the algorithm identifies an excessive amount of background (e.g., > 98%). This strategy reduces the overall execution cost by avoiding fully processing empty tiles. This approach was implemented in the original application workflow [5, 7].

Chapter 3

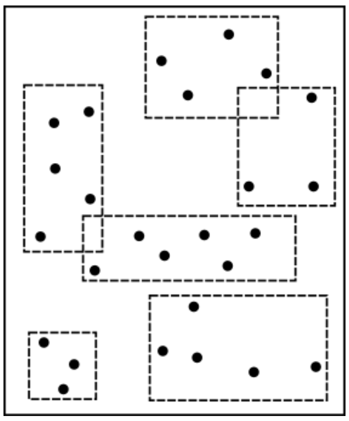
Related Works

In order for the motivating application to support distributed execution spatial data partitioning algorithms are required. These are thus defined and described on this chapter. Further, this chapter also shows a comparative analysis of related works and the current state-of-the-art related to this work.

3.1 Spatial Data Partitioning

In order to expand on data partitioning is important to define the data which is to be partitioned. This work focus on the image-based representation of data (ID), where the environment in which atomic objects (pixels) lie is partitioned [91]. Another important class of data is multidimensional point-based data (PD) [19], which has been extensively studied [39, 92, 93]. Both data formats are shown on Figure 3.1. It is important to acknowledge PD since many of its constraints are similar to ID, meaning that some partitioning algorithms were originally adapted from PD to ID.

By definition, the elements of PD are spread on a sparse space. Partitioning algorithms and data structures for this sort of data focus on balancing the number of elements by partition or generating balanced hierarchic structures for later lookup [19]. Such hierarchic organizations (e.g., trees) can prove themselves useful for improving inter-partition features, for instance, the communication cost when distributing the partitions for parallel execution [94, 18]. These objectives of partition-balancing and inter-partition feature optimization can be transferred for ID, and thus enable the application of such algorithms. The partitioning of the space into regions with the same number of points is equivalent to cost-wise partitioning of ID. As such, we only regard non-overlapping algorithms. Further, the hierarchical partitioning of an ID space can lead to a reduced sum of all perimeters, which is addressed on Chapter 5.


(a) 2-dimensional point data (PD) partitioned.

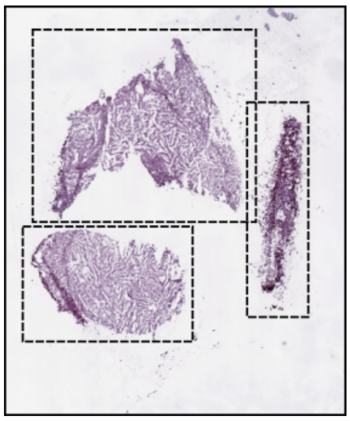

(b) Extraction of regions of interest of image-based data (ID).

Figure 3.1: Partitioning examples of multidimensional point-based data and image-based data.

ID partitions are limited to rectangular regions, which improve storage and access patterns. Further, for both ID and PD, partitions can be overlapping or non-overlapping. Overlapping partitions can lead to redundant computation for ID partitions. Both data representations share two main strategies: Fixed-Grid and Recursive Partitioning [18, 39, 95, 96, 97].

3.1.1 Fixed-Grid Spatial Partitioning

Also seen as regular-mesh [18], Fixed-Grid is a class of regular partitioning algorithms on which all partitions have the same geometrical size. For any given input, Fixed-Grid may return the exact number of partitions required, being possible to vary the grid structure. The two possible ways to generate this partitions are shown in Figure 3.2, on which the partitioning can be performed on all coordinates/dimensions or just one. Partitioning on all coordinates also have another level of complexity related to the grid organization. For instance, 18 partitions on a 2D domain can be achieved by 1×18, 2×9 and 3×6 grids (on both coordinates). Thus, for large numbers of required partitions different algorithms to find the best grid organization can be implemented. With Fixed-Grid algorithms there is no guarantee that the partitions would be balanced cost-wise.

3.1.2 Recursive Spatial Partitioning

Most popular spatial partitioning techniques are variants of recursive bisection. These algorithms performs successive cuts until a desired number of partitions is reached [18]. The partitions are balanced by a chosen criteria, returning equivalent partitions. Two of the most common algorithms are the Quad-Tree and KD-Tree [19]. These algorithms

(a) Grid-based mesh. (b) One-dimensional regular mesh.

Figure 3.2: Fixed-grid partitioning examples.

can be implemented with a hierarchical component, to optimize data search. For the purpose of this work hierarchical structures are not required. Exemplified on Figure 3.3a, Quad-Trees partitions the image into four equal regions. This algorithm can be adapted from PD into ID representation [98]. The KD-Tree algorithm is an example of recursive coordinate bisection (RCB) [18], or also defined as multidimensional binary partitions (MBP) [94]. RCB algorithms bisect a given image or region into two regions with a cut perpendicular to a coordinate axis. This process is done until the expected number of partitions is reached. KD-Trees work better when the expected number of partitions is a power of two. Otherwise, the partitions can be unbalanced cost-wise due to the 2-fold partitioning. An example of how an image can be partitioned with KD-Tree is shown in Figure 3.3b. KD-Tree usually bisects the largest coordinate, thus reducing the total borders. This reduces communication overheads for applications which requires data synchronization between partitions.

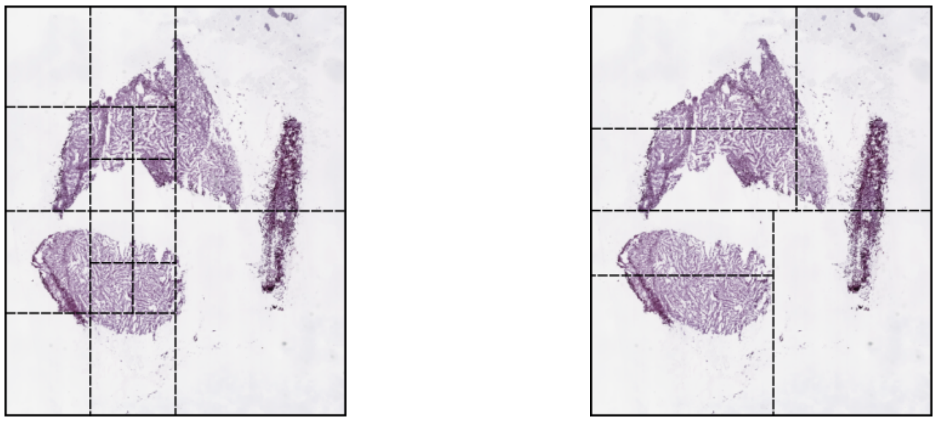

(a) Quad-Tree partitioning. (b) KD-Tree partitioning.

Figure 3.3: Recursive spatial data partitioning examples.

3.2 Related Languages, Frameworks and Tools

This sections presents related work on domain-specific languages (DSL) and tools targeting image analysis applications, and runtime/compiler systems for hybrid machines. The closest related work may be Halide itself, which was described in details in Section 2.3.2, and is extended in this work to support cooperative CPU-GPU execution and distributed memory machines along with cost-aware data partitioning.

The works presented in [70, 71] have extended Halide, respectively, with support for distributed memory execution and a hybrid (CPU-GPU) single node execution. Hybrid Halide [71] enables the use of CPU and GPU cooperatively in the execution by splitting input data for parallel processing. It minimizes host-device communication by copying the GPU output data directly to the final host output memory location. The same mechanism is implemented in our work, but we also perform efficient cost-aware data partition, even on a single node level. The Distributed Halide [70] is an interesting work that introduced a new dimension to the scheduling domain related to distributed memory execution. It allows the developer to choose a data dimension to be distributed along with the introduction of ghost-zones or inter-node communication to resolve border discontinuities. In this case, only CPUs are used, and data distribution is left entirely to the programmer, increasing the development effort. Tiramisu [60] is another interesting extension of Halide that proposes a polyhedral syntax with directives to support distributed execution and explicit data movement management. However, Tiramisu does not support Just In Time (JIT) compilation and parametric tiling of images, i.e., the input size must be known at compilation time. As such, a compilation for each data input size is required and must be known Ahead Of Time (AOT). These aspects impose an important limitation for the use of Tiramisu in our solution.

PolyMage [99] is a python embedded DSL whose applications are compiled to C++ to enable efficient execution. It abstracts application development as pipelines with a composition of image processing operations (e.g., point-wise, stencils, histograms, upsampling, downsampling). While its pipelines can only be represented as Directed Acyclic Graphs (DAGs), it has a time-iterated operation, which can hide the cyclic portions of pipelines. Nevertheless, it does not abstract input size inference, which must be defined by the user, using polyhedral notation. The schedule for the pipeline is found by PolyMage automatically, not supporting any changes on evaluation or traversal order by the user. While this can be practical for inexperienced end-users, it can hamper performance on more peculiar cases when the domain is well-known. In order to reduce the schedule optimization time, the tiles' sizes, which are the only variables of a given pipeline, have predefined valid values that reduce the search space. One advantage of generating the whole scheduling is that it enables sophisticated inter-stage tiling (e.g., parallelogram [100] tiling, split tiling

[101], overlapping [102], etc), thus optimizing data locality. However, PolyMage does not support distributed memory execution, being usable only on local CPU settings.

Several general purpose compilers and scheduling languages targeting GPU based systems have also been designed to facilitate the use of such devices, for instance, in stencil applications or linear algebra [103, 104, 105]. While these solutions fail to offer a high-level language for image analysis, such as proposed in Halide, they introduce interesting optimization aspects targeting GPU equipped machines. Fireiron [103] decouples implementation from scheduling, enabling code transformation to efficiently exploit data locality. Scheduling is performed through specifications or *specs*. These *specs* can be considered a data-structure which describes the computation patterns of a kernel (implementation code), such as memory layout, data movement and access, and the use of special GPU instructions. *Specs* can be decomposed hierarchically, providing more intermediary layers of abstraction for data management. This organization enables fine-grain control over GPU implementations at the expense of usability for end-users.

HSTREAM [104] is a compiler focused on running applications on hybrid machines. This is achieved by leveraging pragma scheduling statements to annotate the application code and a runtime system. Different from newer versions of OpenMP, which can offload computation to single accelerator devices (e.g., GPU), HSTREAM provides mechanisms to perform cooperative execution among multiple heterogeneous devices. This is possible through a source-to-source compiler for target specific code, which is then executed on HSTREAM's runtime system. On the back-end, HTREAM uses OpenMP for CPU scheduling and CUDA for GPUs. Work distribution is performed directly by the code programmer through the definition of data chunks for each device. This simple solution allows other developers to implement their desired workload partition policies. However, HSTREAM's pragma statements are interpreted at compile time, which limits the scheduler flexibility.

Panda is another remarkable compiler based solution [105]. It provides a set of pragma directives to parallelize and distributed stencil tasks to distributed CPU or GPU only settings, or distributed hybrid settings. Execution with multiple GPUs per node is supported. As with HSTREAM, Panda uses a source-to-source compiler, which translates its pragma annotations to MPI, CUDA, OpenMP or a combination of them. The MPI/CUDA/OpenMP code is fully compiled to binary, meaning that no runtime system is required or made available by Panda. While Panda enables efficient stencil execution, it does not offer a higher-level language and scheduling concepts, as with Halide. Further, Panda also does not offer the flexibility of dynamic scheduling as in our solution. In other words, Panda performance on distributed system fully rely on automatic compiler optimizations, and it does not deal with irregular data computations. It also does not offer tools specific

Table 3.1: Related work comparative analysis. Each work is classified considering code availability, whether it is a fully fledged DSL or a set of compilation definitions [3], scheduling decoupling, target processor, distributed execution, and partitioning characteristics.

Work	Code Avail.	Schedule Decoupling	DSL Img. Analysis	Target Device	Dist. Exec.	Cost-Aware Partitioning
Halide [56]	✓	✓	✓	CPU or GPU	✘	✘
Tiramisu [60]	✓	✓	✓	CPU or GPU	✓	✘
Hybrid Halide [71]	✘	✓	✓	Hybrid, CPU,GPU	✘	✘
Distributed Halide [70]	✘	✓	✓	CPU	✓	✘
PolyMage [99]	✓	✘	✓	CPU	✘	✘
Fireiron [103]	✘	✓	✘	GPU	✓	✘
HSTREAM [104]	✘	✘	✘	Hybrid, CPU,GPU	✘	✘
Panda [105]	✘	✘	✘	Hybrid CPU, GPU	✓	✘
Our	✓	✓	✓	Hybrid, CPU, GPU	✓	✓

to image analysis, which is the target of this work.

The related work is summarized in Table 3.1 to facilitate comparison. They are characterized by: (i) public code availability, (ii) whether the schedule can be decoupled from the implementation, (iii) how the system is implemented, (iv) the target computing devices supported, (v) if there is support for distributed memory execution, and (vi) if they support irregular data partition for parallel execution. As may be seen, solution proposed on this work is the only one to enable hybrid and distributed execution with irregular data partition targeting image analysis. As presented in the experimental results, these features are important to fully take advantage of modern computing systems and, consequently, to maximize the performance of our target application domain.

Chapter 4

Supporting Pathology Image Analysis Applications on Distributed Memory Hybrid Systems

This chapter describes the system implementation and other methods for supporting efficient execution of histopathology analysis on a dMIMD environment with both multiple computing nodes on a network and local heterogeneous computing devices. The execution of real-world applications on these modern environments is a complex task that may require intervention from the programmer. For instance, (i) workload must be distributed among compute nodes and among heterogeneous processing elements on each node, (ii) communication must be orchestrated, and (iii) there should be efficient code targeting the available devices. In this work, it is built a system solution that enables efficient use of hybrid systems and addresses these challenges, while being simple to use.

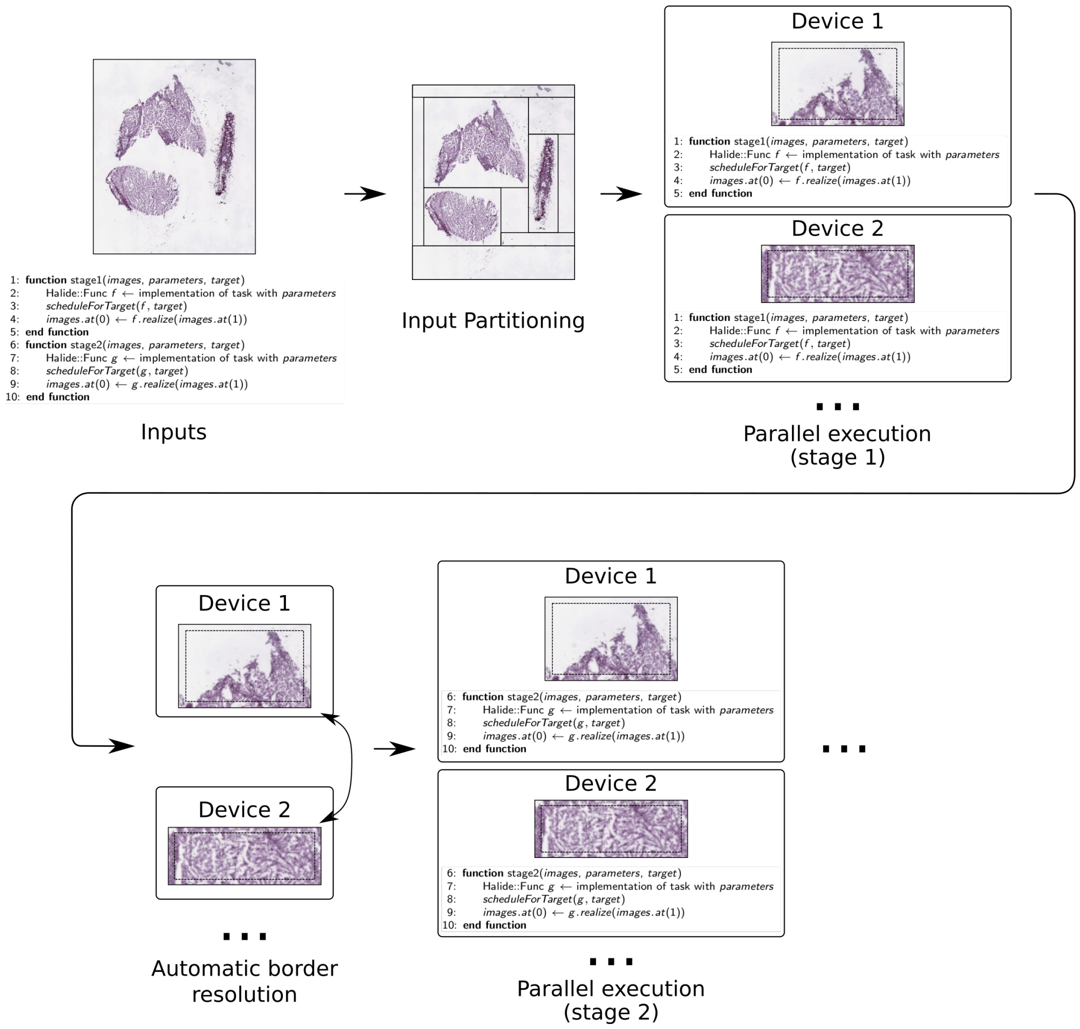
As previously stated, there is a demand for automatic WSI analysis applications. These can be computationally demanding, needing large amounts of time to be completed. HPC solutions are great candidates for improving this issue of high computing costs. However, HPC techniques/resources can be complex to use efficiently [106]. In order to use distributed resources, the application input must be partitioned, which is a complex problem since it adds the issue of workload imbalance, meriting a whole field of research [107, 108]. Dynamic scheduling can alleviate such issue at the cost of more complex execution systems. Static partitioning can result in better system performance regarding overheads, while also rendering the workload imbalance, and thus overall performance, dependent on the partitioning quality. Further, it has been shown that WSI applications are becoming evermore reliant on GPU devices [6, 14, 106, 108]. The use of such resources adds even more complexity to these solutions or systems. GPUs not only require specific programming languages and/or frameworks but also are more difficult to efficiently program. Also, multi-GPU compute resources are also becoming evermore available, for which systems and solutions for distributed memory environments are required for coordinating the execution on all GPU devices [108]. Finally, medical domain experts are not expected to be familiar enough with all the nuances of using these HPC solutions, resulting in a difficulty to efficiently use the available HPC resources.

The system proposed on this work aims to alleviate the described issues, allowing easier and more efficient usage of HPC resources for WSI applications. The user will be able to use a high-level DSL for efficiently programming their applications for both CPU and GPUs. The input WSIs need to be partitioned with regard to workload imbalance. The system also needs to stage computational tasks across distributed hybrid resources, resolving data locality issues. A system which resolves all of these issues is presented on the following sections.

4.1 System Overview

The proposed system employs a simple execution model, on which a user can execute a WSI processing workflow, depicted by Figure 4.1. This workflow can be defined as a directed acyclic graph of compute stages. Stages are modeled through the high-level DSL Halide [56]. Each stage will be executable on any of the available heterogeneous resources due to a Halide integration to the system, detailed on the next section. With an input workflow and an input WSI, or set thereof, the system is able to run a single stage on each available resource. These resources can be a set of CPU cores (e.g., a socket or full node) or GPUs and other accelerators. Before the execution of any stage the system partitions the input WSIs statically with a cost-aware partitioning algorithm with focus on load balancing, which is detailed on Section 5. It is common for WSI applications to partition the input images, which require them to manage what should be done regarding the borders of the partitions [109]. A border resolution semantic allows the independent execution of image partitions. The proposed system resolves the partitions borders by employing a redundant overlapping region between every two partitions, which allows their independent execution. At the end of execution of a partition, data regarding the overlapping regions can be exchanged between partitions which are used for another compute stage. From it the execution process iterates on the remaining stages.

The proposed system, as compatible with RT, employs a worker-dispatcher model. The manager is responsible for partitioning the input WSIs and managing the execution of the stages. Each Worker requests a stage for execution for each of its compute resource. For instance, a hybrid node with dual-socket CPUs and 2 GPUs can request 3 stages, one for all the CPU cores and 1 for each GPU. Before the execution of the actual user-


Inputs
Input Partitioning
Device 1
Device 2
Parallel execution (stage 1)
Device 1
Device 2
Automatic border resolution
Device 1
Device 2
Parallel execution (stage 2)

```
function stage1(images, parameters, target)
    Halide::Func f ← implementation of task with parameters
    scheduleForTarget(f, target)
    images.at(0) ← f.realize(images.at(1))
end function
function stage2(images, parameters, target)
    Halide::Func g ← implementation of task with parameters
    scheduleForTarget(g, target)
    images.at(0) ← g.realize(images.at(1))
end function
```

```
function stage1(images, parameters, target)
    Halide::Func f ← implementation of task with parameters
    scheduleForTarget(f, target)
    images.at(0) ← f.realize(images.at(1))
end function
```

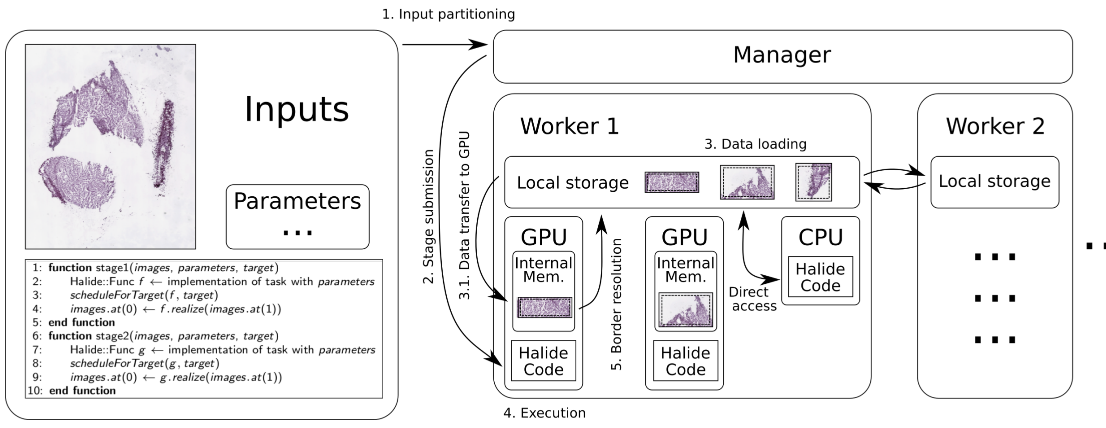
```
function stage1(images, parameters, target)
    Halide::Func f ← implementation of task with parameters
    scheduleForTarget(f, target)
    images.at(0) ← f.realize(images.at(1))
end function
```

```
function stage2(images, parameters, target)
    Halide::Func g ← implementation of task with parameters
    scheduleForTarget(g, target)
    images.at(0) ← g.realize(images.at(1))
end function
```

```
function stage2(images, parameters, target)
    Halide::Func g ← implementation of task with parameters
    scheduleForTarget(g, target)
    images.at(0) ← g.realize(images.at(1))
end function
```

Figure 4.1: Execution model of the proposed system. A set of WSIs and a application workflow are used as inputs. The system then partitions the input WSIs and executes one stage of the workflow at a time. Partitions are executed in parallel. Borders are automatically resolved at the end of the execution of a given stage, before the next stage can be executed.

implemented application code the data dependencies are resolved at the distributed inter-node level by RT and intra-node by the proposed system. Initially, every Worker possesses the input WSIs at local storage to optimize its execution (WSI files are heavily compressed and can be transferred to all nodes beforehand). If an input partition for a given stage belongs to the initial WSI then a Worker is able to load it directly from its local node storage. Otherwise, RT's data management system ensures that the required data is present, transferred from other distributed nodes. When executing on GPUs the data is also transferred automatically to it before the user application code is executed. After the execution and border resolution, the resource which executed the stage is freed, allowing the Worker to request a new stage from the manager. This whole process is shown in Figure 4.2.



```
function stage1($images$, $parameters$, $target$)
    Halide::Func $f \leftarrow$ implementation of task with $parameters$
    $scheduleForTarget(f, target)$
    $images.at(0) \leftarrow f.realize(images.at(1))$
end function
function stage2($images$, $parameters$, $target$)
    Halide::Func $g \leftarrow$ implementation of task with $parameters$
    $scheduleForTarget(g, target)$
    $images.at(0) \leftarrow g.realize(images.at(1))$
end function
```

Figure 4.2: Process of executing a stage. After partitioning and task submission any required data partitions are loaded for the required resource. After execution and border resolution a new stage can be requested.

4.2 Interface for Implementing the Applications

One way to improve the entry barrier for non-experts of HPC employed by the proposed system was the usage of a high-level DSL which enables an unified programming model for both CPU and GPU resources. For such task the Halide DSL [56] was chosen with the goal of providing a high-level language for Region Templates (RT) application developers while enabling distributed memory and hybrid CPU-GPU execution of applications. The combined use of Halide and RT was also motivated by the goal of maintaining application decomposition into a hierarchical workflow. This model is natively supported in RT, with Halide serving as a language for implementing stages' internal code or processing transformations.

In theory, the proposed system could use any sort of DSL as a basis for programing the user-defined application, given that its semantics allow programing for both CPU and GPU without distinction. However, given the motivating domain of WSI applications, Halide is a great fit as an image processing focused tool. As stated, Halide provides an unified semantic for programming for both CPU and GPU devices, which is not true for other lower-level tools, such as CUDA or OpenMP. In order to improve its usability, it is desirable for the chosen DSL or framework to be embedded on C++, excluding more specific solutions, like the Chapel parallel programing language [110]. Also regarding better support, Halide is a consolidated tool, under regular development since 2013 with its original paper currently having over 1200 citations on Google Scholar [56] and relevant research work being done with it [73, 111, 112, 113]. Regarding performance, Halide proves to be efficient to the degree of being used to elevate the performance of legate code [111] or even improve on the performance of operations from the widely used image editing tool Photoshop [73]. Other tools, e.g, the HIPA^{cc} DSL [114, 115], have been proposed as valid alternatives to Halide performance-wide. However, Halide's decision of having scheduling code separated from the remaining of the code allows more flexibility of usage by non-experts on HPC, while still attaining reasonable performance. Finally, there is extensive work on automatic scheduling of Halide workflows, further reducing the difficulty of reaching performant code without extensive knowledge on HPC [57, 58, 59].

Algorithm 4 Example of an application on top of RT with Halide.

```
1: function STAGE1($images$, $parameters$, $target$)
2:     Halide::Func $f \leftarrow$ implementation of task with $parameters$
3:     $scheduleForTarget(f, target)$
4:     $images.at(0) \leftarrow f.realize(images.at(1))$
5: end function
6: function STAGE2($images$, $parameters$, $target$)
7:     Halide::Func $g \leftarrow$ implementation of task with $parameters$
8:     $scheduleForTarget(g, target)$
9:     $images.at(0) \leftarrow g.realize(images.at(1))$
10: end function
11: function MANAGER( )
12:     $params \leftarrow$ list of all application parameters
13:     $partitioner \leftarrow$ RT::CADP($dataPath$, $gzSize$, $n$, $costFunction$, $gpuSpeedup$)
14:     for each $partition$ in $partitioner.getParts()$ do
15:         RT::EXECUTE($stage1$, $partition$, $params$)
16:         RT::EXECUTE($stage2$, $partition$, $params$)
17:         $stage2.depends(stage1)$
18:     end for
19:     RT::STARTUPEXECUTION( )
20:     RT::FINALIZESYSTEM( )
21: end function
```

The proposed solution concentrates on the development and optimization of the domain-specific data transformations (stages). A pseudo-code example for a typical user-defined application with two stages in the proposed system is presented in Algorithm 4. The user is required to implement the stage function that will execute on the Workers, following a predefined template. This template includes used data partitions (input and output *images* with the Halide data types), parameters, and the target device (lines 1 and 6). The output data of each stage is locally assigned (lines 4 and 9) and globally managed by RT. Thus, any data movement required after the completion of a stage is automatically handled. With this model the code within each stage is a set of Halide functions instead of pure C/C++ (or CUDA) as with original RT applications. This eases the burden of application development due to the simpler Halide syntax. Besides decoupling implementation and code optimization, this strategy also reduces the effort on code generation or implementation targeting multiple devices, since we can rely on Halide to generate efficient code. We can thus generate, for instance, CPU and GPU code in order to enable cooperative execution on hybrid compute nodes. Although Halide by itself could execute the same code (lines 1 to 10), it would only be possible to use a single processing element (either CPU or 1 GPU) in a shared memory machine, also requiring data management code before the pipeline implementation.

The Manager side of the application is presented on line 11 of Algorithm 4. The code developed in this case partitions the input data (large input tissue image) for parallel and distributed execution, and instantiates the application workflow by creating stages and setting respective dependencies (line 17). The data partitioning strategy used to divide the application data domain is chosen in line 13. The execution strategy is designed so that communication among tiles (or among processors) is allowed by the exchange of tile borders or ghost zones [72, 109] at the beginning of a stage instance execution. The ghost zones or ghost borders are areas around tiles with defined width that are included in the original tiles for processing. Once border information is added to a tile, the stage instance may process tiles independently. Because the data is written/read to/from the RT storage system at the end/beginning of a stage instance execution, information among tiles (borders) are automatically exchanged without additional explicit communication

As the default partitioner, the Cost-Aware Data Partitioning algorithm (CADP), presented in Chapter 5, has been selected. This strategy receives as input the data to be processed, size of ghost zone for tiles if required ($gzSize$), number of partitions to be generated ($nTiles$), a cost-function used to estimate computing demand of data domain regions ($costFunction$), and expected GPU vs. CPU speedup ($gpuSpeedup$) for cases in which CPUs are used cooperatively with accelerators. The remaining of the Manager code (lines 14 to 20) describes the workflow generation, which dispatches a workflow with a single stage for each input tile, and the actual startup and end of the execution. However, multiple stages and arbitrary application workflows with dependencies are supported in our system.

4.3 Internal Data Management for Hybrid Execution

In order to execute Halide tasks while also supporting hybrid cooperative execution, Region Templates (RT) had its internal tasks representation changed. Mainly, a new Stage class, the AutoStage, was implemented. AutoStage supports (i) the execution of arbitrary code, defined outside RT by the user, (ii) conversion of images to and from Halide's format, and (iii) the early termination of workflows. Since the selection of the target architecture for execution (e.g., CPU/GPU) is passed to the user stage implementation (see Algorithm 4, lines 1 and 6), AutoStage instances are tagged with the architecture it should execute. This tag is defined by the data partitioning algorithm (line 13) and passed through to the user code. It is important to notice that in the manager code (lines 11-21), the user is not required to directly manage images or files. Only Halide's data structures are used on its application's implementation (lines 1 to 10).

After the partitioning of the input images, these must be added to RT's internal data repository. This allows easy and consistent data access on a distributed memory setting. Each input image and intermediary image buffer must have a Region Templates Object (RTO) encapsulating it. A RTO represents a globally accessible image, which can be spatially decomposed and have multiple versions . This also allows in-place execution with a backup of previous versions. RTOs are composed by Data Regions (DR), each representing a partition of concrete data, available locally on memory of a compute node. Since the I/O process of generating the RTOs and its DRs is performed by RT's Manager process, the DRs were updated to allow a lazy creation process. DRs' actual data from .svs files are then only read by Worker processes on demand. This optimized the I/O process of DRs by decentralizing it.

By sending a stage for execution with a data partition (Algorithm 4, lines 15 and 16), an AutoStage instance is created. This processes the input parameters of the application and finds the partition's global RTO reference, with its DRs. Both inputs and outputs are assigned. Each submitted stage is added to the proper global execution queue on the Manager for its tagged execution target. As with the original version of RT, each Worker has an execution thread per computing resource. For instance, for a compute node with two GPUs on a dual-CPU-socket motherboard, three threads are created, one for each GPU device and one for one for all CPU cores.

When a local Worker's thread receives a task (either for CPU or GPU) the process of Algorithm 5 is performed. The first step is the retrieval of DRs (lines 2-6). As mentioned before, DRs are in-memory data objects, which in this work are lazily read from the .svs files. The reading process is performed at this point. If a DR is not a .svs input file then its data is present on RT, which handles the data movement. It is worth noting that the RT scheduler is data-aware, meaning that it is likely that the required data is already present on the executing node. The output DR is also created/allocated at this point. Next, the input DRs are checked for the termination of the workflow (lines 7-12). The motivating domain of this work benefits from early termination of inconsequential partitions (e.g., background only). If a realized task returns a terminated flag, then all subsequent tasks should be terminated (line 17). This signal is propagated by the input/output DRs (lines 7-12 and 18-21).

The input/output data for the executing task are pre-formatted for Halide applications (lines 13-15). The conversion process between RT and Halide objects does not perform any copy, using the internal OpenCV data references, common to both formats. This

Algorithm 5 Internal execution of an AutoStage on a Worker.

```
1: function AutoStage::run( )
2:     $outputDr \leftarrow$ RT::generateDr($outputShape$, $partitionId$, $outputRTO$)
3:     $drList.append(outputDr)$
4:     for each $r$ in $inputRtoList$ do
5:         $drList.append(r.getDr(partitionId))$
6:     end for
7:     for each $dr$ in $drList$ do
8:         if $dr.terminated()$ then
9:             $outputDr.terminate()$
10:            return
11:        end if
12:    end for
13:    for each $dr$ in $drList$ do
14:        $halBufList.append$(RT::cvToHal($dr.getCv()$))
15:    end for
16:    $task \leftarrow$ RT::getTask($taskRef$)
17:    $terminated \leftarrow task.realize(halBufList, parameters, target)$
18:    if $terminated$ then
19:        $outputDr.terminate()$
20:        return
21:    end if
22:    $halOut \leftarrow halBufList.at(0)$
23:    $outputDr \leftarrow$ RT::halToDr($halOut$)
24: end function
```

process is further detailed in the next section. The reference for the task to be executed is retrieved from a RT library. The code reference is accessible by any distributed Worker MPI process through the generation of a local map of references to the user-defined Halide implementations. This map is generated on every distributed process to enable the decoupling of RT code from user code. After the retrieved task is executed completely (line 17), the output data is reformatted back to RT (lines 22 and 23). This also does not perform any local data copy or transfer, but allows for RT to send this data to other nodes. The exception is for GPU execution, on which data is transferred between device and host.

4.4 Halide Integration and Implementation Details

In order to enable the seamless use of Halide by the user, RT need to (i) perform automatic conversion of its internal data representation to/from Halide's representation, (ii) automatically manage GPU data, and (iii) manage execution with multiple GPUs. As mentioned, conversions between DRs and Halide::Buffer's (Halide's data representation) do not perform unwanted data copy or movement. This is possible since both data types use the OpenCV's *cv::Mat* data type as their underlying data container. However, for color *cv::Mat* data the memory layout of color channels can either be interleaved or planar. For planar data, the full image of each color channel is sequentially set on memory, one channel at a time. Interleaved data is set on memory on a per-pixel basis, on which each pixel with all of its channels' values are grouped together, with each pixel set sequentially on memory. This is an important information to know since different layouts can lead to incorrect execution results. This presents a problem for Halide, which uses the interleaved pattern by default since, by definition, the innermost coordinate of a color image is the color channel. Given that the data read from the .svs files can be in the planar format, this planar-interleaved conversion is performed automatically when necessary. For GPU execution, data transfer to/from device is also automatically performed by AutoStage.

As currently available, Halide does not natively supports cooperative execution of its pipelines on distributed memory environments. These environments can be either multiple GPUs or heterogeneous devices (e.g., CPU + GPU). Heterogeneous execution is handled by RT while multi-GPU execution was implemented by extending Halide itself.

GPU execution (as used on this work by Halide) is performed through CUDA. As such it requires the initialization of a reference object to a given GPU device before performing data movement or execution operations. Originally, Halide initializes a single static reference of a GPU. If more than one GPU is available, the initialized reference points to the device with the highest processing capacity (i.e., most CUDA cores). These references are initialized only once, at the first realization call of a Halide pipeline scheduled for CUDA.

Multi-GPU support was implemented by maintaining an internal array of references to each GPU device available. These are instantiated once with a new initialization function, also responsible for configuring the number of GPU devices to be available. This initialization function is invoked at regular RT initialization. One aspect to be aware is that each pipeline can only be executed on the GPU on which its data is present. Also, GPU memory is much scarcer than CPU memory, meaning that execution of multiple pipelines on the same GPU should be avoided. As such, before realizing or performing data movement operations, a GPU reference is allocated by an assignment function (*getGpuRef()*) which locks a single GPU for use, returning its ID.

Figure 4.3 shows how a GPU device can be accessed in a thread-safe manner, while also operating on the correct GPU. At the header of the realization code, a GPU ID is retrieved. The allocated GPU is tagged as unavailable until the finalization step. From an allocated GPU it is possible to execute as many Halide's realizations as desired. All realizations are performed on the same GPU. Any data dependency, which can only exist at the host side, is lazily resolved by Halide. The finalization step frees all temporary data on GPU

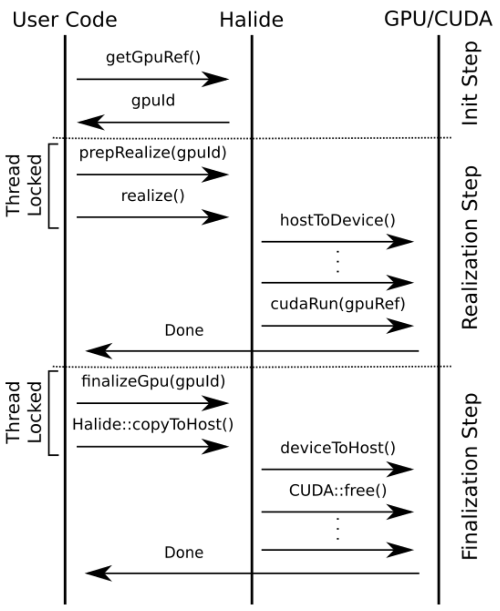


Figure 4.3: An illustration of the processes involved into using multiple GPU devices. Direct communication between CUDA and user code is just a simplification. In reality all CUDA-related calls are performed only through Halide.

(e.g., copied inputs) and enables the transfer of result data back to the host. The design of using preparation functions was devised as a way to inform Halide the correct GPU to use while not interfering with its interfaces (e.g., *realize()*, *copyToHost()*). In order to support multiple GPU executions the preparation functions are thread-safe, locking the whole Halide GPU sub-system and unlocking at either a realization or a finalization. After the beginning of the realization process, the GPU sub-system is unlocked and can attend to other GPUs. It is worth noting that the thread-locked section of this protocol is short enough to not significantly influence the overall execution time. After *finalizeGpu()* finishes all required device-to-host transfers and clear its memory, the GPU is once again available to other tasks.

Chapter 5

Cost-Aware Data Partitioning for Irregular-Cost Applications

This chapter details the problem of partitioning input images for parallel execution on hybrid distributed memory machines. The closest related work, on image analysis, leaves this task to the programmer which performs trivial regular partitioning of the domain [70]. In other application domains a significant number of approaches to automate this task were proposed [39, 116, 117, 118, 119, 120]. However, these strategies consider that processing costs are homogeneous across the data domain and employ data structures like Quad-Trees and KD-Trees to partition the data. In our motivating application domain, on the other hand, the computing cost of a region is heterogeneous and vary, for instance, according to the density and size of objects it contains. Thus, using those data partitioning strategies may lead to significant load imbalance in the parallel execution.

This observation has motivated the development of a new class of automatic partitioning algorithms, called Cost-Aware Data Partitioning (CADP), which takes into consideration the heterogeneity or irregularity of the domain's processing cost to minimize the load imbalance. CADP algorithms consider this irregularity in the partitioning and uses the expected computation cost of partitions created to reduce imbalance among them on a distributed execution environment. Since the computational cost of a region may be dependent of the application processing patterns and data content, the cost estimation used by CADP is provided by a cost-function that can be customized according to the application being executed. The function receives a Region Template (RT) data region as input and is expected to return a value, which is used in CADP to compare the relative cost of different regions. In our motivating application's domain of segmentation for cells nuclei of tissue images, some examples of such metrics could be the number of objects or their areas in a region, or the density of the partition compared to the background area.

5.1 The Expected Cost Bisection (ECB) CADP Algorithm

The first CADP algorithm was designed based on the successive bisection of a larger area, extracting a partition with an expected cost, the ECB. Each partition is set to have a cost proportional to the number of expected partitions. For instance, for 8 partitions each of these should have 1/8 of the processing cost for the whole image. The main phases of ECB proposed here are presented in Algorithm 6. It is composed of three main components: (i) a background/foreground separation that is employed to perform an early detection of background/foreground areas; (ii) a 2-cut cost-wise based partitioning algorithm that performs the data domain division for parallel processing on both homogeneous and heterogeneous environments, and (iii) a cost-function used to guide the data partitioning (see Section 5.3). The first step of Algorithm 6 (line 2) is in charge of detecting areas of the image that are mostly foreground (dense), and separate them for the background areas (sparse). This separation is performed to avoid that very large partitions with a small computing cost are created, because this may increase I/O costs of such partitions, consequently making the balanced partition process harder. This step separates the whole input image in two sets of regions/sub-images: $denseTiles$ and $sparseTiles$. Since the sparse partitions do not contain semantically relevant information for the workflow, only the dense partitions are sent for execution. The $denseTiles$, which are at least one, are then submitted to the cost-aware partitioning (line 3) to return the exact $nTiles$ for parallel execution. It is possible that the initial number of dense tiles is greater than the number of expected tiles, $nTiles$. Currently, the dense partitioner returns the same dense tiles for this case, doing nothing. However, this case is rare for the application domain, on which 1-4 initial dense regions are usually found, with $nTiles$ being in the interval of 8-32. Finally, all tiles are combined and then returned (line 4)

Algorithm 6 Cost-Aware Data Partitioning (CADP) Algorithm.

```
1: function CADP($image, foregroundFunc, costFunc, nTiles, gpuAcc$)
2:     ($denseTiles, sparseTiles$) ← RT::BFSeparation($image, foregroundFunc$)
3:     $denseTiles$ ← RT::denseHybridPartition($denseTiles, costFunc, nTiles, gpuAcc$)
4:     return $denseTiles$
5: end function
```

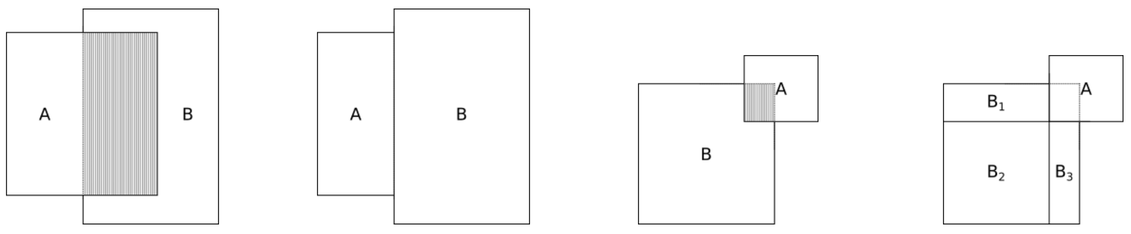
5.1.1 Background Separation/Partitioning

The $background/foreground$ separation ($BFSeparation$) is performed in the following steps (i) find the dense regions minimum bounding boxes (BB), (ii) remove overlapping

Table 5.1: All possible cases of overlapping between two bounding boxes A and B with how they are resolved. By definition B is larger in area than A.

	Case description	Condition	Resolution
1	A inside B	$A.xi \geq B.xi$ **and** $A.xo \leq B.xo$ **and** $A.yi \geq B.yi$ **and** $A.yo \leq B.yo$	Remove region A
2	A left of B vertically inside	$A.xi < B.xi$ **and** $B.xi < A.xo \leq B.xo$ **and** $A.yi \geq B.yi$ **and** $A.yo \leq B.yo$	$A.xo = B.xi - 1$
3	A right of B vertically inside	$B.xi < A.xi \leq B.xo$ **and** $A.xo > B.xo$ **and** $A.yi \geq B.yi$ **and** $A.yo \leq B.yo$	$A.xi = B.xo + 1$
4	A above of B horizontally inside	$A.xi \geq B.xi$ **and** $A.xo \leq B.xo$ **and** $A.yi < B.yi$ **and** $B.yi < A.yo \leq B.yo$	$A.yo = B.yi - 1$
5	A below of B horizontally inside	$A.xi \geq B.xi$ **and** $A.xo \leq B.xo$ **and** $B.yi < A.yi \leq B.yo$ **and** $A.yo > B.yo$	$A.yi = B.yo + 1$
6	A above left of B A smaller then B	$A.xi < B.xi$ **and** $B.xi < A.xo \leq B.xo$ **and** $A.yi < B.yi$ **and** $B.yi < A.yo \leq B.yo$	Break region B on $x = A.xo$, $y = A.yo$
7	A above right of B A smaller then B	$B.xi \leq A.xi < B.xo$ **and** $A.xo > B.xo$ **and** $A.yi < B.yi$ **and** $B.yi < A.yo \leq B.yo$	Break region B on $x = A.xi$, $y = A.yo$
8	A below left of B A smaller then B	$A.xi < B.xi$ **and** $B.xi < A.xo \leq B.xo$ **and** $B.yi < A.yi \leq B.yo$ **and** $A.yo > B.yo$	Break region B on $x = A.xo$, $y = A.yi$
9	A below right of B A smaller then B	$B.xi \leq A.xi < B.xo$ **and** $A.xo > B.xo$ **and** $B.yi < A.yi \leq B.yo$ **and** $A.yo > B.yo$	Break region B on $x = A.xi$, $y = A.yi$

between dense BBs, and (iii) generate background tiles for the remaining image areas. Background areas may be found in tissue images with a threshold cost-function, but it may require more sophisticated approaches in other domains. Thus, the function is customizable by the user depending on the target application ($foregroundFunc$). This function must also be inexpensive, otherwise some of the partitioning benefits would be offset by its cost. This is further discussed in Section 5.3, but we want to anticipate that we use lower resolution versions of the data to perform such computation, thus reducing partitioning costs. With the aid of the $foregroundFunc$, Bounding Boxes (BB) are computed for each dense region, and overlapping BB are resolved to create non-overlapping dense tiles. Another advantage of this strategy is that, for the motivating application, identified background only areas may be aborted earlier at run-time without impacting the final output quality, thus reducing the overall execution cost for the image. Although the execution time for these sparse tiles is expected to be negligible, the I/O



(a) Horizontal case. (b) Horizontal sol. (c) Diagonal case. (d) Diagonal solution.

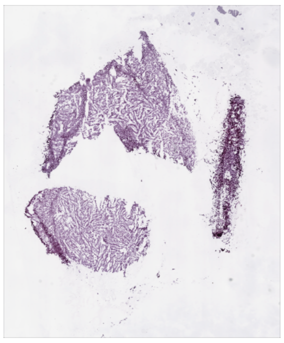
Figure 5.1: Overlapping BB resolution. For the horizontal/vertical cases, A is reduced. For the Diagonal cases, B is broken into 4 tiles, being the fully overlapped tile removed.

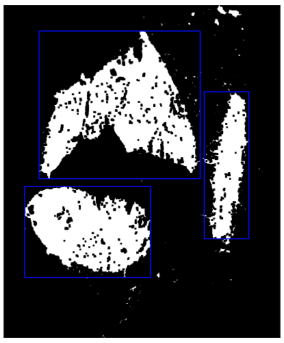
reading times can be expressive since the number of generated sparse tiles, as their sizes, are irregular. This can impact negatively the overall load imbalance of the application. To attenuate this, sparse tiles are partitioned into the smallest multiple of the expected tiles, greater than the initial number of sparse tiles. For instance, for an expected 16 tiles with (i) 8 and (ii) 22 sparse tiles, the partitioning would result in a total of (i) 16 and (ii) 32 sparse tiles. The partitioning algorithm for sparse partitions is the same as the dense partitioner (see Algorithm 9), using the area of the partition as its cost-function.

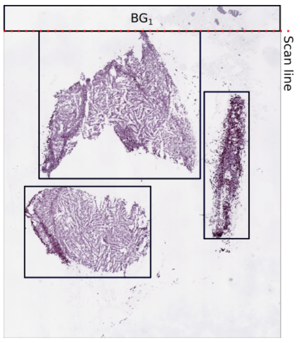
Algorithm 7 Background partitions generator.

```
1: function GenerateBgPartitions($dense$)
2:     $allPart \leftarrow dense$
3:     $open \leftarrow []$
4:     $prevY \leftarrow 0$
5:     SortByYi($dense$)
6:     while $open$ not empty or $dense$ not empty do
7:         if $dense$ not empty then
8:             $cur \leftarrow dense.head()$
9:         end if
10:        $headY \leftarrow open.head\_y()$
11:        if $open$ is empty or ($dense$ not empty and $cur.yi \leq headY.yo$) then
12:            $allPart.insert($MakeBlocks$(open, prevY, cur.yi, w-1))$
13:            $prevY \leftarrow cur.yi$
14:            $open.insert(cur)$
15:            $dense.erase(cur)$
16:        else if $dense$ is empty or ($dense$ not empty and $cur.yi >$ headY.yo) then
17:            $allPart.insert($MakeBlocks$(open, prevY, headY.yo, w-1))$
18:            $prevY \leftarrow headY.yo$
19:            $open.erase(headY)$
20:        end if
21:    end while
22:    $lastPart(xi, yi, xo, yo) \leftarrow (0, prevY, w-1, h-1)$
23:    $allPart.insert(lastPart)$
24:    return $allPart$
25: end function
```

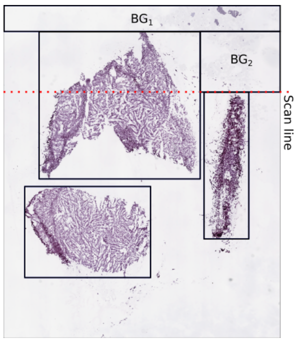
The first step for *BFSeparation*, after generating the initial dense BB is the removal of overlapping regions. Essentially, there are two main types of overlapping, vertical/horizontal and diagonal. Assuming two partitions A and B, being A smaller than B, for the first case, aiming to maintain the same number of initial dense partitions, the smaller partition is reduced in order to not overlap anymore. Thus, the overlap between the two partitions is only executed by partition B (see Figures 5.1a and 5.1b). For the diagonal case it is impossible to remove the overlapping while also returning rectangular BBs which covers all the initial area. For these cases the larger partition (B) is divided in 4


(a) Initial test image from a small section of a whole slide tissue image.

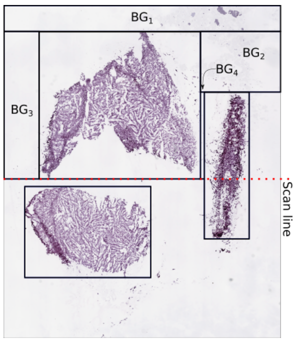

(b) Background binary mask with three dense regions.



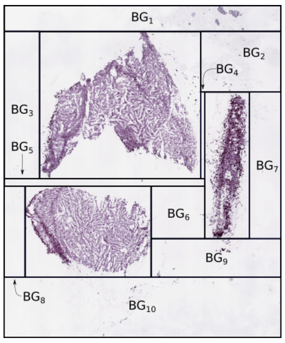
(c) Scan line hits its first dense region, generating all partitions between the beginning of the image and the scan line.



(d) On second dense region only a single partition is generated (BG_2).



(e) Scan line finds the end of a dense region. BG_3 and BG_4 can now be generated.



(f) Final result after scan line reached the bottom of the image.

Figure 5.2: Background partition generation from initial dense partitions using the scan line algorithm. Partitions are numbered according to the order they were generated.

new parts, one of which is fully inside A. For this trivial overlapping case, the smaller and internal partition is destroyed (see Figures 5.1c and 5.1d). Although Figure 5.1 shows only one example of each type of overlapping, Table 5.1 enumerates all remaining analogous overlapping cases, assuming bounding boxes A and B with coordinates A.xi (initial coordinate x), A.yo (final coordinate y), and that A is smaller than B, area-wise.

With the non-overlapping dense regions, the background partitions are generated through a vertical scanning of the image, from top to bottom (see Figure 5.2). As described in Algorithm 7, the dense partitions (as bounding boxes) are copied into the output list (line 2) since the *dense* list itself will be consumed by the algorithm. The dense partitions are then sorted by their upper vertical component (yi) in a non-descending order (line 5), simulating the vertical scan. In addition to the *dense* list, the *open* list is used as a register of which dense regions are currently in the supposed scan line (see Figure 5.2d). This list is populated by transferring a dense region from *dense* to *open* whenever the scan line reaches an upper-bound of it (line 14). The actual background regions are generated on two occasions, (i) when the scan line hits the top of a dense partition (lines 11-15) or (ii) when the scan line hits the end of a dense partition on the *open* list (lines 16-20). The background partitions generated are between $[prevY, cur.yi]$ for (i) (line 12) and between $[prevY, headY.yo]$ for (ii) (line 17). The generation occurs also emulating a scan line, horizontal this time. The regions are generated whenever there is a void between the borders of the image or the borders of dense *open* regions. These partitions are inserted on the output *allPart* list (lines 12 and 17). When all *dense* and *open* partitions have been consumed, the last horizontal partition is created (lines 22-23), from *prevY* to the bottom border of the image, spanning the whole width of the image. Figure 5.2 presents an example, on which three initial dense partitions generate 10 background partitions.

5.1.2 Data Partitioning for Homogeneous Environments

The data partitioning phase will receive as input a list of m dense regions to be partitioned into n (number of nodes) regions with minimum load imbalance, described in Algorithm 8. In this process all partitions are kept on a list sorted by their estimated cost, beginning with the initial m partitions (*initialPart* on line 3). Although able to be parameterized, the *expectedCost* value of each partition is defined as the ratio of the initial full image cost by the number of expected partitions (line 2). It then selects the partition with the higher cost and breaks that partition into two others with (almost) same estimated cost (line 6). The bisection, which can be either vertical or horizontal on a 2-dimensional domain, is sought with a binary-search algorithm in the partition domain.

Algorithm 8 ECB Homogeneous Partitioner Algorithm.

```
 1: function HomogeneousPartition($image$, $initialPart$, $costFunc$, $n$)
 2:     $expectedCost \leftarrow costFunc(image)/n$
 3:     $partitions \leftarrow initialPart$
 4:     RT::sortByCost($partitions$, $costFunc$)
 5:     while $partitions.size() < n$ do
 6:         $expPart, remPart \leftarrow$ RT::BTS($partitions.pop()$, $expectedCost$)
 7:         $partitions.orderedInsert(expPart, costFunc)$
 8:         $partitions.orderedInsert(remPart, costFunc)$
 9:     end while
10:     return $partitions$
11: end function
```

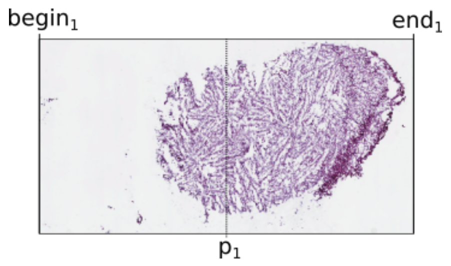
In this process the Binary-search Tile Splitting Algorithm (BTS) developed here (see Algorithm 9) is executed. A cut pivot p is initially set in the midpoint of the current region (*image*), creating two regions $[d_i, p]$ and $[p{+}1, d_f]$ (line 2), which have their costs compared (line 15). If the cost difference is smaller than a given error (imbalance among these two partitions), the process stops and two partitions are returned (line 16). Otherwise, p is updated to the midpoint of the region with the highest cost (lines 7-13) and checked again. This process continues until an acceptable cost difference is found. If the expected cost is impossible to be achieved, the search stops at the closest value, when *pivotLength* reaches zero (line 15).

For an arbitrary number d of dimensions, BTS can be executed on any of the d orthogonal coordinates. In order to reduce the ghost zone overhead in applications that use this strategy, the algorithm should return a partitioning with the least Sum of Perimeters (SoP) of the output tiles. Greater SoP values result in larger ghost zones, increasing I/O and processing costs. BTS solves this by partitioning only the largest current tile dimension (see Algorithm 9, line 2).

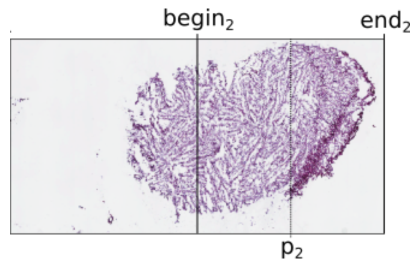
Algorithm 9 Binary-search Tile Splitting Algorithm (BTS).

```
1:  function BTS($image$, $expectedCost$)
2:      $(d_i, d_f) \leftarrow$ (min value of largest dimension, max value of largest dimension)
3:      $p, pivotLength \leftarrow (d_f - d_i)/2$
4:      do
5:          $aTileCost \leftarrow$ cost of $image$ in the current dimension interval $[d_i, p]$
6:          $bTileCost \leftarrow$ cost of $image$ in the current dimension interval $[p+1, d_f]$
7:          if $aTileCost > bTileCost$ and
8:              $expectedCost > bTileCost$ and
9:              $aTileCost > expectedCost$ then
10:             $p \leftarrow p - pivotLength$
11:         else
12:             $p \leftarrow p + pivotLength$
13:         end if
14:         $pivotLength \leftarrow pivotLength/2$
15:     while $aTileCost$ or $bTileCost$ not close enough to $expectedCost$ and
        $pivotLength > 0$
16:     return $(d_i, p), (p+1, d_f)$
17: end function
```

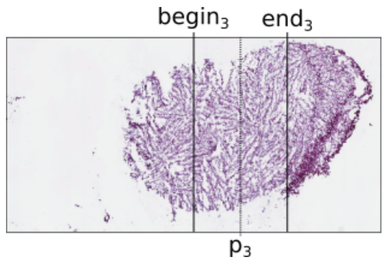
Figure 5.3 shows the BTS process on a region with expected cost of 37 on an initial image with 70 units of cost, assuming 10% imbalance upper limit error. This means that a tile with a cost in the interval [34,41] is searched. The initial split point p_1 on Figure 5.3a represents a partition cost ratio of 15/55. As none of the current partitions' cost are in the expected error interval the search proceeds to the next pivot p_2, with 45/25 cost partition. The process then stops when the desired tile is found in the partition 30/40 of Figure 5.3c, returning the second tile as the one closest to the expected cost. Although



(a) Initial split test with p_1 results in 15/55 cost partition.



(b) The next pivot p_2 is moved to right (costly side), leading to 45/25 partition.



(c) After updating end_3 and p_3 a cost ratio of 30/40 (left/right of p_3) found and the algorithm stops, returning the tile in the right.

Figure 5.3: An example partitioning with at upper limit of 10% error on a tile with expected cost of 37. Initial bounds $begin_1$ and end_1 image limits are defined and tightened after each split attempt until a partition with imbalance smaller than the target is found.

this maximum error margin is configurable, we have used a default of 2% since smaller errors did not result in significant performance differences.

5.1.3 Data Partitioning for Hybrid Environments

This section discusses the extensions to the data partitioning algorithm presented in previous section to support hybrid machines equipped with CPUs and GPUs. This strategy receives as input the initial regions to be partitioned, cost-function, number of partitions to be generated to CPU and GPU, and the expected GPU acceleration as compared to the CPU. Further, the full expected cost of the input data is computed and it then estimates the cost for partitions that each device available (CPU or GPU) should receive. This cost is proportional to the devices' relative performance, mostly, expected GPU vs. CPU speedup (lines 3 and 4 of Algorithm 10). We assume that the GPU acceleration (*gpuAcc*) is obtained in a profiling phase before the actual application execution and is provided to the system by the user. In this work, for instance, we executed the CPU-based and GPU-based versions of the application codes on the smallest input image (Figure reffig:wsi0) to collect this value, which was then employed for the rest of the experiments.

After that point, partitions are sorted according to their cost, and the next partition will always take place on the costly region until the desired number of data partitions

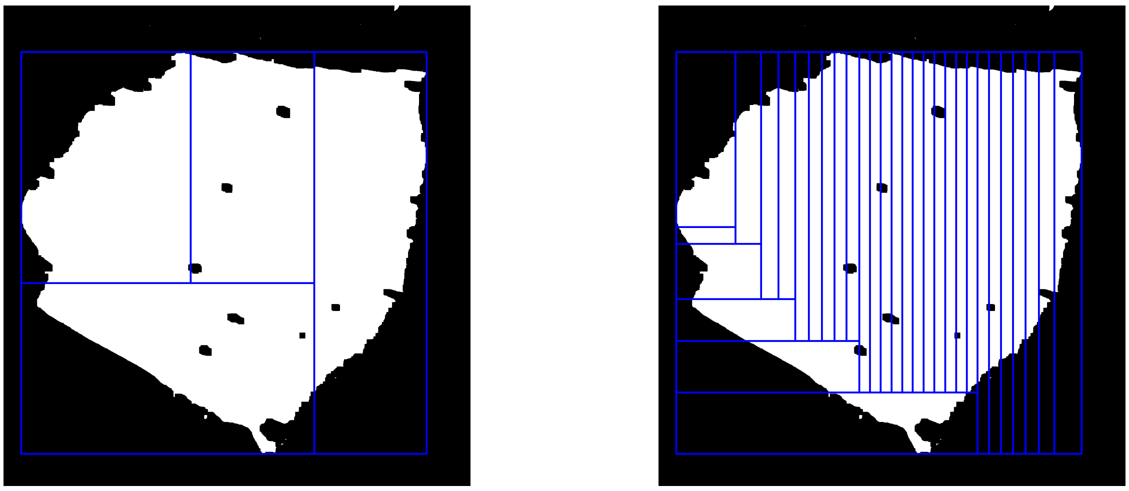
Algorithm 10 Hybrid Partitioner Algorithm.

```
1: function HYBRIDPARTITION($denseParts$, $costFunc$, $nCpu$, $nGpu$, $gpuAcc$))
2:     $fullCost \leftarrow \sum costFunc(part)\ \forall\ part$ in $denseParts$
3:     $gpuCost \leftarrow fullCost * (gpuAcc)/(nCpu + nGpu * gpuAcc)$
4:     $cpuCost \leftarrow fullCost * 1/(nCpu + nGpu * gpuAcc)$
5:     RT::SORTBYCOST($denseParts$, $costFunc$)
6:     $i \leftarrow denseParts.size()$
7:     for $count \leftarrow 0$; $i \leq nGpu + nCpu$; $count++$ do
8:         if $count < nGpu$ then
9:             $expCost \leftarrow gpuCost$
10:        else
11:            $expCost \leftarrow cpuCost$
12:        end if
13:        $expPart, remPart \leftarrow$ RT::BTS($denseParts.pop()$, $expCost$)
14:        $finalPartitions.pushBack(expPart)$
15:        $denseParts.orderedInsert(remPart)$
16:    end for
17:    for each $remPart$ in $denseParts$ do
18:        $finalPartitions.pushBack(remPart)$
19:    end for
20:    return $finalPartitions$
21: end function
```

($nCpu + nGpu$) is reached. The first $nGpu$ partitions created are appropriated to the GPU, each with approximately $gpuCost$. In this process, the BTS algorithm presented in previous section is used to find the tile cut point that results in the desired cost based on the $gpuAcc$ parameter. When it occurs, the partition is performed and the tile with desired cost is inserted in the final set of partitions and marked for execution with the proper target device. This process is repeated to generate the required number of partitions.

5.2 The Background Removal Bisection (BRB) CADP Algorithm

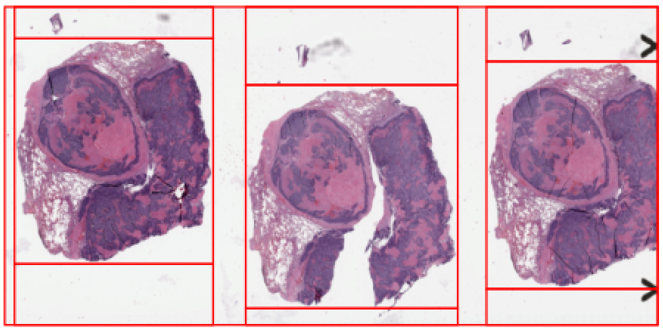
Although the ECB algorithm proved effective for reducing overall load imbalance for partitioning the input WSI images (see Section 6), it also showed few avenues for improvement. When initially tested, the partitions generated by the ECB algorithm were visually reasonable, resulting in partitions which could be generated manually, as shown in Figure 5.4a. However, when scaling the numbers of partitions to be generated a trend was observed. As seen in Figure 5.4b, the partitions generated were long thin strips of the image. This is due to the nature of ECB algorithm. ECB extracts a single partition a time with the expected cost. For 32 partitions, as on Figure 5.4b, each partition would have 1/32 of the overall image cost. Thus, only such thin strips could be generated from the original image (one of the four lateral strips, on top, bottom, left and right), even with the ECB heuristic to prefer more square-shaped partitions. This *strip partitions* phenomenon was later correlated with an increased cost of executing such partitions when compared to more square-shaped partitions. This increased execution cost is related to the Irregular Wavefront Propagation Pattern (IWPP) section of the motivating application. On the worst-case scenario for IWPP, a pixel can be propagated from one end of the image to the other. Assuming two images with the same area, one square and one long and rectangular, and retrieving the longest distance between two pixel in both images, the latter would have the greatest value. And this could impact on the number of propagation operations performed by the IWPP algorithm, increasing this value and consequently its cost. With these considerations, an improvement to restrict the generation of these strip partitions is wanted.


(a) ECB generating 4 partitions.

(b) ECB generating 32 partitions.

Figure 5.4: Two cases of ECB partitioning. Although the partitions cost was similar, the shapes for generating 32 partitions are mostly long and thin strips.

Another opportunity of improvement for the CADP lies on removing more background at a finer-grain. As shown in Figure 5.6, the method proposed above on Section 5.1.1 can only remove so much background area. Every partition generated by the CADP must be rectangular. The dense regions however, are not expected to fit this shape neatly. This limitation of the proposed coarse-grain background removal algorithm results in a missed opportunity to remove the additional background inside each dense region, as portrayed in Figure 5.5b.


(a) Result of coarse-grain background removal, which generates rectangular dense regions.

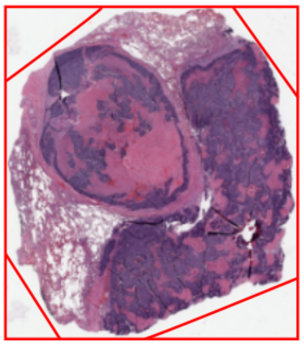
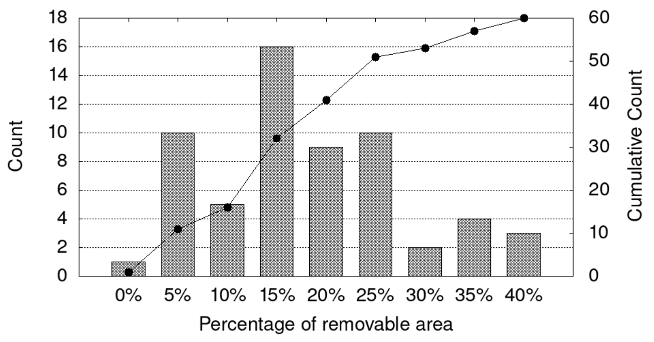
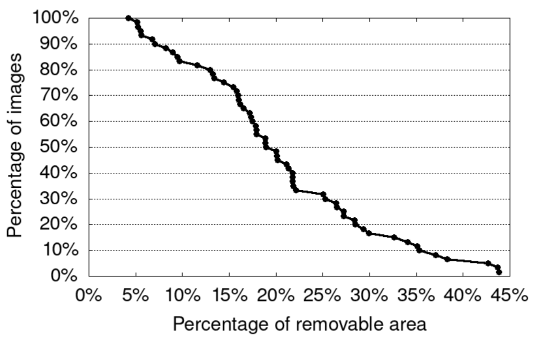

(b) The triangular background regions are not trivially removable.

Figure 5.5: Although the coarse-grain background removal algorithm can remove most of the background, some of it still remains inside the dense rectangular regions.

At first, a viability study was performed to assess how much background could be feasibly removed from such WSIs. From a set of 10 images each of 6 different class of WSI, the available fine-grain background was partitioned manually for later compilation of results. Only background which could be trivially removed from the vicinity of the dense region was considered (i.e., background inside dense regions was not considered). Then, the total sum of removable background area was evaluated with regard to the total area, shown in Figure 5.6a. As seen, the amount of background viable for removal is significant, with over half of all images having at least 16% of removable area (Figure 5.6b), thus indicating the viability of such approach.



(a) Histogram of amount of removable background on the left scale, with the cumulative count on the right scale.



(b) Accumulated counts of how many images had at least a certain amount of removable background. E.g., over 30% of the images had at least 25% of removable background.

Figure 5.6: Viability study for fine-grain background removal using 60 images.

5.2.1 A Method for Hierarchical Partitioning with Background Removal

Two issues should be addressed by the new BRB algorithm, improving on ECB: the improvement on the shape of the partitions and the fine-grain removal of background. As early described, ECB generates thin strip partitions when the number of expected partitions is high. The BTS algorithm however proved to generate reasonable partitions for smaller numbers of expected partitions with ECB. As such, the problem of generating a large number of partitions without the *strip partitions* issue is harder than for generating a smaller number of partitions. With this in mind, BRB aims to reach the same number n of partitions, but through the partitioning with smaller intermediate numbers of expected partitions. One way to achieve the n expected partitions is by extracting the factors of n and performing a hierarchical partitioning for each of these factors. By using the smallest factors of n the partitioning algorithm reduces a hard problem of partitioning a large value of n into multiple small easier problems of partitioning for smaller numbers. Coincidentally, these factors are the prime factors of a number. For instance, the partitioning of an image into 6 partitions can be performed by first partitioning the full image in 2 (see Figure 5.7) to then partition the resulting partitions in 3 (see Figure 5.7b) since $(2, 3)$ are the prime factors of 6. Though this hierarchical process the correct expected number of 6 partitions is reached. However, the problem of generating 6 partitions is reduced to the partitioning with 2 and 3 expected partitions, which are easier to generate better partitions which are not strips.

The next issue to be addressed by BRB is the fine-grain background removal. Initially, the algorithm had a background-removal-centric approach, on which it should remove the most amount of background possible, to later solve balance issues, if any. On this

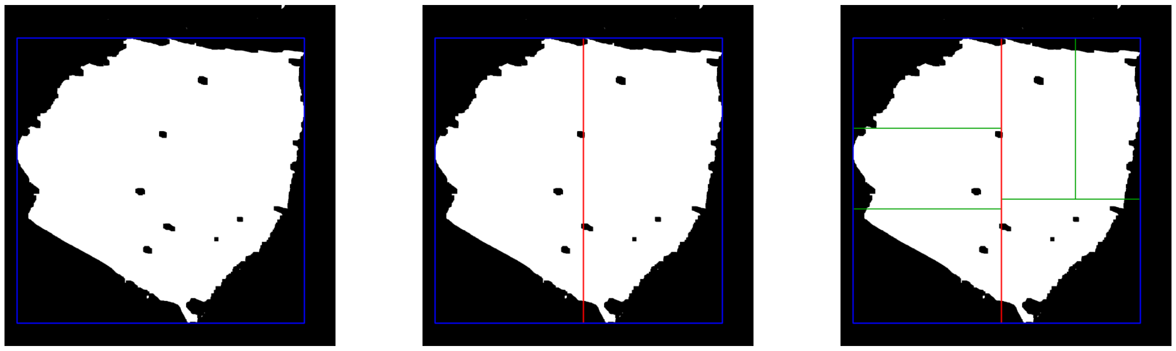
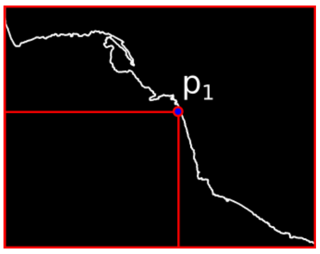
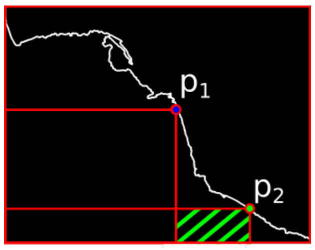

(a) Initial single partition. (b) Partitioning result for 2 expected partitions. (c) Partitioning result for 6 expected partitions.

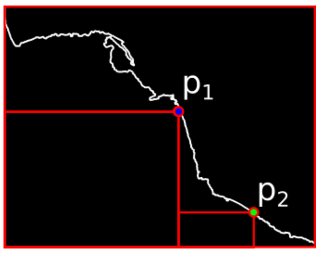
Figure 5.7: Hierarchical partitioning for 6 expected partitions. First, 2 partitions are generated (red) from the single initial dense partition (blue). Then, 3 more partitions (green) are required by each of the previous 2, resulting in 6 partitions.



(a) First background tile generated by creating a rectangle from a point in the dense region contour and the corner of the full image (lower left).



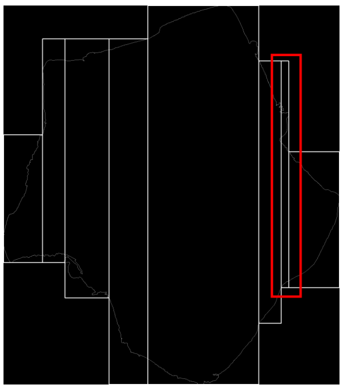
(b) A point on the contour for the second background tile is found, however, it overlaps with the previously generated background tile.



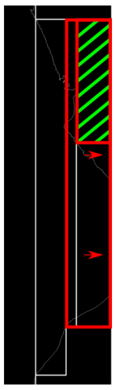
(c) Overlapping with the newly generated background tile is removed, resulting in a second background tile.

Figure 5.8: First approach for background removal. The contour of the dense region is delineated in white, being the background on the lower left corner of the image. Two background tiles are expected to be generated. Background tiles generated are expected to maximize their area.

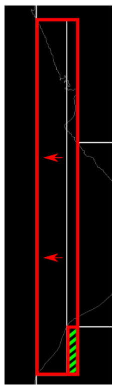
approach a dense region contour would be generated as a convex-hull. From this contour it is possible to generate a rectangle between a point on it and one of the four corners of the image, as shown in Figure 5.8a. By testing all points in the contour, the tested background tile with the largest area would then be chosen. This point selection process could be iterated over again, as many times as needed, removing more background each time (see Figure 5.8). As such, the amount of background to be removed would be proportional to an input parameter of how many tiles should be generated.



(a) All dense partitions are generated, however, there is a partition which is rather small, and should probably be merged with another partition.



(b) First alternative, merge with the right partition.



(c) First alternative, merge with the left partition, adding back less background.

Figure 5.9: Generation of partitions which result in removing previously generated background tiles. The partition highlighted in red is considered too small, and as such should be merged with another partition.

From the background tiles previously generated the actual dense partitions would then be created, as depicted on Figure 5.9a. They can be generated by an algorithm similar to Algorithm 7. The first issue encountered with this approach is the possibility of creating rather small partitions, highlighted in red in Figure 5.9a. This is problematic since there are overheads regarding the execution of a partition, and the execution time of such partitions would be dominated by these overheads. A solution for this was the merger of these small partition to one of its neighbors. By merging them, a new partition covering both would be generated. This new partition would also add some of the removed background, and thus the selected neighbor would be the one which added the least amount of background (see Figures 5.9b and 5.9c).

This early background-removal-centric resulted in highly imbalanced partitions, being slower to execute then ECB on preliminary tests. The reason this algorithm was problematic was twofold: (i) it required the optimization of the parameter of how many background tiles should be generated, and (ii) these tiles could result in poorly shaped dense partitions, which only after many re-partitioning improve its imbalance. With this knowledge, the BRB algorithm would be partition-centric, with the background removal done as an afterthought to further reduce executing costs.

The final algorithm with both the hierarchical partition generation and background removal is presented on Algorithm 11. Similar to ECB, a list of partitions is maintained (line 2), being incremented once at a time with a new partition (line 12). However, in order to allow the hierarchical partitioning the expected number of partitions *nPartitions* is decomposed into its prime factors. From each iterated prime factor the number of expected

Algorithm 11 BRB Homogeneous Partitioner Algorithm.

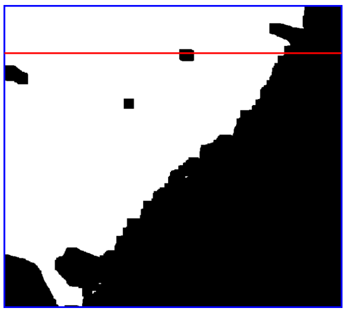
```
 1: function BackgroundRemovalBissection($dense$, $nPartitions$)
 2:     $allParts \leftarrow dense$
 3:     $primeFactors \leftarrow getPrimeFactors(nPartitions)$
 4:     $partialNumPartitions \leftarrow 1$
 5:     for $multiple$ in $primeFactors$ do
 6:         $partialNumPartitions \leftarrow partialNumPartitions * multiple$
 7:         $sumOfCosts \leftarrow$ cost of all partitions on $allParts$
 8:         $expectedCost \leftarrow sumOfCosts/partialNumPartitions$
 9:         while $allParts.size() < partialNumPartitions$ do
10:             $curPart \leftarrow allParts.pop()$
11:             $newParts \leftarrow$ BgRemovalBTS($curPart$, $expectedCost$)
12:             $allParts.orderedInsert(newParts)$
13:         end while
14:     end for
15:     return $allParts$
16: end function
```

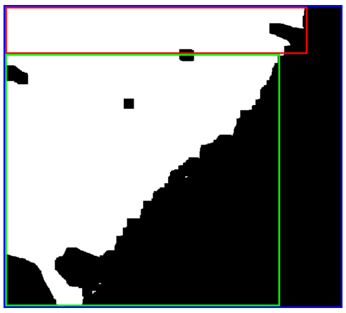
Algorithm 12 Simple Background Removal Algorithm for *BgRemovalBTS*.

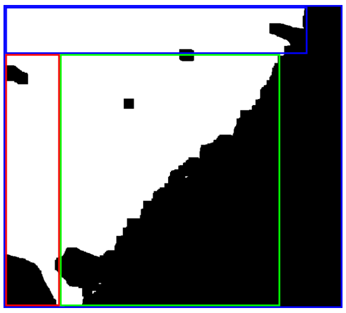
```
 1: function SimpleBgRem(image, costFunc)
 2:     $denseTiles$ ← list of rectangular dense regions of $image$
 3:     $bestXi \leftarrow image.Xo$
 4:     $bestXo \leftarrow image.Xi$
 5:     $bestYi \leftarrow image.Yo$
 6:     $bestYo \leftarrow image.Yi$
 7:     for $tile$ in $denseTiles$ do
 8:         $bestXi \leftarrow min(bestXi, tile.Xi)$
 9:         $bestXo \leftarrow max(bestXo, tile.Xo)$
10:         $bestYi \leftarrow min(bestYi, tile.Yi)$
11:         $bestYo \leftarrow max(bestYo, tile.Yo)$
12:     end for
13:     return $tile(bestXi, bestXo, bestYi, bestYo)$
14: end function
```

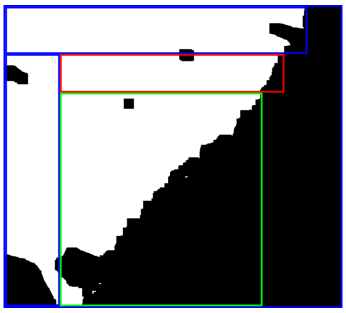
partitions is updated to the number of partitions required after the end of each iterations. For instance, if we were to generate 30 partitions the factors would be $(2, 3, 5)$. Assuming this current order, the first iteration should yield 2 partitions, with the following yielding 6 (3 for each of the previous 2) and 30 (5 for each of the previous 6) partitions respectively. Iterating through the factors (line 5), the expected cost for the first factor 2 would be the total cost divided by 2 (line 7). After the expected number of current partitions is generated (lines 9 - 13), the algorithm goes to the next prime factor 3, updating the expected number of partitions for the current iteration to 6 (line 6). Although the ordering of the prime factors does not influence on the resulting number of output partitions it may influence on the quality of them. It is reasonable to assume that an earlier iteration of the hierarchical partitioning have impact on the overall quality of the output partitions. Using the $(2, 3, 5)$ partitions example, if the first partitioning was inaccurate, generating two partitions a and b with relative cost of 10 and 20 respectively, the imbalance between a and b could not be resolved by the following partitioning of $(3, 5)$. As such, since it was shown that the partitioning for smaller numbers of expected partitions is an easier problem, the prime factors are sorted in a non-descending order.

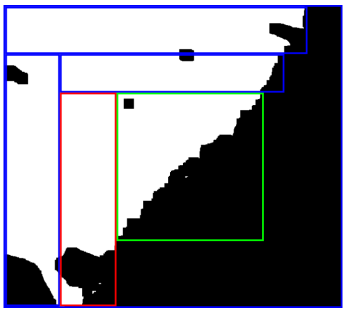
The final component of BRB is the background removal. This is done inside the BTS algorithm. Whenever a new tile is tested by BTS (i.e., have its cost calculated and compared), the background is first removed. The removal of background is simple, only being able to remove one or more of the 4 sides of the image, as depicted by Figure 5.10. The removal process is defined in Algorithm 12. First, a list of all dense regions of the tile is compiled (line 2). From this list, the best coordinates for the four borders of the tile is found by checking each dense sub-region. The final borders should include in its entirely all dense sub-regions. This updated version of BTS is used by Algorithm 11, on line 11.


(a) First cut for a partition if found by BTS.


(b) From the first cut, all background only region is removed. Background is also removed from the remaining partition in green.


(c) Second partition is generated. However no background can be removed without also removing important sections of the partition.


(d) Third partition is generated, also removing more background.


(e) Fourth partition generated.

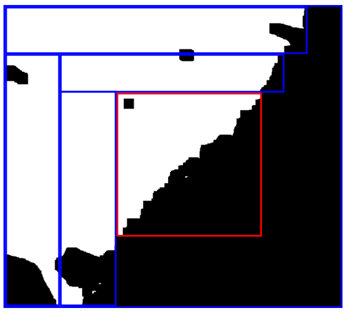
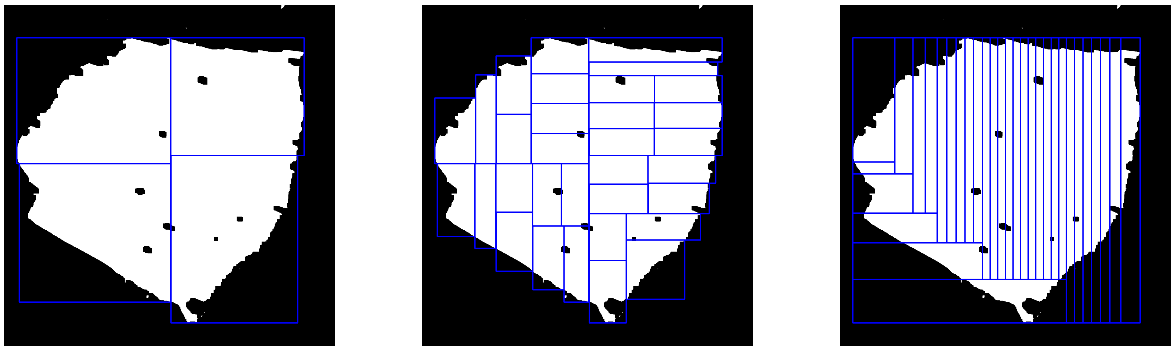

(f) The remaining of the image is the fifth partition.

Figure 5.10: Extraction of partitions with a background removal version of BTS. It is expected to generate 5 partitions. Red lines mark the current partition generated. The remaining partition is shown in green. Previously generated partitions are shown in blue. The black area is the background.

Figure 5.11 shows both cases for a small and large amount of expected partitions when using the BRB algorithm. When compared with the results of ECB from Figure 5.4, BRB manages to output more square-shaped partitions, while also removing fine-grain background regions. It is worth noting that the amount of background removed is proportional to the number of expected partitions. Also, as described in details on the next section, the cost-function estimation is also based on the area of a partition. As such, by removing background, the total cost of the initial image is greater than the final sum of background removed partitions. This may lead to an increased imbalance between partitions for BRB when compared to ECB. While imbalance was shown to be significant for the conducted experiments (see Section 6), the trade-off of removing background and thus reducing the overall execution cost against a reduced balance efficiency resulted in significant speedups.



(a) BRB partitioning for 4 expected partitions. (b) BRB partitioning for 32 expected partitions. (c) ECB partitioning for 32 expected partitions.

Figure 5.11: BRB partitioning algorithm for 4 and 32 expected partitions. ECB partitioning results for the same image are also shown as a comparison to BRB.

5.3 Data Region Cost-Functions

The cost-function is a crucial building block for an irregular partitioner to work properly. While it is important to develop functions that approximate the cost of the data regions well, it is also mandatory for them to be inexpensive in order not to offset the gains of CADP partitioning. In order to enable CADP to work with other applications, we have allowed for the cost-function to be a parameter to CADP that can be customized or developed by the user according to new applications added to the system.

In our target application domain, as previously discussed, the data domain processing cost is heterogeneous and vary according to the number and area of objects in a region. Therefore, it is important to develop a cost-function that approximates that metric and, consequently, correlates with the expected processing time. Computing a separate, expensive segmentation workflow to detect objects, such as nuclei and cells, would not be efficient, because such a workflow is already implemented in the application itself. Thus, we developed a simple and very efficient threshold-based segmentation cost-function to identify objects in the images. After the threshold our function counts the number of foreground pixels (area) that is used as our metric. This function was developed on top of Halide to exploit parallelism at instruction and thread levels (through Halide parallel directives).

The proposed cost-function is not compute intensive, but it could become costly if applied to the full high resolution images as it would be necessary to read the entire data. In order to avoid this cost, we took advantage of the WSIs pyramidal representation natively available in the used microscopy images. It consists of the images being stored by default at multiple magnifications, and we use a low resolution version of the images as input to the cost-function to minimize its execution time. The WSIs are are processed at 40× magnification (.25 micrometers/pixel) by the use-case application. To execute our threshold based segmentation cost-function efficiently, we always used images with the lowest resolution available. This approach resulted in a resolution reduction from 1024× to up to 4096× for such images. We found that using the lowest resolution images had no significant impact to the partitioning algorithm, while it reduced the execution times significantly. However, we note that this effect should be evaluated for each new cost-function to properly optimize the image resolution for the new cost-function.

5.4 Time Complexity Analysis of CADP

For the purpose of time analysis, the whole CADP partitioner can be defined as the following set of components: (i) dense bounding boxes generation, (ii) bounding boxes overlapping resolution, (iii) background tiles generation, (iv) dense partitioning, and (v) the cost-function. The calculations are done with regard to the number of pixels of the image (n), the maximum value between the image's height and width (m), the number of initial dense partitions (d) and the number of expected dense partitions (t). The image dimensions are related to the low-resolution version of the input image, used exclusively for partitioning.

5.4.1 Time Complexity Analysis of ECB

Background tiles are generated by the sequential composition of components (i-iii). The dense region bounding boxes are found through a standard OpenCV algorithm to find all connected components [121]. These are then passed through an erosion/dilation process to remove insignificant (small) regions. The connected components algorithm scales linearly with the number of pixels, as does the erosion/dilation: $\mathcal{O}(n)$. The d_1 generated tiles from (i) are checked for internal, sideways and diagonal overlapping cases. For all three cases a pairwise comparison between all d_1 tiles is required: $\mathcal{O}(d_1^2)$. The initial number of tiles can be reduced by internal overlapping resolutions or increased by diagonal resolutions, resulting in d_2 dense non-overlapping tiles. For simplicity, we define $d = max(d_1, d_2)$, resulting in (ii)'s complexity of $\mathcal{O}(d^2)$. The background partitions are generated by successive $MakeBlocks()$ calls on both initial and end coordinates of all d dense tiles (see Algorithm 7). On the worst case, $MakeBlocks()$ iterates through a full *open* list of tiles. Since $MakeBlocks$ iterating through d open tiles d times would mean that there are a sequence of d horizontal tiles and d vertical tiles, it would also mean that for d tiles there is a number of at least $2d$ tiles being traversed. Since this is impossible, (iii) can never be on the order of d^2 or greater, resulting in $o(d^2)$. Further, (iii) is $\Omega(d)$

given the two best cases of a single vertical or horizontal arrangement of tiles. Thus, the background generation section of CADP is $\mathcal{O}(n + d^2)$.

Whenever $d \geq t$ the dense partitioner returns the initial d tiles immediately, voiding the cost of (iv). Thus, we assume that $d < t$ in order to evaluate the worst case scenario. As shown in both Algorithms 8 and 10, a number of at most t dense tiles is generated by executing the BTS algorithm. BTS uses a recursive logarithmic checking pattern for pivots, always moving forward towards completion by halving the *pivotLenth* at each iteration (see lines 14 - 15 on Algorithm 9). Given that BTS iterates through completion on the worst case (i.e., when $pivotLength = 1$), it may have at most $log_2(m)$ pivots. For each pivot the cost-function is executed on the current tile, with the used function being $\Theta(n)$. As such, the dense partitioner executes in $\mathcal{O}(t\ n\ log_2(m))$. The final time complexity of CADP is then $\mathcal{O}(d^2 + t\ n\ log_2(m))$. Since by definition $m \leq n$ and $d < t$ (as per our initial assumption), CADP can further be simplified to $\mathcal{O}(t^2 + t\ n\ log_2(n))$, with t as the number of expected partitions and n as the number of pixels in the input image.

5.4.2 Time Complexity Analysis of BRB

The main differences between ECB and BRB are hierarchical partitioning and the background removal done on BTS. Regarding the hierarchical partitioning, although the order of partitioning, the amount of partitions generated by BRB is equal in any case to the number of partitions when using ECB. Regarding the updated BTS, the background removal cost of Algorithm 12 is insignificant. Asymptotically, its cost is proportional to the number of dense sub-regions found on a partition. Although this number is rarely greater than 1, the execution cost of performing 4 *min/max* operations is negligible. Thus, the changes on BTS for BRB are not significant enough for its performance to differ from its previous ECB version. Since both changes do not impact the cost of generating a partition or the number of partitions generated (i.e., calls to BTS), the time complexity of BRB is equivalent to the one of ECB

Chapter 6

Experimental Results

This chapter presents the performance evaluation of the proposed algorithms for the distributed execution solution. All experiments were conducted with two types of compute nodes, a CPU-only node and a CPU-GPU node. The former is composed of dual-sockets Intel(R) Xeon(R) Gold 6252 CPUs (24 cores per CPU, 48 per node) with over 370 GB of RAM. The CPU-GPU node, is similarly configured to the CPU-only node with 4 NVIDIA Tesla V100 GPUs. Each V100 GPU with 32 GB of dedicated memory. The experiments used the segmentation phase of the brain cancer studies image analysis application (see Section 2.4) and input data consists of a selection of The Cancer Genome Atlas (TCGA) whole slide tissue images [84] downloaded from the Genomic Data Commons Data Portal (GDC) [83]. A randomly sampled subset of 10 images from a set of 60 images of 6 different cancer types was chosen, as seen on Figure 6.1 and Table 6.1. The chosen images can be classified in two main classes of WSIs: dense (e.g., Figures 6.1a and 6.1h), and sparse (e.g., Figures 6.1d and 6.1e). Dense WSIs are those with a single large and contiguous tissue Region of Interest (RoI) that occupies most of the image area. In contrast, sparse WSIs may contain multiple smaller regions with significant background area. This value was then employed in the partitioning in hybrid environments in the rest of the experiments when processing other images. In all of data partitioning, a ghost zone of 100 pixels was used since it is sufficient to include objects (nuclei) within its borders [109]. All execution times reported refer to the application end-to-end execution times, which includes both I/O and processing times. This is also referred on this text as the application makespan. Initially both partitioning algorithms KD-Tree and Quad-Tree (see Section 3.1) were considered as baselines. However, the Quad-Tree algorithm performed consistently worse than all other approaches, including KD-Tree. As such, in order to improve the presentation of the results, only the KD-Tree algorithm (KDT) was shown as the baseline. All tested algorithms were fed the same inputs, with a goal of 4 partitions per compute resource (e.g., 8 CPU-only nodes would receive 32 partitions).

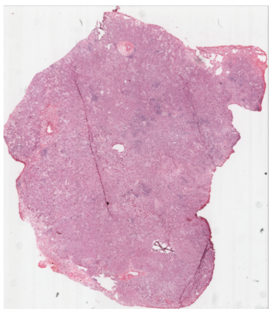
Every test conducted, for each point or configuration, was executed 5 times. For each point, the median makespan was chosen, with its corresponding values for other metrics (e.g., max task time, sum of task times). Further, assuming a normal distribution of samples for each set of 5 executions, it was found that almost all performance comparisons (speedups) between algorithms were significantly different for $p = 0.05$, with only around 18 out of over 1200 test points not being statistically different. Also, no error bars are shown on the graphs since they are too small to be visible (also using $p = 0.05$) and a significant difference was already statistically established.

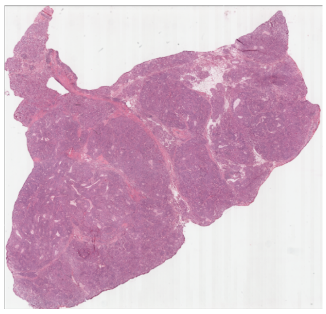
Three main metrics were evaluated across all experiments: speedups, scaling efficiency and balance efficiency. The two former metrics are already well known in HPC. Regarding the latter metric, it was important to quantitatively assess how workload imbalance impacts overall performance. Based on the Percentage of Imbalance metric (λ [122]) a new metric was proposed, the Balance Efficiency. Similar to the equation of Percentage of Imbalance $\lambda = (L_{max}/\bar{L} - 1) \times 100\%$, with L_{max} being the maximum load (the makespan or total time for this domain) and $\bar{L}$ being the average load (average of workers' times), the Balance Efficiency is calculated as $(\bar{L}/L_{max}) \times 100\%$. The ratio L_{sum}/L_{max}, being L_{sum} the sum of all workers' times, approximates the speedup of parallel execution when compared to serialized execution (not considering distribution overheads). Ideally, this speedup should be equal to the number of parallel resources, meaning that the workload is perfectly balanced. Thus, by calculating $(L_{sum}/L_{max})/n_{resources}$ we have an efficiency metric which can also be calculated as $(\bar{L}/L_{max}) \times 100\%$ and is bounded by the range $(0\%, 100\%]$.

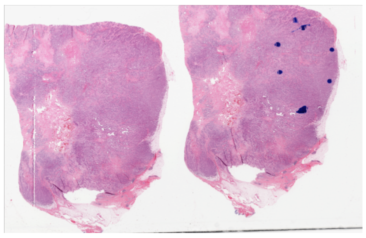
Given that 10 images were tested it is unfeasible to show the performance charts for all of them, for each experimental setting. As such, two images (BR-8682 and B6-A0X1) have their data plotted for each setting. These images were chosen since they represent the two main classes of images, dense: with only a single contiguous region of interest and not much background, and sparse: with one or more contiguous regions of interest and higher ratios of background. Also, speedups, balance and scaling efficiencies are fully displayed on tables for each experimental setting, for completeness.

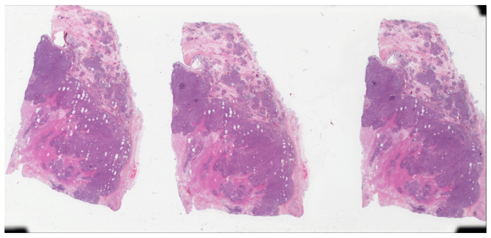
6.1 Single Node Evaluation

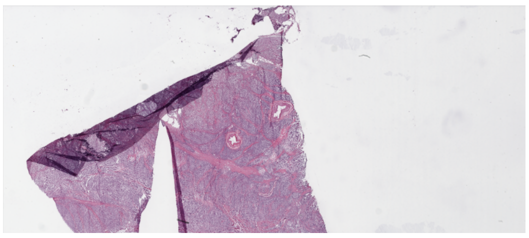
This section evaluates the performance of the proposed system with the motivating application into a single node. Section 6.1.1 compares handwritten code based on OpenCV to automatically generated code using Halide in a sequential execution using CPU. It also evaluates the CPU multi-core scalability of the Halide based code. Section 6.1.2 evaluates

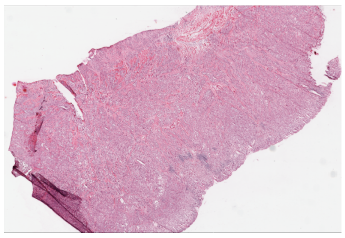

(a) 86-8668 $(87,582 \times 76,160)$.

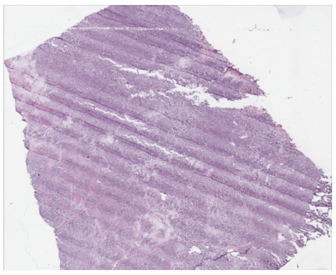

(b) 86-8669 $(90,623 \times 95,200)$.

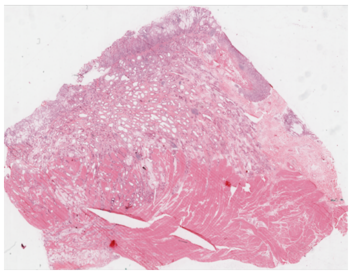

(c) B6-A0RG $(93,657 \times 147,468)$.

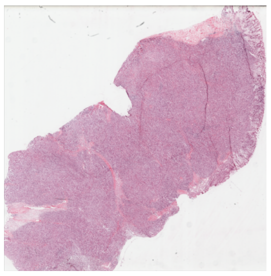

(d) B6-A0X1 $(93,358 \times 198,220)$.


(e) BR-8285 $(70,894 \times 163,344)$.


(f) BR-8296 $(70,894 \times 105,576)$.


(g) BR-8361 $(70,894 \times 87,648)$.


(h) BR-8682 $(77,870 \times 100,912)$.


(i) FA-8693 $(92,241 \times 91,392)$.

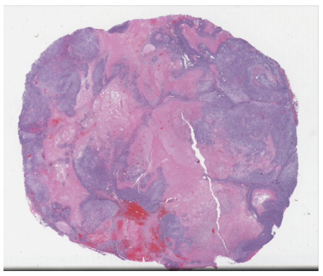

(j) LL-A8F5 $(90,462 \times 107,567)$.

Figure 6.1: All WSIs images used for the experimental evaluations. Each image has its ID and their resolution.

Table 6.1: Information on all images used for the experiments.

TCGA GDC Image Identifier	Database Image Type	Resolution
TCGA-86-8668-01A-01-TS1	Lung Adenocarcinoma	$87,582 \times 76,160$
TCGA-86-8669-01A-01-TS1	Lung Adenocarcinoma	$90,623 \times 95,200$
TCGA-B6-A0RG-01Z-00-DX1	Breast Invasive Carcinoma	$93,657 \times 147,468$
TCGA-B6-A0X1-01Z-00-DX1	Breast Invasive Carcinoma	$93,358 \times 198,220$
TCGA-BR-8285-01A-01-TS1	Stomach Adenocarcinoma	$70,894 \times 163,344$
TCGA-BR-8296-01A-01-TS1	Stomach Adenocarcinoma	$70,894 \times 105,576$
TCGA-BR-8361-01A-01-BS1	Stomach Adenocarcinoma	$70,894 \times 87,648$
TCGA-BR-8682-01A-01-TS1	Stomach Adenocarcinoma	$77,870 \times 100,912$
TCGA-FA-8693-01A-01-TS1	Diffuse Large B-cell Lymphoma	$92,241 \times 91,392$
TCGA-LL-A8F5-01Z-00-DX1	Breast Invasive Carcinoma	$90,462 \times 107,567$

the performance in GPU only devices, and Section 6.1.3 evaluates the application into a hybrid CPU-GPU setting.

6.1.1 Comparison to handwritten and Multi-core scalability

We first compared the application written using Halide to the sequential handwritten code implemented in a previous work [22], both deployed into RT. When executing the code sequentially using a single CPU core, the RT/Halide code version took 1480.19 seconds, while the Handwritten code executed in 1654.28 seconds. Most of the performance gains in the Halide based application, as seen in details on Table 6.2, were concentrated on Erode/Dilate operations executed within the Morphological Reconstruction, which were implemented in the original Handwritten application leveraging OpenCV [55]. These operations execute pixel neighborhood or stencil operations with different stencil shapes and sizes. This is a computing pattern friendly for optimization with Halide that was able to generate hardware specific tiling, improving cache locality and performance although using a single CPU core. This result shows that the Halide enabled Region Templates code is not only higher-level, but it may be more efficient than code base on OpenCV. This is one of the reasons Halide is being used as a backend for some stencil-related functionalities/operations in OpenCV [123, 124]. It should be noted that although using Halide resulted in better performance, being this case expected in most cases, its use may eventually result in slight slowdowns [111].

Table 6.2: Profiling of the pipeline's tasks on a serialized environment for Halide and Handwritten code. I/O times not considered.

Task	GetRBG	Erode	Dilate	IWPP	Dilate2	Erode2	Full time
RT/Halide Code	3.37	414.71	288.29	761.85	5.16	5.69	1480.19
Handwritten Code	3.64	450.53	378.45	810.00	4.63	5.90	1654.28

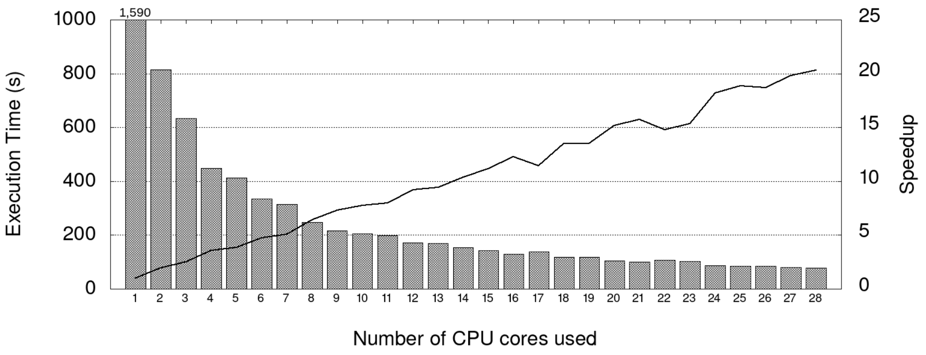


Figure 6.2: Scalability of the motivating application developed using Halide on a single CPU with 28 cores.

Then the scalability for the number of CPU computing cores was further analyzed when using the Halide parallelized code. The results presented in Figure 6.2 reached a 76% scaling efficiency with 28 CPU cores on a CPU-only machine. Sub-linear scalability is, however, expected for the application because of the irregular computation costs in different image regions, memory subsystems competition, and additional costs to synchronize threads and I/O times that do not decrease linearly with the number of CPU cores used. These results agree with previous works on the application domain and Halide [22, 125].

6.1.2 Multi-GPU Execution

This experiment evaluates the performance of our system for multi-GPU compute nodes. This setting compares the CADP algorithms with the baseline KDT algorithm. For each GPU used, a RT Worker was instantiated. For all the cases evaluated here a single GPU could not store the entire image, which was partitioned for out-of-core GPU processing, with 4 partitions per GPU device. The raw size of some of the input images used in this section could reach over 20 GB in size (as shown in Table 6.1) which is larger than the memory of a single GPU. Within a single compute node, 1-4 GPU devices were used on a scaling setting. The performance of all algorithms regarding speedups, scaling and balance efficiency are displayed on the tables of Figure 6.3. Since most results are similar between WSIs, two images were chosen for a deeper analysis: BR-8682 and B6-A0X1.

As shown on the results of Figure 6.3, ECB achieved speedups of 0.88-1.93×, with BRB improving it to 0.97-2.72×, both compared with KDT. Regarding scaling efficiency, all algorithms achieved good efficiency, with only 3/60 cases for both ECB and BRB below 90% efficiency. However, the evaluated scaling efficiency for such few distributed workers was expected. Regarding the worse results from KDT, one source of lower scaling efficiency is the also lower balance efficiency, which is expected to deteriorate even fur-

Figure 6.3: Full data for single node experiments executed on a single GPU-only compute node. For each image, the values are related to the number of compute elements used, from 1-4 GPUs for the used node.

Speedups for ECB vs. KDT

Image	1	2	3	4
86-8668	0.90	1.15	0.99	1.06
86-8669	0.91	1.12	1.01	1.40
B6-A0RG	1.00	1.00	1.07	1.35
B6-A0X1	1.41	1.28	1.38	1.44
BR-8285	1.94	1.64	1.46	1.52
BR-8296	0.97	1.12	0.95	1.29
BR-8361	1.00	1.03	1.06	1.11
BR-8682	1.09	1.08	1.07	1.22
FA-8693	0.95	1.12	0.89	1.28
LL-A8F5	1.06	1.12	1.06	1.23

Speedups for BRB vs. KDT

Image	1	2	3	4
86-8668	1.03	1.38	1.27	1.39
86-8669	1.12	1.37	1.37	1.87
B6-A0RG	0.98	1.04	1.14	1.13
B6-A0X1	1.42	1.34	1.48	1.66
BR-8285	2.51	2.19	2.17	2.72
BR-8296	1.10	1.30	1.18	1.59
BR-8361	1.04	1.15	1.15	1.24
BR-8682	1.18	1.15	1.23	1.42
FA-8693	1.30	1.38	1.29	1.84
LL-A8F5	1.16	1.10	1.11	1.24

Speedups for BRB vs. ECB

Image	1	2	3	4
86-8668	1.15	1.20	1.29	1.31
86-8669	1.24	1.23	1.36	1.34
B6-A0RG	0.98	1.04	1.06	0.84
B6-A0X1	1.00	1.04	1.08	1.15
BR-8285	1.30	1.34	1.49	1.79
BR-8296	1.14	1.16	1.24	1.23
BR-8361	1.04	1.11	1.08	1.11
BR-8682	1.09	1.07	1.15	1.17
FA-8693	1.37	1.23	1.46	1.43
LL-A8F5	1.09	0.98	1.04	1.01

Scaling Efficiency for ECB

Image	2	3	4
86-8668	113%	104%	102%
86-8669	120%	107%	107%
B6-A0RG	105%	105%	109%
B6-A0X1	98%	98%	95%
BR-8285	100%	90%	80%
BR-8296	96%	91%	85%
BR-8361	102%	94%	90%
BR-8682	100%	100%	97%
FA-8693	108%	94%	96%
LL-A8F5	101%	99%	99%

Scaling Efficiency for KDT

Image	2	3	4
86-8668	88%	94%	86%
86-8669	97%	96%	69%
B6-A0RG	105%	98%	81%
B6-A0X1	107%	100%	93%
BR-8285	119%	120%	102%
BR-8296	83%	92%	64%
BR-8361	99%	89%	81%
BR-8682	101%	102%	86%
FA-8693	92%	101%	71%
LL-A8F5	96%	99%	86%

Scaling Efficiency for BRB

Image	2	3	4
86-8668	117%	116%	116%
86-8669	119%	118%	116%
B6-A0RG	112%	115%	94%
B6-A0X1	101%	105%	109%
BR-8285	104%	104%	111%
BR-8296	98%	99%	93%
BR-8361	109%	98%	96%
BR-8682	98%	106%	104%
FA-8693	98%	100%	100%
LL-A8F5	91%	94%	92%

Balance Efficiency for ECB

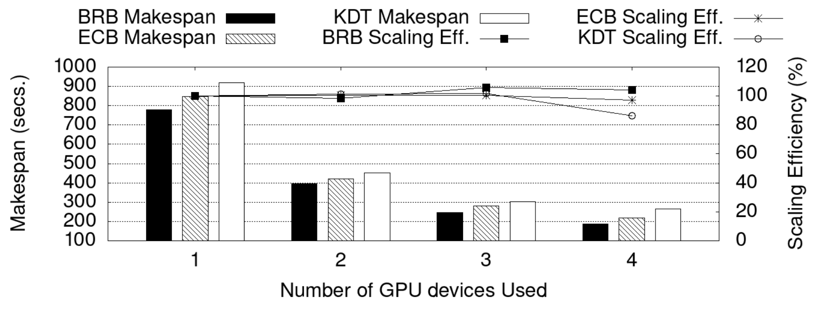
Image	2	3	4
86-8668	98%	93%	91%
86-8669	100%	95%	96%
B6-A0RG	96%	94%	97%
B6-A0X1	93%	99%	97%
BR-8285	98%	94%	93%
BR-8296	99%	97%	97%
BR-8361	100%	99%	97%
BR-8682	97%	99%	96%
FA-8693	100%	98%	97%
LL-A8F5	98%	96%	97%

Balance Efficiency for KDT

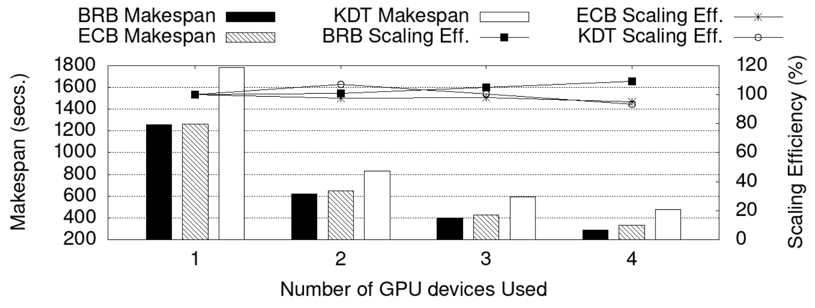
Image	2	3	4
86-8668	86%	94%	88%
86-8669	89%	90%	66%
B6-A0RG	92%	91%	76%
B6-A0X1	97%	95%	89%
BR-8285	92%	94%	84%
BR-8296	83%	98%	68%
BR-8361	98%	93%	87%
BR-8682	95%	98%	83%
FA-8693	86%	98%	68%
LL-A8F5	89%	95%	84%

Balance Efficiency for BRB

Image	2	3	4
86-8668	97%	96%	93%
86-8669	97%	94%	92%
B6-A0RG	98%	88%	79%
B6-A0X1	94%	94%	98%
BR-8285	93%	92%	93%
BR-8296	96%	91%	88%
BR-8361	99%	95%	95%
BR-8682	91%	95%	95%
FA-8693	89%	92%	89%
LL-A8F5	90%	95%	95%

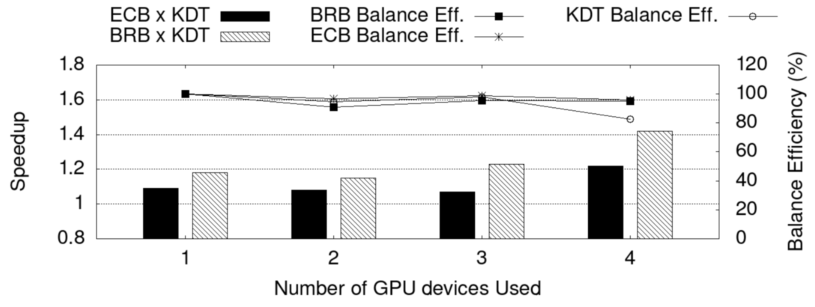


(a) Results for dense image BR-8682.

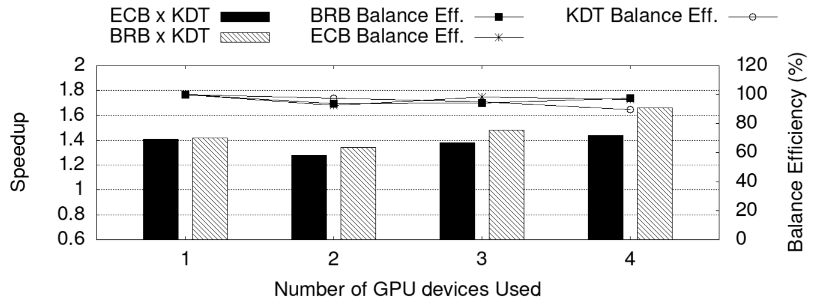


(b) Results for sparse image B6-A0X1.

Figure 6.4: Application makespan of images BR-8682 and B6-A0X1 with related scaling efficiency for 1 to 4 GPUs used.



(a) Results for dense image BR-8682.



(b) Results for sparse image B6-A0X1.

Figure 6.5: Application makespan of images BR-8682 and B6-A0X1 with related scaling efficiency for 1 to 4 GPUs used.

ther its performance for larger numbers of workers. Another interesting aspect regarding balance efficiency is that BRB improved ECB's performance by 0.84-1.79× even with 25/30 multi-GPU results with worse balance efficiency. As mentioned on Section 5.2.1, it was expected for BRB to perform worse than ECB regarding balance efficiency, but improving performance due to background removal. Section 5.2.1 also show how much fine-grain background can be removed, while Section 6.3 shows experimental results regarding background removal and its impacts.

The images BR-8682 and B6-A0X1 were chosen as they represent the two main classes of images, dense and sparse, respectively, with their results shown in Figures 6.4 and 6.5. For image BR-8682 the speedups of BRB compared to ECB are greater than the ones for B6-A0X1. This is due to the amount of fine-grain background which can be removed. The first dense image have almost no coarse-grain background to be removed while also having a significant amount of background which can be easily removed by BRB. This difference allowed BRB to outperform ECB. For B6-A0X1, most of the background which could be removed is coarse-grain, bringing ECB closer to BRB. The speedups however increase with the number of GPUs, and thus partitions generated, allowing for more fine-grain background removal for BRB. Finally, the impacts of balance efficiency on speedups are more visible for these images. As shown in Figures 6.4a and 6.4b, as KDT balance efficiency drops, ECB and BRB speedups grow.

6.1.3 Cooperative CPU/GPU Execution

This section evaluates the benefits of cooperative execution on RT. As previously discussed, our platform can use multiple devices by partitioning the input image into disjoint tiles that are dispatched for processing in available processors. This partitioning is parameterized by the expected acceleration of the GPU as compared to the CPU to create partitions with size/cost that are proportional to their computing power. The speedup values of a GPU vs. CPU multi-core are presented in Table 6.3. For all cases and algorithms 4 partitions were generated per device (GPU or CPU) and an acceleration value of 1.6 was used for hybrid partitioning.

Table 6.3 presents the speedup of the hybrid execution as compared to CPU-only execution. BRB attained the best gains for using hybrid execution for 7/10 images. On some cases, the attained hybrid vs. CPU-only performance was significantly better than the expected theoretical. For instance, image 86-8669 expected speedup for hybrid execution should be 2.48× (1.48+1(CPU)). These cases, more common for BRB, can be explained by more background removal and improved load imbalance due to more partitions being generated. Nevertheless, all algorithms performed well, with the worst

Table 6.3: Speedups of the GPU-only and hybrid CPU-GPU execution vs. CPU-only (48 cores) on a single node. A single V100 GPU was used for the GPU-only and Hybrid cases.

Image	GPU-only vs. CPU	Hybrid vs. CPU		
		KDT	ECB	BRB
86-8668	1.51	2.40	2.49	2,50
86-8669	1.48	2.42	2.44	3.06
B6-A0RG	1.42	2.48	2.31	1.88
B6-A0X1	1.54	2.44	2.40	2.48
BR-8285	1.41	2.93	2.01	2.03
BR-8296	1.52	2.42	2.32	2.73
BR-8361	1.58	2.48	2.43	2.64
BR-8682	1.63	2.55	2.54	2.71
FA-8693	1.58	2.33	2.26	2.62
LL-A8F5	1.55	2.43	2.48	2.28

hybrid execution efficiency (hybrid_speedup/(1+cpu_only_speedup)) around 84% and median of 96%.

When scaling the number of GPUs used (all results on the tables of Figure 6.6), both CAPD algorithms significantly outperformed KDT, with min/average/max speedups of 0.88/1.11/1.55× and 1.02/1.39/2.37× for ECB and BRB respectively. These gains are lower than the ones with GPU-only execution since there is a new source of error to the partitioning process related to using hybrid resources. This is also shown with lower balance efficiency values for all algorithm when compared with GPU-only execution, with BRB being the most affected. This still did not resulted into KDT outperforming BRB given the latter execution cost reduction due to background removal and yet reasonable workload balance efficiency. At last, when comparing balance efficiencies of KDT and ECB the only cases on which speedups are not proportional to balance efficiency are for image BR-8285. Since this image is sparse, the gains can be attributed mostly to background removal. It is also worth noting that, as expected, all makespan values for hybrid execution are better than their GPU-only counterparts, for all images and algorithms.

The results for hybrid execution of images BR-8682 and B6-A0X1 (see Figures 6.7 and 6.8) are similar to the ones for GPU-only execution, with slight worse balance and scaling efficiencies due to the use of hybrid partitioning.

Figure 6.6: Full data for hybrid execution experiments on a single CPU-GPU compute node. For each image, the values are related to the number of compute GPU devices used, from 1-4 GPUs.

Speedups for ECB vs. KDT

Image	1	2	3	4
86-8668	1.03	1.13	1.01	1.05
86-8669	0.94	1.07	0.99	1.13
B6-A0RG	0.95	1.11	1.15	1.06
B6-A0X1	1.27	1.33	1.33	1.55
BR-8285	1.35	1.40	1.40	1.44
BR-8296	0.97	1.00	0.89	1.12
BR-8361	0.99	1.02	1.01	1.07
BR-8682	1.03	1.09	1.04	1.12
FA-8693	0.94	1.01	1.04	1.16
LL-A8F5	1.05	1.06	1.09	1.15

Speedups for BRB vs. KDT

Image	1	2	3	4
86-8668	1.17	1.24	1.28	1.49
86-8669	1.31	1.41	1.26	1.57
B6-A0RG	1.08	1.28	1.32	1.24
B6-A0X1	1.32	1.42	1.49	1.69
BR-8285	2.01	2.13	2.20	2.37
BR-8296	1.26	1.23	1.25	1.41
BR-8361	1.11	1.15	1.15	1.27
BR-8682	1.17	1.30	1.29	1.21
FA-8693	1.49	1.62	1.54	1.76
LL-A8F5	1.03	1.06	1.05	1.08

Speedups for BRB vs. ECB

Image	1	2	3	4
86-8668	1.13	1.10	1.27	1.42
86-8669	1.39	1.32	1.27	1.39
B6-A0RG	1.14	1.15	1.15	1.17
B6-A0X1	1.04	1.07	1.12	1.09
BR-8285	1.49	1.52	1.58	1.65
BR-8296	1.30	1.23	1.41	1.26
BR-8361	1.11	1.12	1.13	1.18
BR-8682	1.13	1.19	1.25	1.08
FA-8693	1.59	1.61	1.48	1.52
LL-A8F5	0.98	1.00	0.96	0.94

Scaling Efficiency for ECB

Image	2	3	4
86-8668	84%	71%	66%
86-8669	84%	78%	68%
B6-A0RG	84%	78%	71%
B6-A0X1	78%	71%	69%
BR-8285	77%	70%	60%
BR-8296	76%	65%	60%
BR-8361	78%	70%	64%
BR-8682	80%	73%	69%
FA-8693	83%	79%	72%
LL-A8F5	80%	73%	69%

Scaling Efficiency for KDT

Image	2	3	4
86-8668	77%	73%	65%
86-8669	73%	73%	57%
B6-A0RG	72%	64%	64%
B6-A0X1	74%	67%	56%
BR-8285	74%	67%	57%
BR-8296	73%	70%	52%
BR-8361	76%	68%	59%
BR-8682	76%	73%	63%
FA-8693	76%	71%	58%
LL-A8F5	80%	70%	63%

Scaling Efficiency for BRB

Image	2	3	4
86-8668	81%	79%	83%
86-8669	79%	71%	68%
B6-A0RG	86%	79%	74%
B6-A0X1	80%	76%	72%
BR-8285	78%	74%	67%
BR-8296	72%	70%	58%
BR-8361	79%	71%	68%
BR-8682	84%	81%	66%
FA-8693	83%	74%	69%
LL-A8F5	82%	72%	66%

Balance Efficiency for ECB

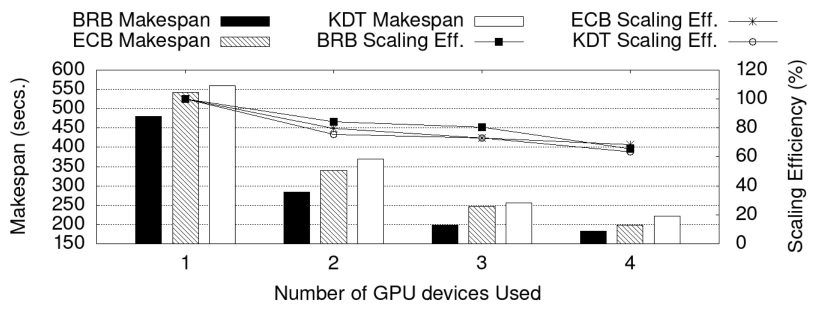
Image	2	3	4
86-8668	85%	93%	90%
86-8669	99%	91%	89%
B6-A0RG	85%	88%	93%
B6-A0X1	82%	98%	91%
BR-8285	84%	94%	76%
BR-8296	83%	97%	95%
BR-8361	97%	90%	89%
BR-8682	99%	87%	92%
FA-8693	100%	89%	90%
LL-A8F5	99%	88%	90%

Balance Efficiency for KDT

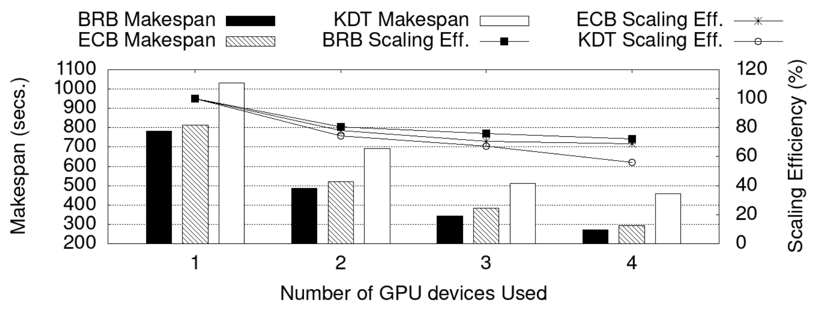
Image	2	3	4
86-8668	97%	90%	81%
86-8669	93%	88%	79%
B6-A0RG	92%	89%	80%
B6-A0X1	81%	97%	77%
BR-8285	99%	91%	91%
BR-8296	83%	88%	79%
BR-8361	83%	87%	86%
BR-8682	81%	97%	86%
FA-8693	79%	85%	66%
LL-A8F5	84%	88%	87%

Balance Efficiency for BRB

Image	2	3	4
86-8668	85%	82%	87%
86-8669	82%	83%	88%
B6-A0RG	83%	85%	95%
B6-A0X1	97%	93%	94%
BR-8285	96%	87%	84%
BR-8296	91%	93%	85%
BR-8361	85%	90%	90%
BR-8682	83%	87%	81%
FA-8693	85%	87%	88%
LL-A8F5	82%	86%	91%

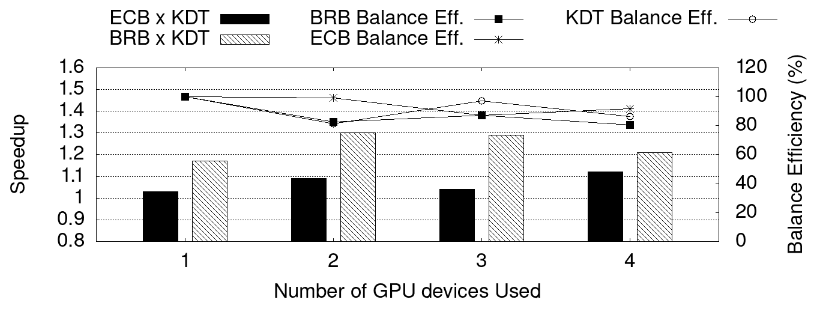


(a) Results for dense image BR-8682.

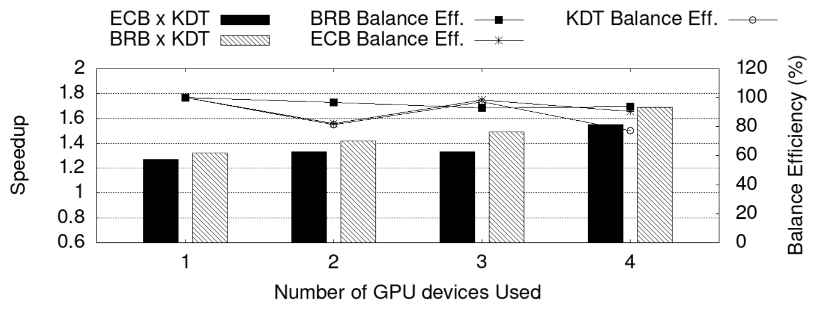


(b) Results for sparse image B6-A0X1.

Figure 6.7: Application makespan of images BR-8682 and B6-A0X1 with related scaling efficiency for 1 to 4 GPUs used for hybrid execution.



(a) Results for dense image BR-8682.



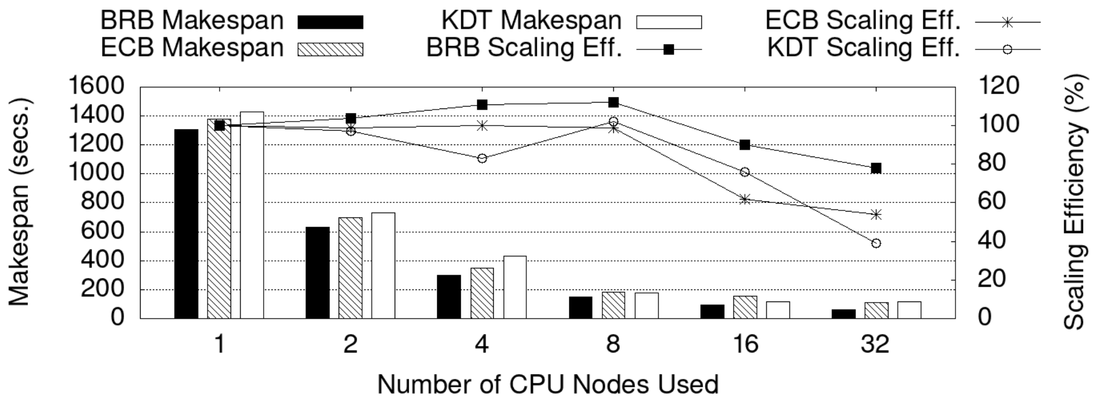
(b) Results for sparse image B6-A0X1.

Figure 6.8: Application makespan of images BR-8682 and B6-A0X1 with related scaling efficiency for 1 to 4 GPUs used for hybrid execution.

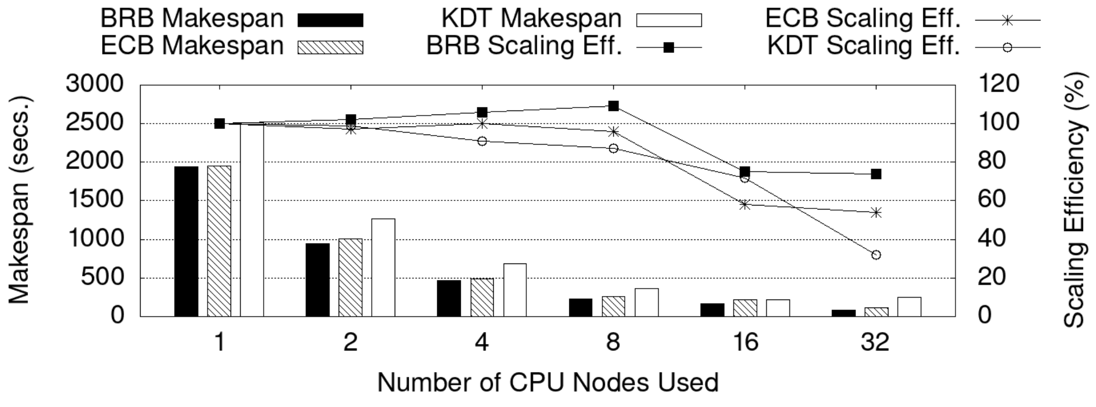
6.2 Distributed Memory Execution

This section evaluates the system in for distributed memory execution using up to 32 CPU-only compute nodes, for a total of 896 CPU-cores. Similarly to the previous sections, the images were processed while using different partitioning algorithms and a ghost zone of 100 pixels and 4 partitions per node for all tested algorithms. The performance of the application with different strategies and images is presented in full on the tables of Figure 6.11. As shown, both CADP algorithms once again performed better than the baseline algorithm in most of the cases, with BRB improving on the performance of both ECB and KDT. Overall, ECB was faster than KDT on 49 out of 60 test points (10 images and 6 distributed settings), with speedups in the range of 0.9× to 2.4×. BRB managed even better, with speedups between 1.02× and 4.5× when compared to KDT, and 0.91× and 3.03× when comparing to ECB.

Similarly with the previous subsections, a couple of selected WSIs are emphasized

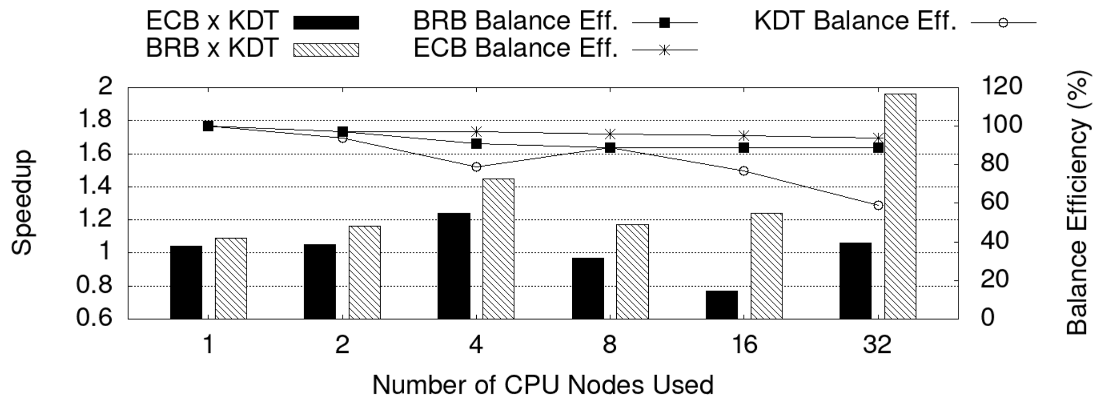


(a) Results for dense image BR-8682.

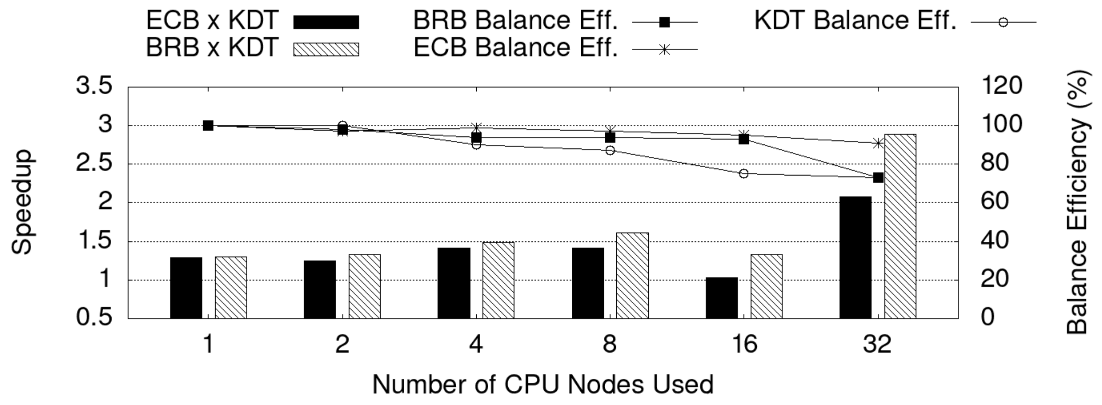


(b) Results for sparse image B6-A0X1.

Figure 6.9: Application makespan of images BR-8682 and B6-A0X1 with related scaling efficiency for 1 to 32 nodes used. Image BR-8682 represents the cases of low background images while B6-A0X1 represents sparse images.



(a) Results for dense image BR-8682.



(b) Results for sparse image B6-A0X1.

Figure 6.10: Application makespan of images BR-8682 and B6-A0X1 with related scaling efficiency for 1 to 32 nodes used. Image BR-8682 represents the cases of low background images while B6-A0X1 represents sparse images.

here to better understand some of the performance results. On the first type of chart of Figures 6.9a and 6.9b it is visible that ECB was significantly better than KDT, with BRB improving on both for every number of used nodes. Regarding scaling efficiency, every algorithm scaled relatively well up to 16 nodes, with lower values at 32 nodes. The case of BRB is a bit different with over 100% efficiency at some cases. This is due to the fact that more nodes means more partitions are generated. With larger numbers of partitions BRB is able to remove ever more background, reducing the overall execution cost to the extent of showing these scaling efficiency values. For KDT, there is one outlier for image BR-8285, which manages to achieve scaling efficiencies of over 100%. This particular case can be explained by the irregular nature of IWPP applications. For this image, the overall cost of executing the 4 partitions with a single node was particularly high (higher number of propagation iterations), resulting in skewed values. Another aspect to observe on the balance efficiency results of Figure 6.11 is that on most cases (36/50) ECB had better results, as discussed before on Section 5.2.1. However, the trade-off of larger

Figure 6.11: Full data for distributed execution experiments on CPU-only compute nodes. For each image, the values are related to the number of compute nodes used, from 1-32 nodes.

Speedups for ECB vs. KDT

Image	1	2	4	8	16	32
86-8668	0.99	1.17	1.08	1.26	1.24	0.94
86-8669	0.93	1.02	1.28	1.67	0.97	2.43
B6-A0RG	1.02	1.03	1.26	1.50	1.72	1.19
B6-A0X1	1.29	1.26	1.41	1.42	1.57	2.21
BR-8285	1.97	1.86	1.57	1.19	1.62	1.27
BR-8296	1.00	1.18	1.34	0.97	1.08	1.02
BR-8361	1.01	1.03	1.14	0.98	0.98	0.90
BR-8682	1.04	1.06	1.25	1.01	1.26	1.43
FA-8693	0.96	1.16	1.30	1.53	0.98	0.98
LL-A8F5	1.03	1.12	1.10	1.22	1.26	1.30

Speedups for BRB vs. KDT

Image	1	2	4	8	16	32
86-8668	1.12	1.39	1.29	1.80	1.84	1.62
86-8669	1.03	1.31	1.82	2.31	1.58	4.52
B6-A0RG	1.42	1.90	2.33	2.74	2.96	1.68
B6-A0X1	1.30	1.34	1.50	1.63	1.84	3.00
BR-8285	2.90	2.89	3.32	2.72	3.69	3.86
BR-8296	1.11	1.27	1.60	1.38	1.78	2.19
BR-8361	1.04	1.05	1.21	1.22	1.37	1.30
BR-8682	1.09	1.17	1.47	1.21	1.63	2.19
FA-8693	1.32	1.45	1.72	2.26	1.52	1.66
LL-A8F5	1.09	1.23	1.31	1.44	1.16	1.28

Speedups for BRB vs. ECB

Image	1	2	4	8	16	32
86-8668	1.13	1.19	1.19	1.43	1.48	1.73
86-8669	1.10	1.29	1.43	1.38	1.63	1.86
B6-A0RG	1.39	1.85	1.84	1.82	1.72	1.41
B6-A0X1	1.00	1.06	1.06	1.14	1.18	1.36
BR-8285	1.48	1.56	2.11	2.29	2.27	3.04
BR-8296	1.10	1.08	1.19	1.42	1.65	2.14
BR-8361	1.02	1.02	1.06	1.24	1.40	1.44
BR-8682	1.06	1.11	1.18	1.19	1.29	1.53
FA-8693	1.36	1.25	1.33	1.48	1.56	1.69
LL-A8F5	1.06	1.10	1.18	1.19	0.92	0.98

Scaling Efficiency for ECB

Image	2	4	8	16	32
86-8668	105%	102%	95%	79%	65%
86-8669	101%	97%	101%	75%	54%
B6-A0RG	97%	97%	98%	91%	63%
B6-A0X1	97%	100%	96%	88%	54%
BR-8285	102%	83%	75%	54%	38%
BR-8296	99%	89%	74%	61%	42%
BR-8361	100%	96%	89%	69%	44%
BR-8682	99%	100%	99%	92%	54%
FA-8693	108%	99%	93%	80%	61%
LL-A8F5	99%	96%	95%	85%	72%

Scaling Efficiency for KDT

Image	2	4	8	16	32
86-8668	89%	94%	75%	63%	69%
86-8669	92%	71%	56%	72%	21%
B6-A0RG	97%	79%	67%	54%	54%
B6-A0X1	99%	91%	87%	73%	32%
BR-8285	108%	103%	124%	65%	59%
BR-8296	85%	67%	76%	57%	41%
BR-8361	98%	85%	92%	72%	49%
BR-8682	97%	83%	102%	76%	39%
FA-8693	90%	74%	59%	79%	60%
LL-A8F5	91%	89%	80%	69%	57%

Scaling Efficiency for BRB

Image	2	4	8	16	32
86-8668	111%	108%	121%	104%	100%
86-8669	118%	126%	126%	110%	91%
B6-A0RG	129%	129%	128%	112%	64%
B6-A0X1	102%	106%	109%	104%	74%
BR-8285	108%	118%	116%	83%	79%
BR-8296	97%	96%	95%	92%	82%
BR-8361	100%	100%	108%	95%	62%
BR-8682	104%	111%	112%	112%	78%
FA-8693	99%	96%	101%	92%	75%
LL-A8F5	103%	107%	106%	74%	67%

Balance Efficiency for ECB

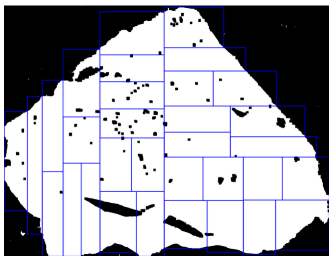
Image	2	4	8	16	32
86-8668	99%	92%	90%	88%	72%
86-8669	95%	93%	89%	78%	72%
B6-A0RG	94%	92%	94%	92%	94%
B6-A0X1	97%	99%	97%	95%	91%
BR-8285	96%	89%	94%	85%	84%
BR-8296	100%	96%	92%	93%	85%
BR-8361	99%	98%	95%	86%	82%
BR-8682	97%	97%	96%	95%	94%
FA-8693	97%	96%	92%	86%	71%
LL-A8F5	99%	91%	93%	90%	88%

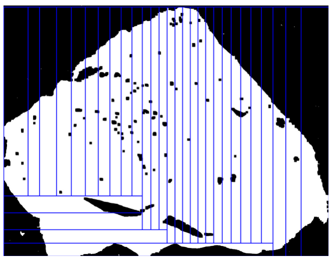
Balance Efficiency for KDT

Image	2	4	8	16	32
86-8668	86%	89%	69%	61%	68%
86-8669	89%	67%	52%	57%	44%
B6-A0RG	94%	77%	65%	55%	58%
B6-A0X1	100%	90%	87%	75%	73%
BR-8285	83%	80%	75%	52%	42%
BR-8296	85%	68%	74%	58%	48%
BR-8361	97%	85%	87%	77%	60%
BR-8682	94%	79%	89%	77%	59%
FA-8693	85%	69%	55%	69%	58%
LL-A8F5	89%	88%	81%	74%	65%

Balance Efficiency for BRB

Image	2	4	8	16	32
86-8668	98%	87%	91%	90%	61%
86-8669	99%	92%	91%	77%	70%
B6-A0RG	100%	99%	96%	90%	81%
B6-A0X1	98%	94%	94%	93%	73%
BR-8285	95%	88%	92%	93%	91%
BR-8296	92%	87%	85%	92%	84%
BR-8361	91%	91%	94%	70%	71%
BR-8682	97%	91%	89%	89%	89%
FA-8693	97%	94%	83%	83%	91%
LL-A8F5	97%	94%	94%	94%	92%


(a) Partitioning with BRB.


(b) Partitioning with ECB.

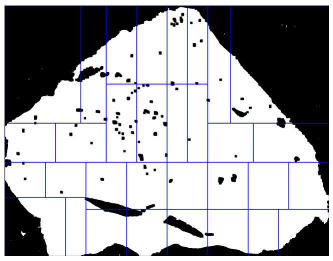
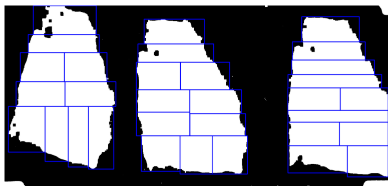

(c) Partitioning with KDT.

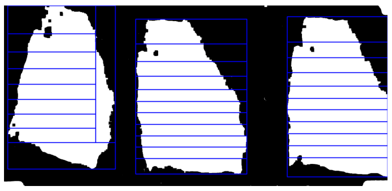
Figure 6.12: Partitioning results for image BR-8682. 32 partitions were generated for the execution with 8 nodes.

amount of fine-grain background removed versus lower balance efficiency still proved to be advantageous to BRB.

On the second type of chart, on Figures 6.10a and 6.10b it becomes easier to see why both ECB and BRB outperform the baseline KDT algorithm. As seen, partitioning with KDT results in less balanced partitions, which in turn hurts the balance efficiency. It is interesting to see that when KDT manages to improve its balance efficiency, its relative performance is also improved, as seen on Figure 6.10a on which ECB and BRB speedups against KDT drop, and on Figure 6.9a with a spike of scaling efficiency.

It is also important to highlight that KDT is naturally prone to perform worse since it is usually executing a larger region of the whole WSI. Figures 6.12 and 6.13 show the partitioning results for the execution of 8 nodes, on which 32 partitions were generated. For Figure 6.12 it is seen that only BRB manages to significantly reduce the amount of background available, resulting in the higher disparity of speedups of BRB and ECB on Figure 6.10a. As such, together with the results of Figure 6.9a, points to the conclusion of how good ECB is at balancing the generated partitions. The partitions of Figure 6.13 show a major limitation of KDT: not being able to work with an arbitrary number of initial dense regions. These scenarios present the best outcomes for both ECB and BRB, which speedups against KDT are generally greater and closer to each other, as seen on Figure 6.10b. As expected, BRB manager to improve significantly on ECB due to its hierarchical partitioning, generating more *square-shaped* partitions and removing more


(a) Partitioning with BRB.


(b) Partitioning with ECB.

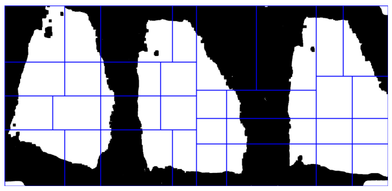
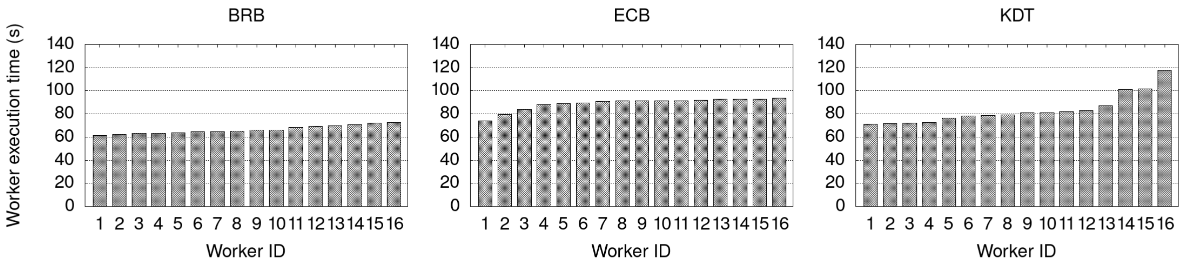
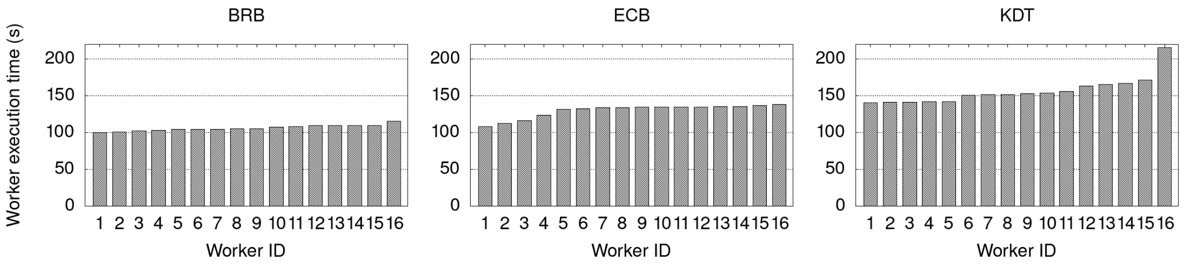

(c) Partitioning with KDT.

Figure 6.13: Partitioning results for image B6-A0X1. 32 partitions were generated for the execution with 8 nodes.



(a) Workload distribution for image BR-8682 on 16 CPU nodes.



(b) Workload distribution for image B6-A0X1 on 16 CPU nodes.

Figure 6.14: Workers workload for 16 CPU nodes configuration with BRB, ECB and KDT respectively, images BR-8682 and B6-A0X1. Execution times are sorted for better visualization.

background with minor impact on the balance efficiency.

Finally, Figure 6.14 shows the workload distribution for 16 compute nodes with the BR-8682 and B6-A0X1 images, better illustrating the actual impact of the CADP approach on workload imbalance. It is important to notice that although BRB seems to outperform ECB regarding balance efficiency this is not the case (95% vs. 89% on BR-8682, and 95% vs. 93% on B6-A0X1 for ECB and BRB respectively). It is also noteworthy that at least 8 CPU-only nodes are required to outperform the use of a single GPU node on hybrid execution with 4 GPUs.

6.2.1 Execution Time of the Partitioning Algorithms

Naturally, The input partitioning process has an execution cost which offsets its makespan gains. As such, there is a tradeoff of generating good partitions and how much time is required to generate these partitions should be observed. As shown in the tables of Figure 6.15, KDT achieves the best partitioning times given its simplicity. For the CADP algorithms, ECB is significantly more computationally expensive than BRB. Although ECB and BRB have equivalent algorithmic complexity, when executing the BTS section of the algorithm BRB is should find the expected cost partition earlier on the binary-search process. For instance, in order to find the first partition out of 32 on a perfectly

Figure 6.15: Execution times in seconds for the partitioning algorithms on CPU-only compute nodes. For each image, the values are related to the number of compute nodes used, from 1-32 nodes.

Partitioning times for ECB

Image	1	2	4	8	16	32
86-8668	1.13	1.97	3.26	6.60	12.29	22.91
86-8669	1.45	2.29	4.03	7.74	14.45	27.61
B6-A0RG	0.66	1.10	1.96	3.80	6.42	12.36
B6-A0X1	0.61	0.83	1.30	2.10	3.40	6.86
BR-8285	0.54	0.74	1.16	2.01	3.52	6.86
BR-8296	1.29	1.89	3.25	6.03	11.45	20.77
BR-8361	1.01	1.74	3.14	5.84	10.57	20.36
BR-8682	1.45	2.14	3.88	7.46	14.68	28.58
FA-8693	1.34	2.26	4.17	7.35	14.58	26.69
LL-A8F5	1.64	2.48	4.61	9.12	17.65	34.28

Partitioning times for BRB

Image	1	2	4	8	16	32
86-8668	1.89	2.54	3.11	3.82	4.76	5.63
86-8669	2.41	3.07	3.84	4.65	5.60	6.75
B6-A0RG	0.90	1.35	1.10	2.03	2.37	2.79
B6-A0X1	0.92	1.34	1.76	2.19	2.76	3.30
BR-8285	0.82	0.98	1.11	1.28	1.45	1.59
BR-8296	2.07	2.73	3.36	4.16	5.12	5.98
BR-8361	1.93	2.42	3.04	3.77	4.68	5.68
BR-8682	2.28	2.92	3.65	4.52	5.47	6.68
FA-8693	2.12	2.70	3.38	4.22	5.18	6.07
LL-A8F5	2.45	3.31	4.19	5.05	6.46	7.42

Partitioning times for KDT

Image	1	2	4	8	16	32
86-8668	0.38	0.47	0.53	0.63	0.73	0.86
86-8669	0.52	0.59	0.67	0.77	0.88	1.04
B6-A0RG	0.26	0.30	0.36	0.41	0.47	0.52
B6-A0X1	0.33	0.37	0.44	0.48	0.55	0.60
BR-8285	0.25	0.29	0.39	0.38	0.42	0.47
BR-8296	0.51	0.52	0.65	0.68	0.78	0.86
BR-8361	0.39	0.45	0.51	0.56	0.65	0.73
BR-8682	0.48	0.54	0.61	0.69	0.81	1.00
FA-8693	0.51	0.57	0.67	0.75	0.97	1.03
LL-A8F5	0.54	0.64	0.74	0.84	0.97	1.13

homogeneous image, ECB should require at least 5 cuts ($log(32)$) while BRB could perform a single cut due to its decomposition of 32 into $2 \times 2 \times 2 \times 2 \times 2$ and its hierarchical partitioning. Finally, although the partitioning costs can be expensive for increasing numbers of partitions, these algorithms still achieved positive overall makespan speedups. However, further optimization these algorithms still remains and open problem.

6.3 Impact of Background Removal

It is important to evaluate the speedup impacts of using background removal. This, however, can mainly be done for ECB, since BRB performs fine-grain background removal even after the coarse-grain removal. Alternatively, ECB's background removal/partitioning step is fully executed before the actual balanced partitioning algorithm. This section evaluates this impact.

After running both ECB an BRB without coarse-grain background removal both algorithms worsen their performance. First, the overall speedups of ECB fell from the max/avg/min values of 2.42/1.24/0.90× to 1.43/1.05/0.61×. For ECB, 38% of all speedups are below 1×, compared with 18% when using coarse-grain background removal, shown in the tables of Figure 6.16. When evaluating the balance efficiency, it becomes clear that its worsening relates to some of the performance losses, as shown in tables of Figure 6.17 with the balance efficiency difference and speedups between/of both versions. One source

Figure 6.16: Speedups for ECB vs. KDT. ECB with and without coarse-grain background removal shown.

Original Speedups for ECB vs. KDT

Image	1	2	4	8	16	32
86-8668	0.99	1.17	1.08	1.26	1.24	0.94
86-8669	0.93	1.02	1.28	1.67	0.97	2.43
B6-A0RG	1.02	1.03	1.26	1.50	1.72	1.19
B6-A0X1	1.29	1.26	1.41	1.42	1.57	2.21
BR-8285	1.97	1.86	1.57	1.19	1.62	1.27
BR-8296	1.00	1.18	1.34	0.97	1.08	1.02
BR-8361	1.01	1.03	1.14	0.98	0.98	0.90
BR-8682	1.04	1.06	1.25	1.01	1.26	1.43
FA-8693	0.96	1.16	1.30	1.53	0.98	0.98
LL-A8F5	1.03	1.12	1.10	1.22	1.26	1.30

ECB w/o coarse-grain background removal vs. KDT

Image	1	2	4	8	16	32
86-8668	0.99	1.16	1.03	0.89	1.27	0.83
86-8669	1.02	1.10	1.36	1.12	1.01	0.99
B6-A0RG	0.98	1.05	1.21	0.91	1.39	1.30
B6-A0X1	0.99	0.99	1.09	0.62	1.06	0.84
BR-8285	1.21	1.13	1.18	0.80	0.88	0.86
BR-8296	1.00	1.16	1.39	0.72	1.22	1.43
BR-8361	1.02	1.03	1.14	0.64	0.95	1.06
BR-8682	1.02	1.01	1.19	1.11	1.05	0.87
FA-8693	0.96	1.16	1.29	0.94	0.96	0.95
LL-A8F5	1.00	1.09	1.15	1.02	1.22	1.32

which explains such worsening of balance efficiency is the quality of the partitioning. As seen in Figure 6.18 the partitions generated for images B6-A0X1 and BR-8285 (images with the worst results) by KDT are more reasonable than the ones generated by ECB. This can be explained by (i) ECB low sensitivity to partitions' sizes, which influence on I/O times, and (ii) that ECB generation is not hierarchical, resulting in the long strips shown in Figures 6.18a and 6.18c.

Further, there are still very few cases on which balance efficiency does not correlate with improved performance, i.e., KDT manages to outperform ECB even with worse balance efficiency. As mentioned on Section 5.2, one of the motivations of BRB was to improve on the quality of the partitions as to generate more square-shaped partitions (see Figure 6.18c), since the stripe-shaped partitions could increase the overall cost of processing them due to IWPP. This effect can be seen on Figure 6.19, on which ECB workers execution times are more balanced, but overall higher than the times of KDT. By removing the coarse-grain background removal step the losses in performance were such that improving load balance was not enough for ECB to still attain gains when compared to KDT.

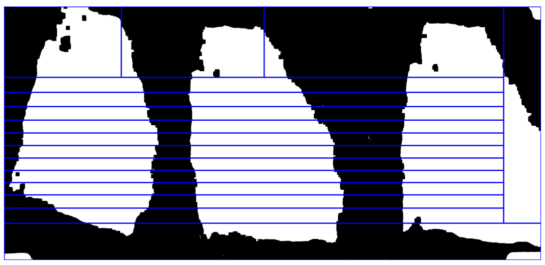
Figure 6.17: Comparative data of ECB with vs. without coarse-grain background removal.

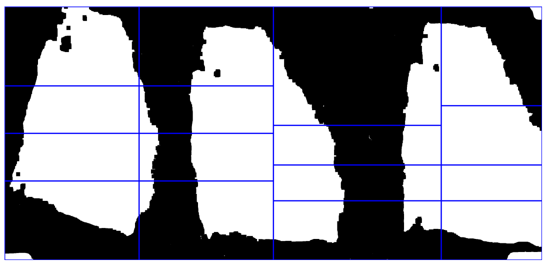
ECB Balance eff. difference with-w/o background removal

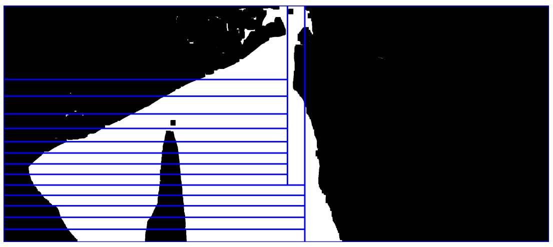
Image	2	4	8	16	32
86-8668	-0.46	2.29	-17.73	1.31	6.13
86-8669	4.84	1.54	-17.52	4.61	-0.09
B6-A0RG	3.52	0.76	-23.94	-9.48	-9.99
B6-A0X1	2.34	-3.23	-33.05	-7.33	-2.70
BR-8285	-8.47	0.76	-8.97	-7.71	-12.91
BR-8296	-0.01	-1.65	-12.01	-4.12	1.47
BR-8361	0.30	-0.22	-24.49	-0.12	0.17
BR-8682	-0.71	-2.14	-20.64	-2.74	-22.02
FA-8693	0.05	-1.32	-27.58	-0.28	8.96
LL-A8F5	0.04	5.34	-5.93	-0.90	0.47

Speedup of ECB with vs. w/o background removal

Image	1	2	4	8	16	32
86-8668	1.01	1.02	1.04	1.39	0.95	1.08
86-8669	0.92	0.92	0.94	1.48	0.95	0.99
B6-A0RG	1.04	0.97	1.04	1.63	1.21	0.93
B6-A0X1	1.31	1.26	1.29	2.29	1.49	1.18
BR-8285	1.63	1.64	1.34	1.49	1.55	1.47
BR-8296	1.00	1.01	0.97	1.34	0.88	0.79
BR-8361	1.00	1.00	1.00	1.50	1.00	0.87
BR-8682	1.02	1.04	1.04	1.45	1.06	1.05
FA-8693	1.00	1.00	1.01	1.61	1.00	1.00
LL-A8F5	1.03	1.03	1.00	1.27	1.08	1.01


(a) Partitioning of B6-A0X1 with ECB.


(b) Partitioning of B6-A0X1 with KDT.


(c) Partitioning of BR-8285 with ECB.

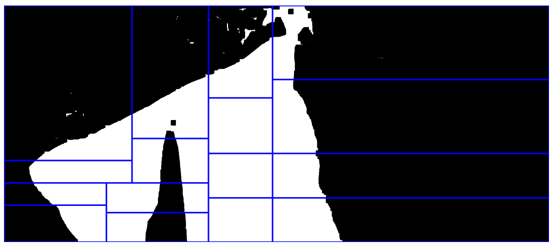

(d) Partitioning of BR-8285 with KDT.

Figure 6.18: Partitioning results for images B6-A0X1 and BR-8285 using algorithms ECB and KDT. 16 partitions were generated for the execution with 4 nodes.

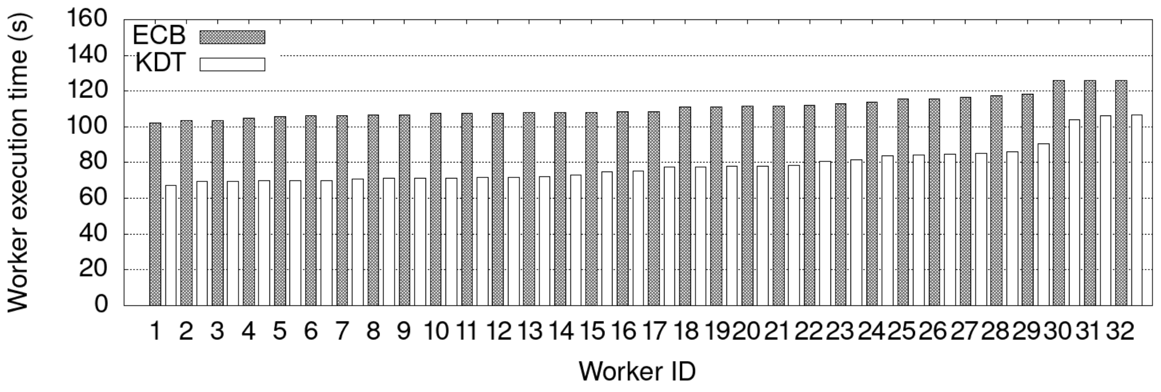
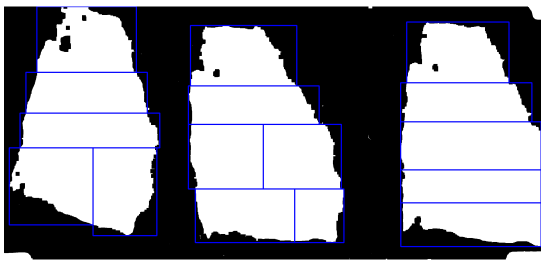


Figure 6.19: Workers workload for 32 CPU nodes configuration with ECB and KDT for image B6-A0X1. Execution times are sorted for better visualization.

Table 6.4: Speedups of BRB with vs. without coarse-grain background removal.

Image	1	2	4	8	16	32
86-8668	1.00	1.01	0.99	1.36	0.90	1.05
86-8669	0.95	0.97	0.97	1.04	0.95	1.01
B6-A0RG	1.38	1.84	1.81	1.76	1.46	1.00
B6-A0X1	1.23	1.27	1.33	1.55	1.58	1.12
BR-8285	1.19	1.30	1.45	1.64	0.92	1.15
BR-8296	1.01	1.02	1.01	1.22	0.95	1.09
BR-8361	1.00	0.92	0.96	1.01	0.99	1.00
BR-8682	1.00	1.01	0.99	1.38	1.02	0.90
FA-8693	1.00	1.06	0.91	1.31	0.90	0.98
LL-A8F5	1.00	1.10	1.09	1.13	0.96	1.20


(a) Partitioning of B6-A0X1 using BRB with coarse-grain background removal.

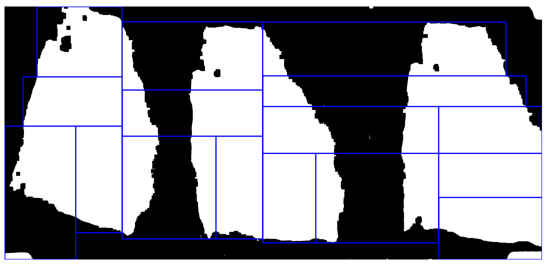

(b) Partitioning of B6-A0X1 using BRB without coarse-grain background removal.

Figure 6.20: Partitioning results for image B6-A0X1 using algorithm BRB with and without coarse-grain background removal. 16 partitions were generated for the execution with 4 nodes.

At last, BRB performance is only slightly affected by the use of coarse-grain background removal, as seen on Table 6.4 which shows the speedups of using coarse-grain background removal, ranging from 0.89-1.84×. Three images showed the highest discrepancy of performance: B6-A0RG, B6-A0X1 and BR-8285, all being sparse images. As seen on Figure 6.20b, although BRB is able to remove a significant amount of background for Image B6-A0X1, it was not able to generate partitions which not include the background between dense regions. This happened due to the number of dense regions being a multiple of 3 while the number of partitions is a multiple of 2.

6.4 Impact of Inaccurate Cost-Function Estimates

In this section, we evaluate the impact of errors or inaccuracy of the cost estimates provided to CADP to its performance. To evaluate this effect, we have intentionally inserted errors to the estimated cost in a systematic manner. Our strategy consisted on defining an $X\%$ error (positive or negative), in which a value between $-X$ and $+X$ (at random) is added to the calculated value returned by the cost-function. The $X\%$ error is then kept fixed within each experiment, and we varied this percentage and executed multiple experiments to evaluate the systems performance. The experiments were executed using 16 nodes and ECB partitioning algorithm in order to avoid any performance impacts from fine-grain background removal.

The performance of the application for CADP normalized by the case in which there are no errors inserted in the cost-function are shown in Figure 6.21. As may be observed, the usage of an error skewed cost-function for CADP only slightly impacts its performance until 15% of inserted error. However, there is a more significant performance degradation for error in the range of 30%. Higher error rates at the cost-function level may lead the partitioning algorithms to lose the capacity to distinguish sparse from dense regions,

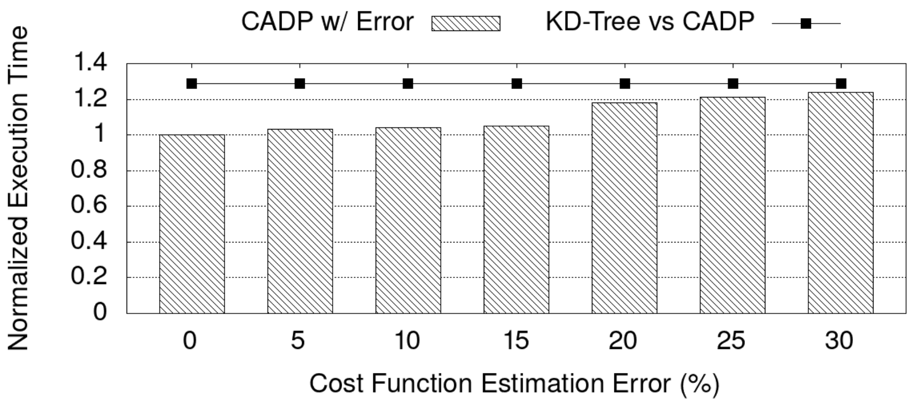


Figure 6.21: Normalized execution time of CADP using an error-prone cost-function against regular CADP. KD-Tree performance added as a baseline.

required by the CADP. It is worth noting that KDT attained similar performance with CADP for high added error values, over 30%.

Chapter 7

Conclusion

In this work, the use of distributed memory hybrid machines to efficiently execute histopathology image analysis applications was studied. A runtime system that enables easy and efficient application deployment on hybrid computing systems, while achieving high performance and reasonable scalability was proposed. This runtime system enabled the execution of general workflow implementations for both CPU and GPU using a high-level DSL as an embedded solution to ease the implementation process for domain experts.

Histopathology image processing applications with irregular execution cost were the focus of the solution proposed by this work. It was shown that large scale experimentation using Whole-Slide Tissue Images (WSI) is a compute intensive task, for which High-Performance Computing (HPC) solutions should be employed. Two main issues arise from using HPC: (i) the ease of implementing a local analysis algorithm to large scale distributed and sometimes hybrid compute environments, and (ii) the efficient distribution of work among such distributed resources.

A straightforward runtime system was implemented, based on the Region Templates Framework (RT), which allowed the execution of workflows on partitioned WSIs. Also, this new system had the Halide DSL embedded as a way to ease the implementation process for such workflows while allowing simple yet efficient implementations for both CPU and GPU devices with the same code. Finally, many spatial partitioning algorithms were developed to enable efficient distributed execution of these WSI workflows.

The Cost-Aware Data Partitioning (CADP), the solution used for smart partitioning of WSIs, is comprised of two main steps. First, the dense regions of the image are segmented, thus reducing the overall execution cost by not executing the workflow on background regions. Then, the dense regions are partitioned for distributed execution. This novel solution took into consideration the expected cost of a data/image partition when splitting it for parallel processing.

Initially, the Expect-Cost Bisection (ECB) CADP algorithm was developed. While

ECB already showed significant gains compared to the KD-Tree (KDT) partitioning algorithm, used as a baseline for comparison, it also showed some limitations. Although ECB managed to produce load-balanced partitions, as the number of generated partitions increased, the quality of the partitions decreased. These partitions would become evermore stripe-shaped. Given the use of an Irregular Wavefront Propagation Pattern algorithm (IWPP) by the motivating application, such stripe-shaped partitions would be more expensive to execute than more square-shape partitions with the same area. This behavior was later seen experimentally. Also, a study on further removing background from the dense regions was done to assess how beneficial could the further removal of them be. This study on a large set of WSIs showed that at least 16% of background area could be removed from over half of all evaluated images.

With the limitations of ECB and a new goal of removing background even more a new CADP algorithm was implemented. The Background Removal Bisection (BRB) algorithm used a hierarchical partitioning method to avoid generating stripe-shaped partitions. Further each generated partition would undergo a fine-grain background removal step to reduce even further the overall WSI execution cost. Both of these changes resulted in an improved BRB performance when compared to ECB, while also reducing the partitioning time.

The proposed CADP algorithm were evaluated experimentally on a distributed environment of up to 32 CPU-only compute nodes and 4 GPUs on a single hybrid node. At first, the use of Halide as a DSL for implementing the workflows was evaluated, showing reasonable scaling efficiency (at least 76% for 28 CPU cores) and slight overall speedups. These gains can be explained by data access patterns, which naturally arise on Halide implementations, improving spatial data locality. Regarding the use of CADP, it has shown significant improvements on the baseline KDT approach. When using the more sophisticated BRB algorithm, speedups of up to 4.52× were achieved. The GPU-only and hybrid executions have shown good scaling efficiency and balance efficiency, with the hybrid setting showing slight worse performance. Further, CADP was validated for a large scale execution environment of up to 32 compute nodes, with a total of 896 CPU cores. Compared to the hybrid execution results, at least 8 compute nodes would be necessary to outperform a single hybrid node with 4 GPUs. Also, the large scale distributed study better showed that CADP improved on KDT performance by (i) reducing workload imbalance and (ii) smartly removing background. Although ECB managed to perform best regarding balance efficiency, BRB managed to improve upon it by removing even more background at the cost of some of the balance efficiency. It was also shown that BRB improved on the problem of stripe-shaped partitions, which was shown to significantly impact overall workflow performance. At last, it was shown CADP's robustness to errors added to the cost-function, an important section of the overall solution. CADP was shown to still attain enough gains, even with an added random error of up to 30% to the cost estimations.

Given the time complexity of CADP of $\mathcal{O}(t^2 + t\ n\ log_2(n))$, with t as the number of expected partitions and n as the number of pixels in the input image, attempts were made to improve its partitioning cost. A caching system on which the full image was partitioned into fixed-sized tiles was implemented. Each partition generated by CADP would have its cost as the sum of the cost of tiles inside it. This solution, however, was not able to improve the CADP performance. In order to return accurate cost values, the number of tiles would need to be loo large, resulting in a higher cost to aggregate all of the ones inside a partition. Further, experimentally, the tiles' cost reuse rate was low. As such, on this trade-off space between less tiles/quicker partitioning/worse quality and more tiles/slower partitioning/better quality, a point on which CADP performance was improved and its partitioning quality was still acceptable was not found.

Finally, there was an attempt at finding an improved cost-function. Since the cost-function is executed multiple times throughout the CADP execution, it cannot be an expensive function. Thus, the LightGBM framework was used to generate regression models. As input, a set of over 20,000 partitions of different sizes were generated from 10 WSIs. Well known morphological features were used for training, such as: total area, dense area, solidity, relative centroid, sum of perimeters, circularity, euler number, convexity, and convex hull perimeter and area. Some feature selection algorithms were used, such as: boruta, recursive feature elimination and addition. Also, a leave-one-image-out scheme was used, as a regular train-test-validation data splitting. Finally, the input partitions were balanced according to their sizes and execution costs. However, the best model generated was still not able to outperform the simple threshold cost-function already in use.

With the above considerations, there are still more directions to explore from the current state of this work. The cost of partitioning, although having the CADP somewhat optimized, was not fully addressed. Since the more complex task of performing cost-function calls reuse though the combination of other calls did not perform well, perhaps the calls themselves can be optimized. Currently, the cost-function is implemented on Halide, and thus is executed in parallel using all CPU cores. It could then be moved to the GPU for even better execution times. However, a study on the overheads of running just the cost-function on a GPU should be done. The whole partitioning algorithm can also be moved to the GPU, which could solve some of these overheads. A new implementation a more parallel version of CADP can also be pursued, on which more than one partition is generated at a time.

It is believed that the main reason that the regression models for calculating the cost-function failed was that the chosen morphological features were not representative enough. In theory, the regular computing section of the workflow could be estimated relatively well since it only depends on the sizes of the inputs. This is not so simple for the irregular computing section, the IWPP. Although some correlation to the ratio of height/width exists (shown experimentally on Section 6.3), accurate cost estimation is still an open problem. Given the inefficacy of more classical morphological features, using generative deep learning solutions. Though them, a new set of non-trivial features that better relate to the execution times could be found. However, by using such neural networks there is the issue of input sizes, for which are fixed. Thus, a generative method would require a solution for managing the execution of irregular sized partitions.

At last, this work was developed with irregular cost applications as the main motivating domain. More specifically, applications which use classical image processing workflows. Although there are plenty of different applications with these features, a study on whether other applications could benefit from the solutions presented on this work should be done. For instance, among the many applications of histopathology, Content-Based Image Retrieval (CBIR) is a great candidate for such. CBIR applications may query partitions of different sizes, thus adding a cost-awareness aspect to be observed. Another more present class of applications on histopathology regards machine learning and deep and/or convolutions neural networks. Although the computing patterns of such applications are more regular, a study on the possible gains of using the solutions here proposed would also be interesting.

References

[1] R. Duncan, "A survey of parallel computer architectures," *Computer*, vol. 23, no. 2, pp. 5–16, 1990. xii, 7, 8, 9

[2] NVIDIA, "Cuda refresher: The cuda programming model," 2021. xii, 13

[3] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, *et al.*, "A taxonomy of task-based parallel programming technologies for high-performance computing," *The Journal of Supercomputing*, vol. 74, no. 4, pp. 1422–1434, 2018. xvi, 34

[4] M. Peikari, S. Salama, S. Nofech-Mozes, and A. L. Martel, "A cluster-then-label semi-supervised learning approach for pathology image classification," *Scientific reports*, vol. 8, no. 1, pp. 1–13, 2018. 1

[5] L. A. Cooper, D. A. Gutman, C. Chisolm, C. Appin, J. Kong, Y. Rong, T. Kurc, E. G. V. Meir, J. H. Saltz, C. S. Moreno, and D. J. Brat, "The Tumor Microenvironment Strongly Impacts Master Transcriptional Regulators and Gene Expression Class of Glioblastoma," *The American Journal of Pathology*, vol. 180, no. 5, pp. 2108 – 2119, 2012. 1, 27, 28

[6] M. Graziani, I. Eggel, F. Deligand, M. Bobák, V. Andrearczyk, and H. Mueller, "Breast histopathology with high-performance computing and deep learning.," *Comput. Informatics*, vol. 39, no. 4, pp. 780–807, 2020. 1, 2, 25, 26, 35

[7] J. Kong, L. A. D. Cooper, F. Wang, J. Gao, G. Teodoro, T. Mikkelsen, M. J. Schniederjan, C. S. Moreno, J. H. Saltz, and D. J. Brat, "Machine-Based Morphologic Analysis of Glioblastoma Using Whole-Slide Pathology Images Uncovers Clinically Relevant Molecular Correlates," *PLoS ONE*, 2013. 1, 2, 4, 26, 27, 28

[8] K.-H. Yu, C. Zhang, G. J. Berry, R. B. Altman, C. Ré, D. L. Rubin, and M. Snyder, "Predicting non-small cell lung cancer prognosis by fully automated microscopic pathology image features," *Nature communications*, vol. 7, pp. 12474; 12474–12474, 08 2016. 2

[9] G. Teodoro, T. M. Kurç, L. F. R. Taveira, A. C. M. A. Melo, and Y. Gao, "Algorithm sensitivity analysis and parameter tuning for tissue image segmentation pipelines," *Bioinform.*, vol. 33, no. 7, pp. 1064–1072, 2017. 2, 4, 27

[10] J. Gomes, W. Barreiros Jr, T. Kurc, A. C. Melo, J. Kong, J. H. Saltz, and G. Teodoro, "Sensitivity analysis in digital pathology: Handling large number of parameters with compute expensive workflows," *Computers in biology and medicine*, vol. 108, pp. 371–381, 2019. 2, 5

[11] X. Guo, F. Wang, G. Teodoro, A. B. Farris, and J. Kong, "Liver steatosis segmentation with deep learning methods," in *2019 IEEE 16th International Symposium on Biomedical Imaging (ISBI 2019)*, pp. 24–27, IEEE, 2019. 2

[12] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux, *et al.*, "Understanding gpu errors on large-scale hpc systems and the implications for system design and operation," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 331–342, IEEE, 2015. 2

[13] Z. Li, X. Zhang, H. Müller, and S. Zhang, "Large-scale retrieval for medical image analytics: A comprehensive review," *Medical image analysis*, vol. 43, pp. 66–84, 2018. 2, 25, 26

[14] S. Dash, B. Hernández, A. Tsaris, F. T. Alamudun, H.-J. Yoon, and F. Wang, "A scalable pipeline for gigapixel whole slide imaging analysis on leadership class hpc systems," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1266–1274, IEEE, 2022. 2, 26, 35

[15] A. Chowdhury, H. Kassem, N. Padoy, R. Umeton, and A. Karargyris, "A review of medical federated learning: Applications in oncology and cancer research," in *Brainlesion: Glioma, Multiple Sclerosis, Stroke and Traumatic Brain Injuries: 7th International Workshop, BrainLes 2021, Held in Conjunction with MICCAI 2021, Virtual Event, September 27, 2021, Revised Selected Papers, Part I*, pp. 3–24, Springer, 2022. 2, 26

[16] S. Mitra, N. Das, S. Dey, S. Chakraborty, M. Nasipuri, and M. K. Naskar, "Cytology image analysis techniques toward automation: systematically revisited," *ACM Computing Surveys (CSUR)*, vol. 54, no. 3, pp. 1–41, 2021. 2, 26

[17] C. Li, X. Li, X. Li, M. M. Rahaman, X. Li, J. Wu, Y. Yao, and M. Grzegorzek, "A state-of-the-art survey of artificial neural networks for whole-slide image analysis: from popular convolutional neural networks to potential visual transformers," *arXiv preprint arXiv:2104.06243*, 2021. 2, 26

[18] M. J. Berger and S. H. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors," *IEEE Transactions on Computers*, vol. 36, no. 05, pp. 570–580, 1987. 4, 29, 30, 31

[19] H. Samet, *Foundations of multidimensional and metric data structures.* Morgan Kaufmann, 2006. 4, 29, 30

[20] J. Gomes, J. Kong, T. Kurc, A. C. Melo, R. Ferreira, J. H. Saltz, and G. Teodoro, "Building robust pathology image analyses with uncertainty quantification," *Computer Methods and Programs in Biomedicine*, p. 106291, 2021. 4, 26, 27

[21] G. Teodoro, T. M. Kurc, T. Pan, L. A. Cooper, J. Kong, P. Widener, and J. H. Saltz, "Accelerating large scale image analyses on parallel, cpu-gpu equipped systems," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 1093–1104, IEEE, 2012. 4, 26, 27

[22] G. Teodoro, T. Pan, T. M. Kurc, J. Kong, L. A. Cooper, N. Podhorszki, S. Klasky, and J. H. Saltz, "High-throughput analysis of large microscopy image datasets on CPU-GPU cluster platforms," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 103–114, IEEE, 2013. 4, 26, 27, 67, 68

[23] W. Barreiros, G. Teodoro, T. Kurc, J. Kong, A. C. Melo, and J. Saltz, "Parallel and efficient sensitivity analysis of microscopy image segmentation workflows in hybrid systems," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 25–35, IEEE, 2017. 4, 26, 27

[24] W. Barreiros Jr, J. Moreira, T. Kurc, J. Kong, A. C. Melo, J. H. Saltz, and G. Teodoro, "Optimizing parameter sensitivity analysis of large-scale microscopy image analysis workflows with multilevel computation reuse," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 2, p. e5403, 2020. 4, 5, 26

[25] W. Barreiros Jr, A. C. Melo, J. Kong, R. Ferreira, T. M. Kurc, J. H. Saltz, and G. Teodoro, "Efficient microscopy image analysis on cpu-gpu systems with cost-aware irregular data partitioning," *Journal of Parallel and Distributed Computing*, vol. 164, pp. 40–54, 2022. 5, 26, 27

[26] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966. 7

[27] S. K. Raman, V. Pentkovski, and J. Keshava, "Implementing streaming simd extensions on the pentium iii processor," *IEEE micro*, vol. 20, no. 4, pp. 47–57, 2000. 8

[28] C. Lomont, "Introduction to intel advanced vector extensions," *Intel white paper*, vol. 23, 2011. 8

[29] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming.* Pearson Education, 2013. 8, 14, 20

[30] A. Krikelis and C. C. Weems, "Associative processing and processors," *Computer*, vol. 27, no. 11, pp. 12–17, 1994. 8

[31] M. C. Herbordt, Y. Gu, T. VanCourt, J. Model, B. Sukhwani, and M. Chiu, "Computing models for fpga-based accelerators," *Computing in science & engineering*, vol. 10, no. 6, pp. 35–45, 2008. 8

[32] K. Sano, C. Takagi, R. Egawa, K.-i. Suzuki, and T. Nakamura, "A systolic memory architecture for fast codebook design based on mmpdcl algorithm," in *International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004.*, vol. 1, pp. 572–578, IEEE, 2004. 8

[33] Z. Yan and D. V. Sarwate, “New systolic architectures for inversion and division in gf (2/sup m/),” *IEEE Transactions on Computers*, vol. 52, no. 11, pp. 1514–1519, 2003. 8

[34] Srini, “An architectural comparison of dataflow systems,” *Computer*, vol. 19, no. 3, pp. 68–88, 1986. 10

[35] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, “Data-driven and demand-driven computer architecture,” *ACM Computing Surveys (CSUR)*, vol. 14, no. 1, pp. 93–143, 1982. 10

[36] S.-Y. Kung, S.-C. Lo, S.-N. Jean, and J.-N. Hwang, “Wavefront array processors-concept to implementation,” *Computer*, vol. 20, no. 07, pp. 18–33, 1987. 10

[37] T. Blank and J. R. Nickolls, “A grimm collection of mimd fairy tales,” in *The Fourth Symposium on the Frontiers of Massively Parallel Computation*, pp. 448–449, IEEE Computer Society, 1992. 10

[38] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, “A single-program-multiple-data computational model for epex/fortran,” *Parallel Computing*, vol. 7, no. 1, pp. 11–24, 1988. 10

[39] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, “A survey on parallel computing and its applications in data-parallel problems using gpu architectures,” *Communications in Computational Physics*, vol. 15, no. 2, pp. 285–329, 2014. 10, 29, 30, 45

[40] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, “An introduction to the mpi standard,” *Communications of the ACM*, vol. 18, 1995. 11

[41] L. Dagum and R. Menon, “Openmp: an industry standard api for shared-memory programming,” *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998. 12

[42] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001. 12

[43] M. Martineau, S. McIntosh-Smith, and W. Gaudin, “Evaluating openmp 4.0’s effectiveness as a heterogeneous parallel programming model,” in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 338–347, IEEE, 2016. 12

[44] E. Lindholm, M. J. Kilgard, and H. Moreton, “A user-programmable vertex engine,” in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 149–158, 2001. 12

[45] E. S. Larsen and D. McAllister, “Fast matrix multiplies using graphics hardware,” in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pp. 55–55, 2001. 12

[46] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming," *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012. 12

[47] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "Gpu cluster for high performance computing," in *SC'04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, pp. 47–47, IEEE, 2004. 12

[48] J. Krüger and R. Westermann, "Linear algebra operators for gpu implementation of numerical algorithms," in *ACM SIGGRAPH 2005 Courses*, pp. 234–es, 2005. 12

[49] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha, "Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware," in *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pp. 3–3, IEEE, 2005. 12

[50] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: using data parallelism to program gpus for general-purpose uses," *ACM SIGPLAN Notices*, vol. 41, no. 11, pp. 325–335, 2006. 12

[51] K. Group, "Opencl," 2021. 12

[52] NVIDIA, "Cuda," 2021. 12

[53] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, 2019. 14

[54] I. Culjak, D. Abram, T. Pribanic, H. Dzapo, and M. Cifrek, "A brief introduction to opencv," in *2012 proceedings of the 35th international convention MIPRO*, pp. 1725–1730, IEEE, 2012. 14

[55] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000. 15, 67

[56] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013. 15, 34, 36, 38

[57] S. Sioutas, S. Stuijk, T. Basten, H. Corporaal, and L. Somers, "Schedule synthesis for halide pipelines on gpus," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 3, pp. 1–25, 2020. 15, 23, 39

[58] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, pp. 1–11, 2016. 15, 23, 39

[59] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, *et al.*, "Learning to optimize halide with tree search and random programs," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 4, pp. 1–12, 2019. 15, 23, 39

[60] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 193–205, IEEE, 2019. 16, 21, 23, 32, 34

[61] M. Jankowski, "Erosion, dilation and related operators," *Department of Electrical EngineeringUniversity of Southern Maine Portland, Maine, USA*, 2006. 16

[62] P. Hudak and J. H. Fasel, "A gentle introduction to haskell," *ACM Sigplan Notices*, vol. 27, no. 5, pp. 1–52, 1992. 17

[63] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, IEEE, 2004. 19, 20

[64] C. Lattner, "Llvm and clang: Next generation compiler technology," in *The BSD conference*, vol. 5, 2008. 19

[65] A. W. Appel, "Ssa is functional programming," *ACM SIGPLAN Notices*, vol. 33, no. 4, pp. 17–20, 1998. 19

[66] D. Seal, *ARM architecture reference manual.* Pearson Education, 2001. 20

[67] G. Kane, *mips RISC Architecture.* Prentice-Hall, Inc., 1988. 20

[68] R. Johnson and J. Vlissides, "Design patterns," *Elements of Reusable Object-Oriented Software Addison-Wesley, Reading*, 1995. 20

[69] A. Graves, S. Fernández, and J. Schmidhuber, "Bidirectional lstm networks for improved phoneme classification and recognition," in *International conference on artificial neural networks*, pp. 799–804, Springer, 2005. 20

[70] T. Denniston, S. Kamil, and S. Amarasinghe, "Distributed halide," *ACM SIGPLAN Notices*, vol. 51, no. 8, pp. 1–12, 2016. 21, 32, 34, 45

[71] S.-W. Liao, S.-Y. Kuang, C.-L. Kao, and C.-H. Tu, "A halide-based synergistic computing framework for heterogeneous systems," *Journal of Signal Processing Systems*, vol. 91, no. 3, pp. 219–233, 2019. 21, 32, 34

[72] J. Meng and K. Skadron, "A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations," *International Journal of Parallel Programming*, vol. 39, no. 1, pp. 115–142, 2011. 23, 40

[73] M. B. S. Ahmad, J. Ragan-Kelley, A. Cheung, and S. Kamil, "Automatically translating image processing libraries to halide," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 6, pp. 1–13, 2019. 23, 38

[74] G. Teodoro, T. Pan, T. Kurc, J. Kong, L. Cooper, S. Klasky, and J. Saltz, "Region templates: Data representation and management for high-throughput image analysis," *Parallel Computing*, vol. 40, no. 10, pp. 589–610, 2014. 23, 26

[75] H. R. Tizhoosh and L. Pantanowitz, "Artificial intelligence and digital pathology: challenges and opportunities," *Journal of pathology informatics*, vol. 9, no. 1, p. 38, 2018. 25

[76] K. Doi, "Computer-aided diagnosis in medical imaging: historical review, current status and future potential," *Computerized medical imaging and graphics*, vol. 31, no. 4-5, pp. 198–211, 2007. 25

[77] E. Yildirim, S. Duan, and X. Qi, "A distributed deep memory hierarchy system for content-based image retrieval of big whole slide image datasets," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pp. 43–49, IEEE, 2019. 25, 26

[78] M. N. Gurcan, L. E. Boucheron, A. Can, A. Madabhushi, N. M. Rajpoot, and B. Yener, "Histopathological image analysis: A review," *IEEE reviews in biomedical engineering*, vol. 2, pp. 147–171, 2009. 25

[79] J. Ye, Y. Luo, C. Zhu, F. Liu, and Y. Zhang, "Breast cancer image classification on wsi with spatial correlations," in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1219–1223, IEEE, 2019. 25

[80] Y. Peng, Y. Chu, Z. Chen, W. Zhou, S. Wan, Y. Xiao, Y. Zhang, and J. Li, "Combining texture features of whole slide images improves prognostic prediction of recurrence-free survival for cutaneous melanoma patients," *World Journal of Surgical Oncology*, vol. 18, no. 1, pp. 1–8, 2020. 25

[81] Z. Xu, Y. Li, Y. Wang, S. Zhang, Y. Huang, S. Yao, C. Han, X. Pan, Z. Shi, Y. Mao, *et al.*, "A deep learning quantified stroma-immune score to predict survival of patients with stage ii–iii colorectal cancer," *Cancer Cell International*, vol. 21, pp. 1–12, 2021. 25

[82] K. Zhao, Z. Li, S. Yao, Y. Wang, X. Wu, Z. Xu, L. Wu, Y. Huang, C. Liang, and Z. Liu, "Artificial intelligence quantified tumour-stroma ratio is an independent predictor for overall survival in resectable colorectal cancer," *EBioMedicine*, vol. 61, p. 103054, 2020. 25

[83] NIH, "Genomic Data Commons Data Portal," 2020. available at `https://portal.gdc.cancer.gov/`. 26, 64

[84] NCI, "The Cancer Genome Atlas Project," 2020. available at `https://www.cancer.gov/about-nci/organization/ccg/research/structural-genomics/tcga`. 26, 64

[85] S. Ouyang, D. Dong, Y. Xu, and L. Xiao, "Communication optimization strategies for distributed deep neural network training: A survey," *Journal of Parallel and Distributed Computing*, vol. 149, pp. 52–65, 2021. 26

[86] M. Andreux, J. O. du Terrail, C. Beguier, and E. W. Tramel, "Siloed federated learning for multi-centric histopathology datasets," in *Domain Adaptation and Representation Transfer, and Distributed and Collaborative Learning: Second MICCAI Workshop, DART 2020, and First MICCAI Workshop, DCL 2020, Held in Conjunction with MICCAI 2020, Lima, Peru, October 4–8, 2020, Proceedings 2*, pp. 129–139, Springer, 2020. 26

[87] Q. Li, Y. Diao, Q. Chen, and B. He, "Federated learning on non-iid data silos: An experimental study," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pp. 965–978, IEEE, 2022. 26

[88] M. Adnan, S. Kalra, J. C. Cresswell, G. W. Taylor, and H. R. Tizhoosh, "Federated learning and differential privacy for medical image analysis," *Scientific reports*, vol. 12, no. 1, p. 1953, 2022. 26

[89] "Diagnostic image analysis group. the camelyon16 challenge." `https://camelyon16.grand-challenge.org/Data/`. Accessed: 2023-2-02. 26

[90] L. Vincent, "Morphological grayscale reconstruction in image analysis: applications and efficient algorithms," *IEEE transactions on image processing*, vol. 2, no. 2, pp. 176–201, 1993. 28

[91] J. D. Foley, F. D. Van, A. Van Dam, S. K. Feiner, J. F. Hughes, and J. Hughes, *Computer graphics: principles and practice*, vol. 12110. Addison-Wesley Professional, 1996. 29

[92] E. Gafton and S. Rosswog, "A fast recursive coordinate bisection tree for neighbour search and gravity," *Monthly Notices of the Royal Astronomical Society*, vol. 418, no. 2, pp. 770–781, 2011. 29

[93] R. Lubbe, W.-J. Xu, D. N. Wilke, P. Pizette, and N. Govender, "Analysis of parallel spatial partitioning algorithms for gpu based dem," *Computers and Geotechnics*, vol. 125, p. 103708, 2020. 29

[94] G. CYBENKO and T. G. ALLEN, "Multidimensional binary partitions: distributed data structures for spatial partitioning," *International Journal of Control*, vol. 54, no. 6, pp. 1335–1352, 1991. 29, 31

[95] M. Deveci, S. Rajamanickam, K. D. Devine, and Ü. V. Çatalyürek, "Multi-jagged: A scalable parallel spatial partitioning algorithm," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 803–817, 2015. 30

[96] A. Eldawy, L. Alarabi, and M. F. Mokbel, "Spatial partitioning techniques in spatialhadoop," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1602–1605, 2015. 30

[97] H. Vo, A. Aji, and F. Wang, "Sato: a spatial data partitioning framework for scalable query processing," in *Proceedings of the 22nd ACM SIGSPATIAL international conference on advances in geographic information systems*, pp. 545–548, 2014. 30

[98] R. A. Finkel and J. L. Bentley, “Quad trees a data structure for retrieval on composite keys,” *Acta informatica*, vol. 4, no. 1, pp. 1–9, 1974. 31

[99] R. T. Mullapudi, V. Vasista, and U. Bondhugula, “Polymage: Automatic optimization for image processing pipelines,” *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 429–443, 2015. 32, 34

[100] D. Wonnacott, “Using time skewing to eliminate idle time due to memory bandwidth and network limitations,” in *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pp. 171–180, IEEE, 2000. 32

[101] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, “A stencil compiler for short-vector SIMD architectures,” in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pp. 13–24, 2013. 33

[102] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, “High-performance code generation for stencil computations on GPU architectures,” in *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 311–320, 2012. 33

[103] B. Hagedorn, A. S. Elliott, H. Barthels, R. Bodik, and V. Grover, “Fireiron: A Data-Movement-Aware Scheduling Language for GPUs,” *29th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2020. 33, 34

[104] S. Memeti and S. Pllana, “HSTREAM: A directive-based language extension for heterogeneous stream computing,” in *2018 IEEE International Conference on Computational Science and Engineering (CSE)*, pp. 138–145, IEEE, 2018. 33, 34

[105] M. Sourouri, S. B. Baden, and X. Cai, “Panda: A Compiler Framework for Concurrent CPU + GPU Execution of 3D Stencil Computations on GPU-accelerated Supercomputers,” *International Journal of Parallel Programming*, vol. 45, no. 3, pp. 711–729, 2017. 33, 34

[106] P. Czarnul, J. Proficz, and K. Drypczewski, “Survey of methodologies, approaches, and challenges in parallel programming using high-performance computing systems,” *Scientific Programming*, vol. 2020, 2020. 35

[107] Z. Li, “Geospatial big data handling with high performance computing: Current approaches and future directions,” *High Performance Computing for Geospatial Applications*, pp. 53–76, 2020. 35

[108] A. Pupykina and G. Agosta, “Survey of memory management techniques for hpc and cloud computing,” *IEEE Access*, vol. 7, pp. 167351–167373, 2019. 35, 36

[109] B. Kong, Z. Li, and S. Zhang, “Toward large-scale histopathological image analysis via deep learning,” in *Biomedical Information Technology*, pp. 397–414, Elsevier, 2020. 36, 40, 64

[110] B. L. Chamberlain, D. Callahan, and H. P. Zima, “Parallel programmability and the chapel language,” *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007. 38

[111] C. Mendis, J. Bosboom, K. Wu, S. Kamil, J. Ragan-Kelley, S. Paris, Q. Zhao, and S. Amarasinghe, “Helium: Lifting high-performance stencil kernels from stripped x86 binaries to halide dsl code,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 391–402, 2015. 38, 67

[112] S. Sioutas, S. Stuijk, L. Waeijen, T. Basten, H. Corporaal, and L. Somers, “Schedule synthesis for halide pipelines through reuse analysis,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 2, pp. 1–22, 2019. 38

[113] R. R. L. Machado, A. M. Maidl, and D. Weingaertner, “Profiling halide dsl with cpu performance events for schedule optimization,” in *Proceedings of the XXIII Brazilian Symposium on Programming Languages*, pp. 38–45, 2019. 38

[114] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert, “Hipa cc: A domain-specific language and compiler for image processing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 210–224, 2015. 38

[115] J. Fang, C. Huang, T. Tang, and Z. Wang, “Parallel programming models for heterogeneous many-cores: a comprehensive survey,” *CCF Transactions on High Performance Computing*, vol. 2, pp. 382–400, 2020. 38

[116] M. Kertész, F. Csillag, and A. Kummert, “Optimal tiling of heterogeneous images,” *International Journal of Remote Sensing*, vol. 16, no. 8, pp. 1397–1415, 1995. 45

[117] X.-D. Liu, J.-Z. Wu, and C.-W. Zheng, “KD-tree based parallel adaptive rendering,” *The visual computer*, vol. 28, no. 6, pp. 613–623, 2012. 45

[118] H. Wang and A. Chandramowlishwaran, “Pencil: a pipelined algorithm for distributed stencils,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16, 2020. 45

[119] M. Wang, C.-c. Huang, and J. Li, “Supporting very large models using automatic dataflow graph partitioning,” in *Proceedings of the Fourteenth EuroSys Conference 2019*, pp. 1–17, 2019. 45

[120] I. Z. Reguly, G. R. Mudalige, and M. B. Giles, “Loop tiling in large-scale stencil codes at run-time with OPS,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 4, pp. 873–886, 2017. 45

[121] K. Wu, E. Otoo, and K. Suzuki, “Optimizing two-pass connected-component labeling algorithms,” *Pattern Analysis and Applications*, vol. 12, no. 2, pp. 117–135, 2009. 62

[122] O. Pearce, T. Gamblin, B. R. De Supinski, M. Schulz, and N. M. Amato, “Quantifying the effectiveness of load balance algorithms,” in *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 185–194, 2012. 65

[123] D. Matveev, “Opencv graph api,” *Intel Corporation*, vol. 1, 2018. 67

[124] OpenCV, “How to enable Halide backend for improve efficiency ,” 2021. available at `https://docs.opencv.org/3.4/de/d37/tutorial_dnn_halide.html`. 67

[125] Y. Zhang and Y. Zhang, “Making Halide Efficient for Multicore Systems,” in *2018 4th International Conference on Big Data Computing and Communications (BIGCOM)*, pp. 213–218, IEEE, 2018. 68