ABSTRACT

Title of dissertation: MODEL-BASED DESIGN FOR HIGH-PERFORMANCE SIGNAL PROCESSING APPLICATIONS

Jiahao Wu
Doctor of Philosophy, 2019

Dissertation directed by: Professor Shuvra S. Bhattacharyya
Dept. of Electrical and Computer Engr., and
Institute for Advanced Computer Studies

Developing high-performance signal processing applications requires not only efficient signal processing algorithms but also effective software design methods that can take full advantage of the available processing resources. An increasingly important type of hardware platform for high-performance signal processing is a multicore central processing unit (CPU) combined with a graphics processing unit (GPU) accelerator. Efficiently coordinating computations on both the host (CPU) and device (GPU), and managing host-device data transfers are critical to utilizing CPU-GPU platforms effectively. However, system designers find such coordination challenging, given the complexity of modern signal processing applications and the stringent constraints under which they must operate.

Dataflow models of computation provide a useful framework for addressing this challenge. In such a modeling approach, signal processing applications are represented as directed graphs that can be viewed intuitively as high-level signal flow diagrams. The formal high-level abstraction provided by dataflow principles provides a useful foundation

to investigate model-based analysis and optimization for new challenges in the design and implementation of signal processing systems.

This thesis presents a new model-based design methodology and an evolution of three novel design tools. These contributions provide an automated design flow for high performance signal processing. The design flow takes high-level dataflow representations as input and systematically derives optimized implementations on CPU-GPU platforms.

The proposed design flow and associated design methodology are inspired by a previously-developed application programming interface (API) called the Hybrid Task Graph Scheduler (HTGS). HTGS was developed for implementing scalable workflows for high-performance computing applications on compute nodes that have large numbers of processing cores, and that may be equipped with multiple GPUs. However, HTGS has a limitation due to its relatively loose use of dataflow techniques (or other forms of model-based design), which results in a significant designer effort being required to apply the provided APIs effectively.

The main contributions of the thesis are summarized as follows:

(1) Development of a companion tool to HTGS that is called the HTGS Model-based Engine (HMBE). HMBE introduces novel capabilities to automatically analyze application dataflow graphs and generate efficient schedules for these graphs through hybrid compile-time and runtime analysis. The systematic model-based approaches provided by HMBE enable the automation of complex tasks that must be performed manually when using HTGS alone. We have demonstrated the effectiveness of HMBE and the associated model-based design methodology through extensive experiments involving two case studies: an image stitching application for large scale microscopy images and a background

subtraction application for multispectral video streams.

(2) Integration of HMBE with HTGS to develop a new design tool for the design and implementation of high-performance signal processing systems. This tool, called HMBE-Integrated-HTGS (HI-HTGS), provides novel capabilities for model-based system design, memory management, and scheduling targeted to multicore platforms. HMBE takes as input a single- or multi-dimensional dataflow model of the given signal processing application. The tool then expands the dataflow model into an expanded representation that exposes more parallelism and provides significantly more detail on the interactions between different application tasks (dataflow actors). This expanded representation is derived by HI-HTGS at compile-time and provided as input to the HI-HTGS runtime system. The runtime system in turn applies the expanded representation to guide dynamic scheduling decisions throughout system execution.

(3) Extension of HMBE to the class of CPU-GPU platforms motivated above. We call this new model-based design tool the CPU-GPU Model-Based Engine (CGMBE). CGMBE uses an unfolded dataflow graph representation of the application along with thread-pool-based executors, which are optimized for efficient operation on the targeted CPU-GPU platform. This approach automates complex aspects of the design and implementation process for signal processing system designers while maximizing the utilization of computational power, reducing the memory footprint for both the CPU and GPU, and facilitating experimentation for tuning performance-oriented designs.

MODEL-BASED DESIGN FOR HIGH-PERFORMANCE SIGNAL PROCESSING APPLICATIONS

by

Jiahao Wu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2019

Advisory Committee:
Professor Shuvra S. Bhattacharyya, Chair/Advisor
Dr. Walid Keyrouz
Professor Joseph F. JaJa
Professor Manoj Franklin
Professor Donald Yeung
Professor Yang Tao, Dean's Representative

Dedication

To my family.

Acknowledgments

The completion of this dissertation would not have been possible without the support of many people.

First and foremost, I would like to express my sincere gratitude to my research advisor Professor Shuvra Bhattacharyya, for his unceasing support and unwavering guidance throughout the journey of my Ph.D. study. His patience and kindness make it a great pleasure to work with him.

I also want to thank my dissertation committee members. I would like to thank Dr. Walid Keyrouz for all the suggestions he provided for my research and careful proofreading on my dissertation. I am also appreciative of Professor Joseph JaJa, Professor Manoj Franklin, Professor Donald Yeung, and Professor Yang Tao for serving in my committee and providing insightful and valuable feedback on my dissertation.

I am grateful to Professor Jani Boutellier, Professor Marilyn Wolf, Dr. Timothy Blattner, Dr. William Plishker, Dr. Shuoxin Lin, Dr. Yanzhou Liu, Dr. Lin Li, Dr. Kyunghun Lee, Mr. Alexandre Bardakoff, Mr. Adrian Sapio, Mr. Honglei Li, and other colleagues and research project collaborators for all the thoughtful discussions and the wonderful time.

Last but not least, I give my special thanks to my family, for their unconditional love and support during my study.

This research underlying this thesis was supported in part by The National Institute of Standards and Technology (NIST).

Table of Contents

List of Tables

List of Figures

List of Abbreviations

APEG	Acyclic Precedence Expansion Graph
API	Application Programming Interface
BDF	Boolean Dataflow
BLAS	Basic Linear Algebra Subprograms
CP	Critical Path
CPU	Central Processing Unit
CGMBE	CPU-GPU Model-Based Engine
DIF	Dataflow Interchange Format
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
GSLD	Global Static Local Dynamic
HEFT	Heterogeneous Earliest-Finish-Time
HI-HTGS	HMBE-Integrated-HTGS
HMBE	HTGS Model-Based Engine
HTGS	Hybrid Task Graph Scheduler
ICP	Inverse Critical Path
MBS	Multispectral Background Subtraction
MDC	Multi-Dataflow Composer
MPSC	Multiple Producer Single Consumer
NUMA	Non-Uniform Memory Access
ORCC	Open RVC-CAL Compiler
PCIAM	Phase Correlation Image Alignment Method
PiSDF	Parameterized and Interfaced Synchronous Dataflow
PSDF	Parameterized Synchronous Dataflow
SADI	Single Actor, Dynamic Invocation
SAD	Shortest APEG Distance
SDF	Synchronous Dataflow
SPMC	Single Producer Multiple Consumer
SRSDF	Single Rate Synchronous Dataflow
SVD	Singular-Value Decomposition
SNR	Signal-to-noise Ratio
UMA	Uniform Memory Access
VCA	Vertex Component Analysis
WDC	Worker Thread Capacity
WSDF	Windowed Synchronous Dataflow

Chapter 1: Introduction

Modern platforms for signal processing are often equipped with multicore central processing units (CPUs) and massively parallel graphics processing units (GPUs). These platforms are useful for a wide variety of high-performance signal processing applications because such a hybrid CPU-GPU architecture can potentially exploit large degrees of parallelism to address challenging performance requirements. However, developing performance-oriented applications that simultaneously satisfy the following three requirements is difficult:

1. Effective in exploiting the computational capabilities of the targeted hardware system (e.g., CPU cores and GPUs).

2. Understandable from a performance perspective.

3. Relatively easy to implement and platform-portable.

Conventional techniques, based on statement- or function-level parallelism techniques, are not adequate for an application to take full advantage of modern signal processing platforms. Applications often contain parallelism that spans multiple loops and function calls; exploiting this parallelism can be critical for obtaining performance within these applications. Furthermore, these approaches are difficult for system designers to

understand and implement with high efficiency. They require designers to develop a specialized design for each combination of application and platform, and make it difficult to systematically leverage previous design experience when developing new implementations.

1.1 Motivation of Model-Based Design

When using modern multicore CPUs, an application's threading model plays an important role in determining whether computational resources can be used efficiently by the application. Spawning too many threads does not necessarily improve overall performance because the overhead of spawning, destroying, and context switching between threads can consume significant system resources. On the other hand, if the number of threads is too small, the application cannot utilize all available CPU cores. Moreover, multi-threading leads to potential problems associated with data races. Although designers can use locks and mutexes to guarantee mutually exclusive access to data, excessive use of such mechanisms can significantly degrade performance.

Just as a single thread cannot take full advantage of a multicore CPU, a single GPU kernel typically does not consume all the computational capacity of a powerful GPU device. For NVIDIA GPUs, system designers can use CUDA streams to execute multiple kernels concurrently; a CUDA stream is analogous to an OpenCL command queue. However, using CUDA streams incorrectly may lead to increased overhead due to device synchronization which pause all executing kernels on the GPU device. An example of a common operation that leads to device synchronization is GPU memory allocation.

In addition to coordinating CPU threads and GPU kernels, memory management is another important challenge in efficient CPU-GPU software implementations. Most CPU-GPU platforms have separate address spaces for device and host memory. To launch a kernel on the GPU, the system has to first ensure the data is available in device memory or copy it from host to device memory.

This need for transferring data between host and device memory raises a number of issues. First, although overlapping data motion with computation usually yields higher performance when compared to performing these steps sequentially, developing efficient schedules that incorporate such overlapped communication is challenging. Second, dynamically allocating GPU memory when needed triggers device synchronization and, as such, reduce GPU performance. Moreover, device memory is limited in size when compared to the much larger amounts of memory available in hosts. As such, it is not feasible to move all data to device memory at once when processing large data sets. In conventional CPU-GPU design flows, the designer is responsible for dividing a large data set into small batches that fit in device memory, organizing the transfer of these batches to the device, and overlapping the transfers with kernel executions.

The CUDA parallel programming platform and application programming interface (API) supports *unified memory* so data can be automatically moved between host and device memory. This feature can simplify program development. However, in analogy to virtual memory, it must be used with care because it does not nullify the cost of data transfers and designers must remain aware of these costs. A designer who wants to develop an efficient system must still exert a significant effort to solve the related problems of scheduling tasks on CPU cores and GPUs, and coordinating data movement for tasks

that require host-device communication.

The (*Hybrid Task Graph Scheduler*) (HTGS) is a recently introduced software tool that addresses the challenges of integrated scheduling and memory management for implementing scalable workflows for high-performance computing applications on compute nodes with high core counts and multiple GPUs [1]. HTGS includes novel abstractions, application programming interfaces (APIs), and runtime system components to facilitate the development and iterative optimization of scalable, high-performance, and performance-portable CPU-GPU implementations. It uses an explicit task graph representation of the application and delivers similar performance as manually developed systems while simplifying the code substantially and reusing existing compute kernels. However, HTGS has two shortcomings. It requires a developer to manually design memory management strategies (known as *memory release rules* in HTGS) and provides no theoretical guarantees as to the safe execution of tasks.

To address the problems described above, it is critical to view signal processing applications at a higher abstraction level that formally captures characteristics of global execution patterns. *Dataflow models of computation* provide such abstractions and are useful for many kinds of analyses and optimization objectives in design and implementation of signal processing systems [2, 3].

In this form of dataflow, the designer describes an application using a graph structure, where graph vertices represent computational modules (*actors*) and edges represent FIFO (First In First Out) queues for communication between computations. In general, the actors can be of arbitrary complexity, ranging from primitive operations to individual kernels to functions that invoke multiple kernels.

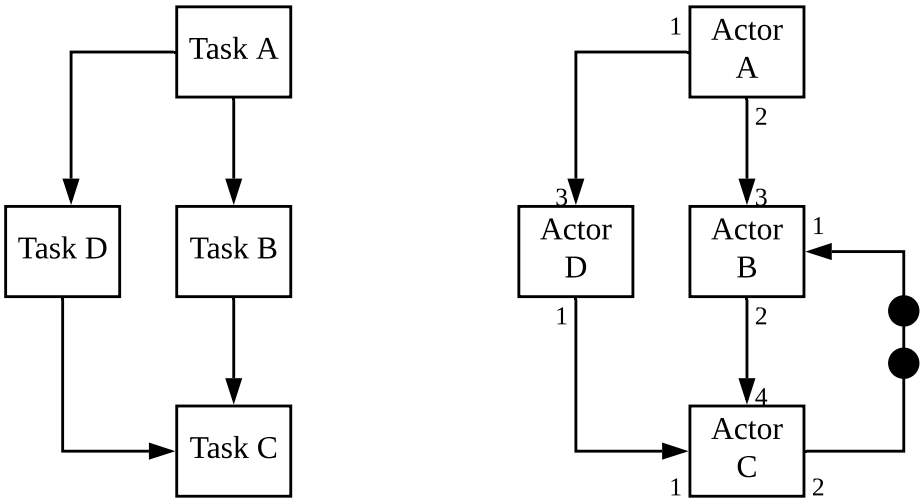
Fig. 1.1 illustrates differences between task graph models and dataflow models. The numbers next to the actor ports in Fig. 1.1(b) indicate the number of data values (*tokens*) that are produced or consumed on the associated output or input edges, respectively. For example, Actor D consumes 3 tokens and produces 1 token on each invocation. The emphasis on characterizing such production and consumption rates (in terms of tokens per actor invocation) is a distinguishing characteristic of signal-processing-oriented dataflow models of computation compared to tasks graph representations.

Additionally, different from the task graph model, where the topology is a directed acyclic graph (DAG) and all tasks execute at the same rate (in terms of invocations per graph iteration), a dataflow model allows the designer to model feedback paths as cycles in the graph, and allows actors to execute at different rates. This latter capability is important, for example, when modeling multirate signal processing systems. For more background on dataflow based modeling and design for signal processing systems, we refer the reader to Bhattacharyya et al. [3].

In this research, we address new challenges and constraints involved in the design and implementation of high performance signal processing systems using dataflow models. We outline the contributions of this thesis in the next section.

1.2 Contribution

In this thesis, we address important challenges in mapping signal processing applications into efficient implementations on hybrid CPU-GPU platforms. Our approach applies a model-based design methodology and introduces novel methods for dataflow-



(a) Task Graph Model

(b) Dataflow Graph Model

Figure 1.1: An illustration of differences between task graph models and dataflow models.

based analysis and optimization.

Fig. 1.2 shows a typical workflow of a dataflow-model-based design methodology. We develop a model-based design tool that automates the complicated and error-prone steps of exploring and exploiting parallelism for signal processing applications. To make it easier for designers to implement applications efficiently and understand performance bottlenecks, we have developed a novel dynamic scheduler in our design tool. By a *dynamic scheduler*, we mean a runtime subsystem that dynamically assigns actors to processing resources and determines the ordering of actors. The dynamic scheduler applies compile-time analysis of the dataflow model, and uses the results of this analysis to adapt scheduling decisions at runtime based on variations in actor execution times and interprocessor communication performance that are not statically predictable. The dynamic scheduler operates efficiently on hardware that supports a large degree of concurrency. Additionally, it generates task-level profiling results with minimal runtime overhead.

These profiling results help to provide insights about performance to system designers, which in turn aids in their process of iterative performance optimization.

To demonstrate the design methods and tools developed in this research, we employ case studies in the domain of high performance image processing. This is a domain that is of increasing relevance due to advances in areas such as computer vision and machine learning on visual data. We envision that the core contributions of the thesis can be adapted naturally to other signal processing domains. Exploring such adaptations is an interesting direction for future work.

The contributions of the thesis are presented in three main parts. First, we develop the HTGS Model-Based Engine (HMBE) scheduler as a separate tool that links dataflow based modeling and scheduling with the software component (actor) library of HTGS. Then we present a new version of HTGS in which HMBE is integrated seamlessly within HTGS rather than applied as a separate companion tool. We demonstrate that this new version provides significant further advantages compared to the previous, loosely-coupled integration of HMBE and HTGS. To distinguish it from the original HTGS and HMBE tools, we refer to the new integrated toolset as HMBE-Integrated-HTGS (HI-HTGS). Finally, we introduce the CPU-GPU Model-Based Engine (CGMBE), which builds on HMBE to extend the class of targeted systems to multicore CPU platforms that are combined with GPU accelerators.

We provide brief introductions to HMBE, HI-HTGS, and CGMBE in Sections 1.2.1, 1.2.2, and 1.2.3, respectively.

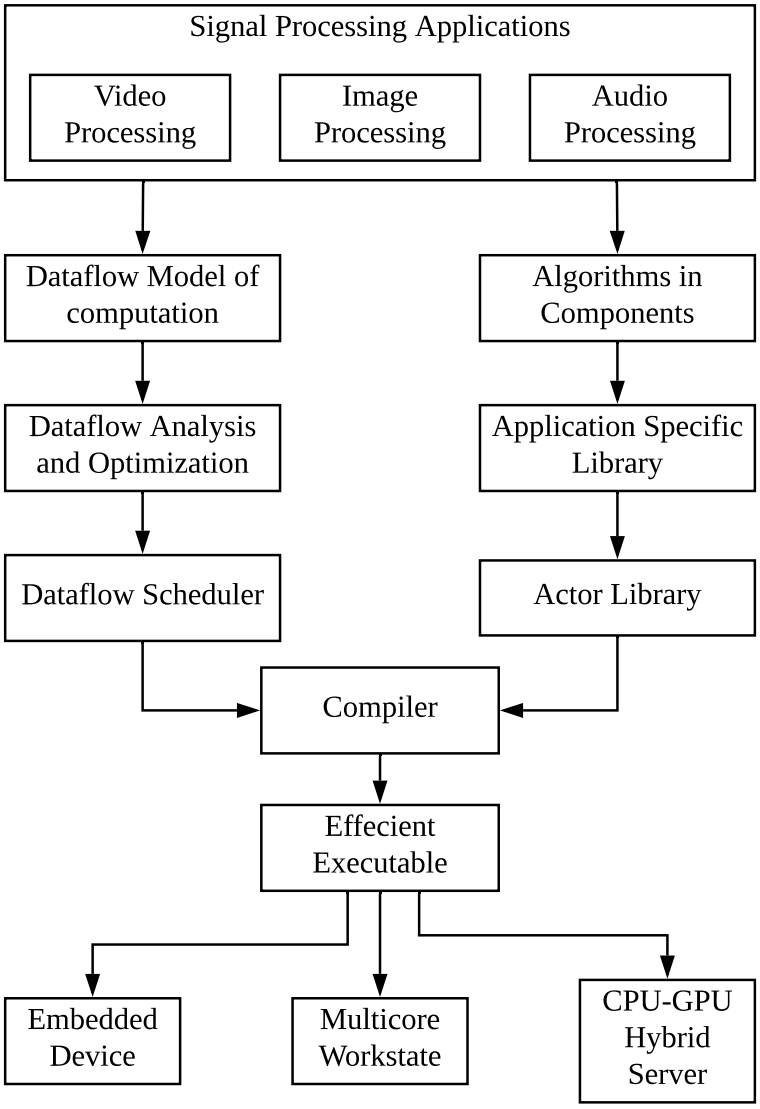


Figure 1.2: A typical workflow for a model-based, signal processing design methodology.

1.2.1 HTGS Model-Based Engine

To automate the powerful but complex steps in the HTGS framework, we first present a companion tool to HTGS, which we refer to as the *HTGS Model-Based Engine* (*HMBE*). The objective in the design of HMBE is to integrate the advanced design optimization techniques provided in HTGS with model-based approaches founded on dataflow principles. HMBE further raises the abstraction level of the graphical representations in HTGS-based workflows so that the graphs are formulated in terms of dataflow principles. HMBE thereby enables dataflow-based analysis and optimization techniques to be integrated with implementations using HTGS. The class of analysis and optimization techniques enabled by HMBE includes those associated with the *scheduling* of tasks and communication operations, which is a critical aspect of dataflow-based design processes for signal and information processing [4].

Modern multicore architectures require the use of threads to achieve high performance. However, threads provide a highly non-deterministic model of computation [5]. This non-determinism interferes with the application of conventional static or self-timed multiprocessor scheduling techniques for static dataflow models, such as Synchronous DataFlow (SDF) and Windowed SDF [6, 7], that rely on predictable execution of and interaction between execution subsystems [4]. Furthermore, when the predictability in the underlying execution model decreases, as in thread-based models, the assumptions made during static or self-timed scheduling are increasingly inaccurate and the efficiency of the resulting implementations suffers. This trend motivates the use of dynamic scheduling in HMBE.

It is important to have a high-level representation of the application model at compile-time to enable advanced analysis and optimization. It is equally important to have an expanded global representation of the application's execution state at runtime. The application model is feature-rich and flexible, while the execution state representation should be simple and efficient so as to reduce runtime overhead. To achieve these requirements, HMBE uses an automated conversion to bridge the gap between an expressive application model and a lightweight runtime model.

The HMBE scheduler provides lock-free and race-condition-free exploitation of parallelism without requiring programmer experience in parallel programming. To reduce the runtime overhead of the dynamic scheduler, HMBE performs most graph transformations and analysis at compile-time and exports an expanded model, which is efficient for the scheduler to process at runtime. HMBE integrates a modified form of ready list scheduling with an expanded representation of the input dataflow model that exposes parallelism effectively. The process of producing a multithreaded executable from the given input dataflow model is fully automated through the scheduling and code generation features incorporated in HMBE.

In Section 4.2.1, we show that HMBE yields comparable performance as HTGS. HMBE achieves this comparable level of performance while greatly reducing programmer effort through automation of scheduling and memory management processes. We also experiment with HMBE on a background subtraction application for multispectral video streams to demonstrate that HMBE can effectively exploit parallelism in applications that have large amounts of sequential execution constraints.

Details of the HTGS Model-Based Engine are presented in Chapter 4.

1.2.2 HMBE-Integrated HTGS

HTGS significantly reduces the complexity of implementing scalable and high-performance multicore applications through a task-graph based application representation [1], and novel provisions for integrating optimizations for high-performance multicore processing and memory management within this representation. Using HTGS effectively requires iterative performance tuning, which in turn requires significant designer effort to derive efficient schedules and manage memory usage patterns.

First, we develop the HMBE-based scheduler as a separate tool that links dataflow based modeling and scheduling with the software component (actor) library of HTGS. Then we present a new version of HTGS in which HMBE is integrated seamlessly within HTGS rather than applied as a separate companion tool. We demonstrate that this new version provides significant further advantages compared to the previous, loosely-coupled integration of HMBE and HTGS. To distinguish it from the original HTGS and HMBE tools, we refer to the new integrated toolset as HMBE-Integrated-HTGS (HI-HTGS).

HI-HTGS automates major parts of the HTGS-based design process using the model-based design methodology from HMBE, along with new developments that integrate HMBE tightly as a subsystem within HTGS. Our development of HI-HTGS includes three major advances. First, we resolve the semantic mismatch between the schedulers of HTGS and HMBE through a hierarchical scheduling approach in which a global scheduler interacts with multiple local schedulers to automatically optimize overall memory usage and execution time. Second, we replace the application modeling interface with one that is based on the Dataflow Interchange Format (DIF) Language [8]. Third, we

migrate most of the setup-time and runtime analysis to compile-time, which results in a significantly more efficient, lightweight runtime system.

We demonstrate the effectiveness of HI-HTGS through a case study involving an image stitching application for large scale microscopy.

Details of HMBE-Integrated HTGS are presented in Chapter 5.

1.2.3 CPU-GPU Model-Based Engine

The contributions of the CPU-GPU Model-Based Engine (CGMBE) build upon our development of HMBE and HI-HTGS. HMBE builds, in turn, on advances introduced in HTGS and provides novel models and methods that provide a principled approach, based on the Windowed Synchronous Dataflow (WSDF) model of dataflow systems [7]. WSDF is applied in HMBE to guarantee the absence of deadlocks and race conditions, and to automate some steps that HTGS had relegated to developers [9]. However, HMBE is restricted to target platforms that consist only of CPU cores; it does not provide support for GPU-accelerated platforms.

In Chapter 6, we introduce a new model-based design tool, the CPU-GPU Model-Based Engine (CGMBE), for image processing system design. CGMBE extends HMBE to target multicore CPU platforms with single-GPU acceleration. In view of the significant challenges discussed earlier in this chapter for CPU-GPU implementation, CGMBE incorporates major new models and methods compared to the first-version tool, HMBE. CGMBE replaces the HMBE runtime engine with an executor pattern [10] to achieve the separation of concerns between (1) the management of the global state of computation

of the dataflow graph, including the selection of ready tasks that can be executed asynchronously, and (2) the coordination of compute resources and task execution based on the availability of resources and the priority of tasks.

CGMBE allows a designer to specify applications using a flexible model of computation that integrates the Synchronous Dataflow (SDF) model [6] with an extensible class of bounded-memory patterns that allows pre-defined forms of dynamic dataflow behavior. CGMBE provides important theoretical guarantees—in particular, determinate, deadlock-free, race-free and bounded memory execution—for implementations that are developed using the tool. Furthermore, CGMBE incorporates a novel hybrid compile-time/runtime scheduling strategy for the efficient scheduling of dataflow graphs that may incorporate both SDF and dynamic dataflow subsystems.

Through an automated process, CGMBE performs most of its exploration of parallelism at compile-time, and caches results of its compile-time analysis for use by the runtime system. The runtime system executes asynchronously and overlaps data motion with computations to maximize the utilization of all compute resources. Additionally, CGMBE provides efficient profiling capabilities with very low runtime overhead. This low profiling overhead allows developers to readily gain insight into the performance bottlenecks of an application, such as the most time-consuming task. They can then iterate over their system-level design by moving tasks between the CPU and GPU or changing the structure of their application's model. They can also be more targeted when working with algorithm designers to improve specific kernel implementations.

Details of the CPU-GPU Model-Based Engine are presented in Chapter 6.

1.3 Outline of the Thesis

The remainder of this thesis is organized as follows. Chapter 2 introduces related software tools and background on signal-processing-oriented dataflow models. In Chapter 3, we present dataflow modeling techniques that are used extensively in this research. The next three chapters present the main contributions of this thesis: Chapter 4 presents the *HTGS Model-Based Engine* (*HMBE*), Chapter 5 presents HMBE-Integrated-HTGS (HI-HTGS), and Chapter 6 presents the CPU-GPU Model-Based Engine (CGMBE). Finally, in Chapter 7, we summarize the thesis and discuss directions for future work.

Chapter 2: Background

The efficient development of applications on CPU-GPU platforms has been widely studied from various aspects, which include developing programming APIs, presenting static or dynamic schedulers, and proposing formal models for applications or hardware platforms. A taxonomy of task-based parallel programming methods has recently been proposed in Thoman et al. [11]. This taxonomy is useful for classifying task-based environments. By analogy with this taxonomy, we classify parallel programming environments from the aspect of formal modeling, where the functional organization of components have a model-based definition, existing approaches can be categorized into task-based systems, such as StarPU and OmpSs [12, 13], and workflow-based systems, such as Pegasus [14].

2.1 Task-based and Workflow-based Systems

These task-based systems do not address the problem in its entirety, especially in the context of real-time image processing systems, where predictable, real-time operation on continuous image streams is of major importance. For example, StarPU allows developers to provide alternative CPU and GPU implementations for a kernel, where the alternative implementations will be invoked automatically or explicitly. However, StarPU

expects developers to manage memory for both the CPU and GPU, and does not provide a runtime representation of the application. Without access to a model-based runtime representation, it is difficult for StarPU to optimize application execution automatically at runtime, and developers have to manually ensure that their StarPU-based applications are free from deadlocks or data-race conditions.

A second example of a task-based system is OmpSs. OmpSs uses a traditionally supported, directive-based approach to running heterogeneous task kernels on GPU or FPGA devices by extending OpenMP with new directives [13]. To use OmpSs, one has to use the Mercurium compiler and the Nanos++ runtime [15]. OmpSs provides some support for reducing the complexity of application mapping via user-specified directives (e.g., automated memory allocation and data motion). Nevertheless, OmpSs requires developers to shoulder most of the burden of developing parallel software systems, including designing memory access patterns, reducing memory usage, and avoiding data-races and deadlocks.

Legion [16] is another task-based system for CPU-GPU platforms. Legion uses a data-centric programming model for distributed heterogeneous architectures. The Legion system requires developers to explicitly define static data properties for tasks and then uses this information to manage data movement operations. However, in cases involving dynamic or data-dependent task execution, where data dependencies are undefined until runtime, there is insufficient information about dataflow properties to enable Legion to extract parallelism effectively.

Other task-based approaches include PaRSEC, PLASMA and NLAFET [17–19]. PaRSEC is an event-driven system whose runtime can perform dynamic, opportunis-

tic scheduling decisions. PaRSEC represents an algorithm as a Directed Acyclic Graph (DAG) and its front-end compiler performs dataflow analysis from source code and produces a parameterized task graph. However, PaRSEC's task graph model is less expressive than a full-fledged dataflow model. PaRSEC's model is limited to DAGs and assumes single token rates for task invocation whereas a full-fledged dataflow model allows cycles and more flexible First-In-First-Out (FIFO) rates. PLASMA can be viewed as a redesign of LAPACK [20] and ScaLAPACK [21]. PLASMA relies on tiling algorithms to provide fine granularity parallelism [22]. Parallel Numerical Linear Algebra for Extreme Scale Systems, also called NLAFET, which seeks to develop numerical algorithms and software libraries that take advantage of Exascale systems [23]. Their basic approach is to develop parallel algorithms that are architecture-aware and avoid communication and synchronization, and that explore advanced scheduling strategies and runtime systems. However, NLAFET primarily targets linear algebra software and it is not clear how their approach can be used in applications that are not centered on a small number of linear algebra operations.

Workflow-based systems such as Pegasus are designed to automate computational workflows at the level of the grid and to provide traceability of results [24]. These workflow systems invoke independent applications and are not designed to deliver performance within a single application that will run on a powerful node with many CPU cores and possibly multiple GPUs.

Compared to task-based tools, the model-based design methodologies presented in this thesis utilize dataflow methods to operate at a higher abstraction level, and exploit parallelism beyond what is derived from data dependencies among tasks. As such, these

methodologies can yield more thorough optimization. Rather than scheduling multiple applications on a grid of nodes as considered in related workflow approaches, this research focuses on efficiently scheduling a single application on a single powerful node to increase application performance and reduce execution time. This latter design scenario is typical in application domains of embedded signal processing.

2.2 Dataflow-based Systems

Dataflow graph approaches allow the designer to model applications at a higher abstraction level than that of conventional programming environments and enable automated parallelism exploration and optimizations in a top-down manner (e.g., see [2, 3]). Various tools have been developed for modeling signal and information processing applications in terms of dataflow graphs, and automatically generating hardware or software implementations from the model-based representations. Examples of such tools include CAL/Orcc [25, 26], PREESM [27], Halide [28], the Multi-Dataflow Composer (MDC) [29], HTGS (introduced in Chapter 1), and the Dataflow Interchange Format (DIF) [30].

CAL/Orcc and PREESM provide user-friendly graphical interfaces to enable visual editing of dataflow graphs. These tools also provide code generation capabilities. DIF features a textual language that supports a wide range of dataflow models. It is also accompanied by a library of analysis and code generation tools that operate on supported models within the tool. Halide is a programming language for image processing that employs dataflow modeling techniques and incorporates features for decoupling algorithm

representations from execution strategies. It represents an implementation as an algorithm and a schedule. The Halide compiler uses both representations to generate optimized machine code. Halide allows a developer to experiment with different schedules to achieve better performance.

HTGS introduces new methods for integrating optimized memory management with the design of task graphs and the implementation of high-performance multithreaded applications [1]. The carefully formulated APIs of HTGS enable the construction of code that is highly optimized for complex high-performance platforms, while also providing scalability and ease of performance tuning across different types of platforms. A limitation of HTGS is its relatively loose use of dataflow techniques (or other forms of model-based design), which results in a significant designer effort being required to apply the provided APIs effectively. Although the methods developed in this thesis are driven by developing complementary capabilities to HTGS, their formulation in terms of formal dataflow principles, such as Synchronous Dataflow (SDF), Windowed Synchronous Dataflow (WSDF), Acyclic Precedence Expansion Graphs (APEGs), and unfolded Homogeneous Synchronous Dataflow (Unfolded HSDF) makes them readily adaptable to other environments. Details on the use of SDF, WSDF, APEG, and unfolded HSDF representations in our model-based design methodology are presented in Chapters 4 and 6.

2.3 Scheduling Models

In this thesis, we go beyond the developments of HTGS by raising the level of abstraction further and casting design optimization problems targeted by HTGS in terms

of dataflow principles. This results in a novel approach to dynamic scheduling for heterogeneous platforms that use expanded dataflow graph representations of applications (i.e., APEGs) at runtime to guide scheduling decisions. These expanded representations expose parallelism in greater detail when compared to more compact dataflow representations that are typically used by designers [6]. Our work also provides a basis for relating the advances in HTGS to systematic dataflow-based design methods and automation techniques. Furthermore, the contributions of this thesis have significant potential for adaptation to other environments.

Lee and Ha develop a taxonomy for scheduling models based on whether certain scheduling tasks are performed at compile time or runtime [31]. These models provide a range of trade-offs between predictability and flexibility, and include self-timed, static-assignment, fully-static, and dynamic models. The concept of "assignment" in these models is that of assigning computational tasks to processors. Among these models, the self-timed and static assignment models have natural adaptations to the thread-based implementation context targeted in this thesis. In these adaptations, which we refer to as the *adapted self-timed* and *adapted static assignment* models, the assignment of tasks to threads (rather than to processors) is performed at compile-time. In adapted self-timed scheduling, the ordering of tasks that share the same thread is performed at compile time, while in adapted static assignment scheduling, this ordering is performed at run time.

Kwok and Ahmad [32] provide an extensive survey of *static scheduling* approaches, which fix the mapping from actors to processors and their invocation orders at compile time. An example of such an approach is Heterogeneous Earliest-Finish-Time (HEFT) [33], which aims at statically scheduling task graphs onto heterogeneous platforms. Such ap-

proaches can be used to schedule dataflow-based application models that conform to SDF semantics (see Section 6.1). However, to target platforms that provide massive concurrency, such as modern CPU-GPU-hybrid heterogeneous platforms, it is useful for schedulers to incorporate *dynamic scheduling* capabilities. This is due to the highly non-deterministic characteristics of execution models associated with CPU threads and GPU streams.

In HMBE, we apply a scheduling model that lies between adapted static assignment and adapted self-timed scheduling in terms of the trade-off between predictability and flexibility. We refer to this model as the *Single Actor, Dynamic Invocation* (*SADI*) assignment. In a SADI assignment, a given thread is dedicated at compile time to a single actor, while the set of actor firings that executes through that thread is determined dynamically. We elaborate more on the SADI scheduling model and its application to HMBE in Chapter 4. Intuitively, we prefer SADI to adapted self-timed scheduling because the lack of predictability in the thread-based model of computation (see Chapter 1) makes adapted self-timed schedules too rigid in our context.

Additionally, HMBE is generalized to incorporate support for actors that have internal states. These actors are represented in an expanded intermediate representation with dependence edges that connect successive firings of the actors. Two case studies are discussed to help demonstrate the ability of HMBE to handle applications with diverse characteristics. The first case study, presented in Section 4.2.1, involves a large-scale image stitching application, which is data-intensive and has complicated data dependencies. The second case study, presented in Section 4.2.2, involves a background subtraction application for multispectral video streams. The background subtraction application ex-

hibits a significant amount of sequential scheduling dependencies and many actors in this application have internal states.

HI-HTGS uses HMBE to automate the design process of HTGS's local schedulers, which are also referred to as *bookkeeper rules* in HTGS. HI-HTGS embeds the results of HMBE-based analysis in a generalized bookkeeper rule, which automatically resolves data dependencies for adjacent HTGS tasks. HI-HTGS also utilizes a hierarchical scheduling approach, where local schedulers resolve data dependencies and a global scheduler coordinates execution across the local schedulers. Details of the hierarchical scheduling approach used in HI-HTGS are presented in Section 5.3.

CGMBE further extends the capabilities of HMBE. More specifically, the novelty in CGMBE is its inclusion of a unique combination of models and methods that are useful for mapping complex image processing applications onto CPU-GPU platforms. First, CGMBE incorporates a dynamic scheduler (part of its runtime system), which efficiently utilizes both the CPU and GPU without statically binding tasks to threads or cores. Second, CGMBE provides the ability to support dynamic dataflow application behavior by systematically integrating well-behaved (bounded memory) patterns of dynamic behavior. Third, CGMBE separates the management of the execution state of the dataflow model and the coordination of computation resources. This separation gives the system designer a simple API to implement dataflow-based image processing systems, while providing flexibility if the designer wants to customize the execution.

Moreover, while HMBE and HI-HTGS are targeted to multicore platforms, CGMBE targets the more general class of platforms that consist of a multicore platform together with a GPU accelerator.

Chapter 3: Modeling

In this chapter, we introduce the modeling techniques used in HMBE and CGMBE respectively. Material in this chapter was published in partial, preliminary form in Wu et al. [9, 34].

HMBE maintains two dataflow-based models of an application: (1) a compact representation, the *application graph*, which may incorporate multirate dataflow semantics, and (2) an expanded single-rate representation, which exposes more explicitly the dataflow relationships and parallelism among individual actor firings within an application. The expanded representation is based on multirate-to-single-rate expansion concepts introduced by Lee and Messerschmitt in the context of the synchronous dataflow (SDF) model of computation [6]. We refer to the expanded representation as the *Acyclic Precedence Expansion Graph* (*APEG*).

The application graph is the system designer's entry point to the HMBE system, whereas the APEG is an intermediate representation that is derived from the application graph and used by the HMBE runtime system to guide dynamic scheduling decisions. HMBE generates the APEG automatically as part of the compilation process.

The core scheduling techniques used in HMBE can be applied to any dataflow model of computation which (a) has a well-defined concept of a graph iteration, and

(b) can represent the dependencies statically among individual firings through an APEG representation. In this context, the APEG representation X of an application graph G is a second graph in which individual actor firings (within an iteration of G) correspond to vertices in X and an edge connects vertices v_1 to v_2 in X if and only if x_1 produces data consumed by x_2 in the same graph iteration. Here, x_1 and x_2 are the actors in G associated with the firings v_1 and v_2, respectively.

If the underlying dataflow model is sufficiently predictable, then we can construct expanded representations by adapting the SDF-based APEG construction mechanisms introduced by Lee and Messerschmitt [6]. Such adaptations have been demonstrated, for example, with the multidimensional synchronous dataflow, and cyclo-static dataflow models of computation [35,36]. HMBE uses APEGs for dynamic scheduling via SADI at runtime in contrast with conventional methods that apply APEGs to scheduling problems that are focused on static scheduling techniques.

HMBE adapts Keinert's *Windowed Synchronous DataFlow* (*WSDF*) model [7] and uses it as the foundation for application graph modeling. A main advantage of WSDF is that it efficiently models the sliding-window behavior of an application. This is directly applicable to high-performance image processing, an application area of particular interest to the HTGS and HMBE projects. At the same time, WSDF belongs to the class of models for which APEGs can be derived and is, therefore, compatible with the central assumption for application modeling in HMBE.

The adaptations that we incorporate in WSDF are relatively minor and stem from our focus on applications that process very large-scale individual images, rather than streams of images. These adaptations include the following aspects.

- The adapted WSDF model simplifies the multi-dimensional balance equations because the model is not used to represent streaming applications.

- The adapted model focuses on coordinating image tokens generated from each step of the sliding window, while the original WSDF model has more complicated but powerful parameters for modeling sliding window behavior on image pixels.

Because the methods developed in this thesis are applicable to any form of dataflow that supports the derivation of APEGs, neither these adaptations to WSDF nor the use of WSDF itself are limiting factors of our work. For details on WSDF, we refer the reader to Keinert [7]. Because the adaptations to WSDF that we employ in HMBE are minor, we do not distinguish in the remainder of this thesis between the original WSDF model and our adapted form of WSDF.

Fig. 3.1 is a concrete example of the WSDF model used in this thesis, This model contains three actors and two edges. Because the actor `SRC` is a source actor, it has a frame size associated with it; this frame size is $f_{\text{src}} = (3,4)$. The `SRC` actor reads tokens from a file, using a 3×4 grid, as shown in Fig. 3.1.a. It then produces tokens on edge e_1 as a grid of tokens; this is illustrated in Fig. 3.1.b. For edge e_1, the window size is $c_1 = (1,2)$, which means that each window encompasses 1 row and 2 columns. The dashed arrows indicate the direction of window movement from one actor firing to the next. For example, since $d_1 = (1,1)$, the window moves right 1 token a time until it reaches the right boundary of the grid after 3 movements. Then the window moves to the beginning of the second row. Similar edge attribute notations are used on edge e_2. Because actor `A` is invoked three times per row, the token grid on edge e_2 has size 3×3.

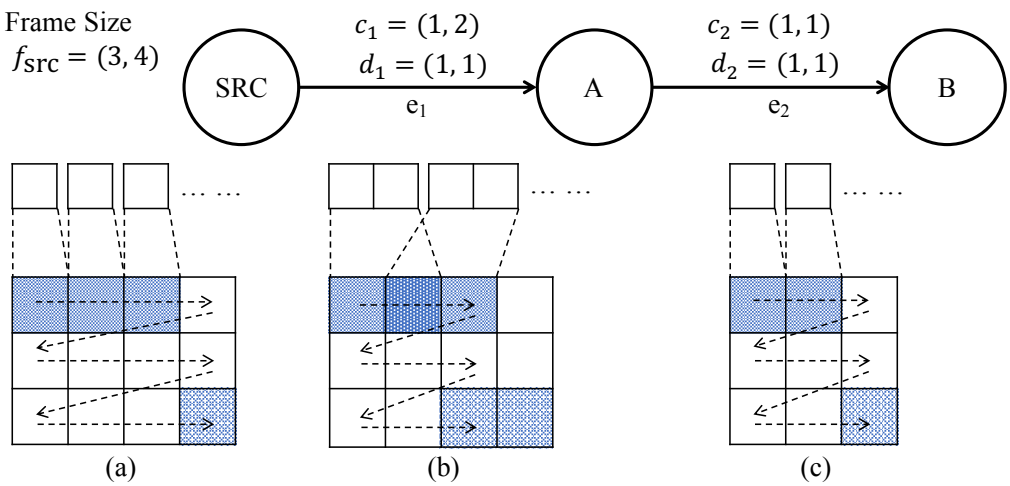


Figure 3.1: A demonstration of the adapted WSDF model used in this thesis.

Fig. 3.2 illustrates the design flow for HMBE-based development of high-performance image processing applications. In this design flow, an implementation for a given problem description is first developed in terms of WSDF semantics. This implementation has two parts: the application model, which describes the dataflow graph, and the actor library, which provides detailed functionality for the individual actors in the graph. In the current version of HMBE, the application model is specified using *protocol buffers*, which provide methods and tools for representing, reading, and writing serialized data structures [37], and the actors are implemented in C++ following interfacing conventions similar to those in HTGS [1]. The protocol buffer aspect of the prototype is represented by the blocks labeled Prototxt Format and Protobuf Library, which are defined as part of the protocol buffer toolset [37].

The application model is stored internally within HMBE as a WSDF graph (a WSDF Object) and the graph is converted automatically by HMBE into an APEG representation. The WSDF and APEG graph data structures and the analysis techniques in HMBE that operate on them build on the Boost Graph Library [38, 39]. The executable

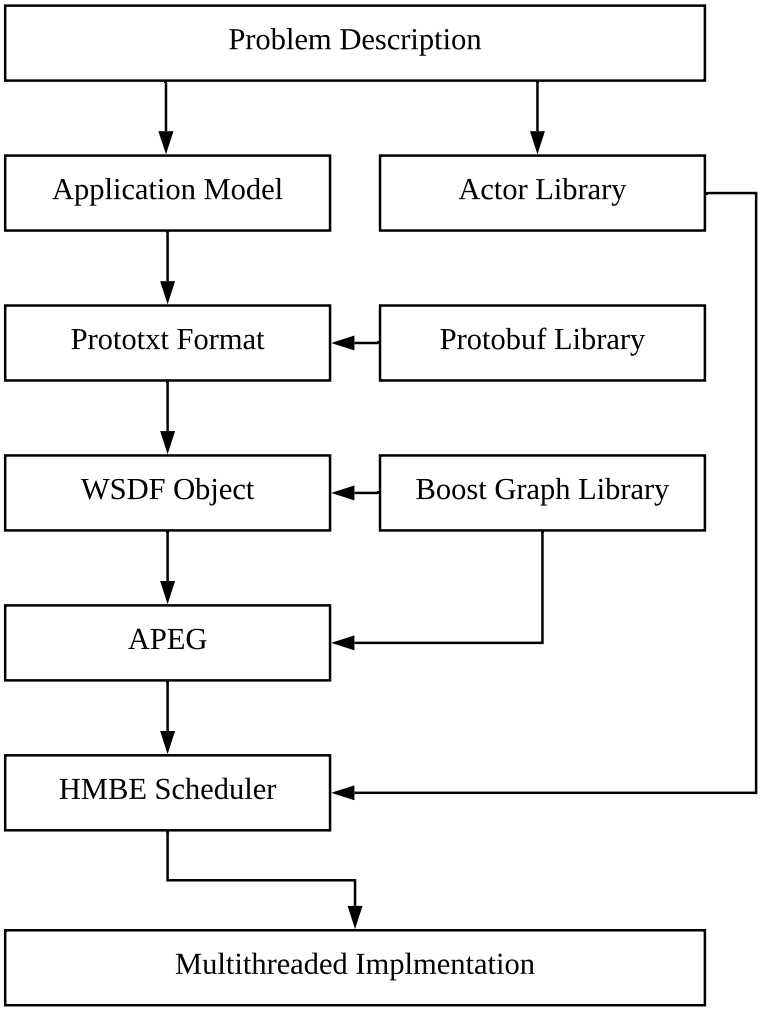


Figure 3.2: An illustration of the design flow for HMBE-based development of high-performance image processing applications.

for the multithreaded implementation (the bottom-most block in Fig. 3.2) is constructed by linking the actor library and HMBE scheduler. This executable embeds the APEG that is constructed at compile time so that the HMBE scheduler can access it to guide dynamic scheduling decisions.

Chapter 4: Scheduling

In this chapter, we present the *HTGS Model-Based Engine* (*HMBE*), which is a novel tool for the design and implementation of dataflow-based signal processing applications on multi-core architectures. HMBE provides complementary capabilities to HTGS, which was introduced in Chapter 1. HMBE integrates model-based design approaches, founded on dataflow principles, with advanced design optimization techniques provided in HTGS. This integration contributes to the following goals.

- Making the application of HTGS more systematic and less time consuming,
- Incorporating additional dataflow-based optimization capabilities with HTGS optimizations, and
- Automating significant parts of the HTGS-based design process using a principled approach.

This chapter describes the design of HMBE with an emphasis on the novel dynamic scheduling techniques that are developed as part of the tool. We demonstrate the utility of HMBE through two case studies: an image stitching application for large-scale microscopy images and a background subtraction application for multispectral video streams.

Material in this chapter was published in partial, preliminary form in Wu et al. [9, 34].

4.1 HMBE Scheduler

In this section, we present the SADI scheduling technique used in HMBE and which we refer to as the *HMBE scheduler*. In this development, we denote the input dataflow application graph by G_{app}, the associated APEG by G_{apeg}, and the number of processor cores in the target platform as N_C. Furthermore, we refer to actors (vertices) in G_{apeg} as *firings*. Given a dataflow graph $G = (V, E)$, the set of actors V in G is $actors(G)$ and the set of edges E is $edges(E)$.

In discussing the HMBE scheduler, it is useful to distinguish between three different stages of dataflow graph implementation and execution. The first stage, *compile-time*, refers to the steps involved in mapping the dataflow-based application model into an executable program P. The second stage, *setup-time*, refers to the steps involved in initializing the execution of the compiled dataflow graph. This includes all operations between τ_P and τ_f, where τ_P denotes the time at which Program P begins execution, and τ_f denotes the time when the first actor firing f begins execution. The third stage, referred to as *runtime*, encompasses all operations that are executed during the given execution of P from τ_f onward.

4.1.1 Allocation of Threads

The HMBE scheduler is a SADI scheduler that operates on a set of threads that is allocated at setup-time. This allocation is performed at setup-time rather than compile-time because (1) it allows the developer to control the degree of concurrency for each task and (2) threads of the application process exist only when the application starts running. One of the allocated threads, called the *control thread*, encapsulates the dynamic scheduling functionality, and the other threads, called *worker threads*, carry out execution of firings from G_{apeg} under the direction of the control thread.

At setup time, each actor $A \in actors(G)$ is allocated a set of worker threads $S(A)$. Each thread $t \in S(A)$ is dedicated to executing firings of A throughout the execution of G_{app}. Thus, each worker thread t has a unique actor $\alpha(t) \in actors(G_{app})$ that it is assigned to. For each actor A, the number (thread count) of threads $N_T(A)$ (i.e., the cardinality of $S(A)$) can be specified by the system designer before executing an HMBE-based implementation. If $N_T(A)$ is not specified by the designer, it is configured using the default setting $N_T(A) = N_C$. The construction of the sets $\{S(A) \mid A \in actors(G)\}$ is deferred to setup-time to allow the designer to configure $N_T(A)$ differently for different executions of the application graph (e.g., as part of an iterative design space exploration and optimization process).

At any given time during execution, a thread is in one of two states, which we refer to as the *dormant* and *active* states. A dormant thread is blocked until it receives a specific type of message from another thread which then transitions the dormant thread to its active state. When a thread becomes blocked, its associated processor core is released

so that the core can execute other threads.

A worker thread w that is dormant becomes active when it receives an *activation message* from the control thread. Upon receiving such an activation message, the thread proceeds to execute a single firing of $\alpha(w)$. Once this firing is complete, the worker thread sends a *completion message* to the control thread, becomes dormant again, and waits for its next activation message from the control thread.

The control thread becomes dormant when the number of active threads reaches a threshold, which we refer to as the *Worker Thread Capacity* (*WTC*) of the given HMBE scheduler configuration. WTC is a scheduler parameter that can be varied, in general, at runtime although our current implementation assumes that WTC's value is determined at setup-time and remains fixed throughout the execution of G_{app}. Investigation of configuring WTC at runtime is an interesting direction for future work. Section 4.1.3 presents further details about WTC. When the control thread is dormant, it remains in its dormant state until it receives a completion message from a worker thread. At that point, the number of active threads is known to have decreased below WTC, so the control thread can transition to its active state.

When the system begins execution, the control thread is initialized to be in its active state while all worker threads are initialized to be in their dormant states.

4.1.2 Dynamic Activation of Threads

Through its model-based approach, including the use of WSDF application modeling and an APEG-based intermediate representation, the HMBE scheduler provides lock-

free and race-condition-free exploitation of parallelism. This approach helps to mitigate some of the unpredictability of the underlying thread-based execution model. However, the unpredictability in thread execution times and thread-to-core assignment remains. The SADI scheduling approach in HMBE is designed to dynamically adapt the assignment of actors to threads and to dynamically activate threads to optimize performance in the presence of such uncertainty.

The HMBE scheduler uses a modified form of *ready list scheduling*, a form of scheduling which maintains a list of the tasks with sufficient data available to execute [4, 40]. When a processing resource becomes available, the scheduler selects a task from the ready list based on a greedy criterion and assigns the task to the processing resource. Different greedy criteria, such as a critical path metric (e.g., refer to Sriram [4]), can be incorporated into HMBE to select the next firing for execution from the ready list. When task θ completes execution, the scheduler updates the ready list to include tasks that have become ready (have sufficient input data) as a result of the completion of θ.

In the current version of HMBE, as described above, a task is determined to be ready if it has sufficient data. The availability of sufficient empty space in output buffers is not included in the criterion for actor "readiness". This is because HMBE uses dynamic memory allocation to expand buffer sizes on demand. Furthermore, dataflow graphs in HMBE are executed with pre-specified finite numbers of iterations. This prevents problems due to unbounded accumulation of tokens within buffers. HMBE is readily adaptable to incorporate checking for empty buffer space and processing of graphs without predefined iteration count limits. This is a useful direction for further work on HMBE.

HMBE modifies ready-list scheduling to maintain a separate ready list $L(A)$ for each

WSDF actor A. At any given time during execution, $L(A)$ contains the set of firings of A (APEG vertices associated with A) whose *precedent firings* have completed execution. Here, a precedent firing of an APEG vertex v_1 is a vertex v_2 such that there is an APEG edge directed from v_2 to v_1.

By using separate "actor-level" ready lists in HMBE, different criteria can be used to select the next WSDF actor for execution or the next firing of the selected actor (in case the firings are independent). In the current version of HMBE, we use a pair of simple selection criteria that use the inverse of the critical path (lower critical path values receive higher priority) and the shortest distance from the APEG vertex that represents the first actor firing during execution. We refer to these criteria as *Inverse Critical Path* (*ICP*) and *Shortest APEG Distance* (*SAD*), respectively. The ICP and SAD criteria are used to select the next actor and next firing, respectively. These heuristic criteria are chosen in an effort to reduce the lifetime of intermediate data throughout graph execution. Such lifetime reduction is important when memory management is a dominant concern, as in the targeted class of large-scale data-intensive image processing applications. Deeper investigation into alternative ready-list selection criteria in HMBE is a useful direction for future work.

The HMBE scheduler examines only the APEG's execution boundary (i.e., the set of ready actors) at any given scheduling iteration. As a result, dynamic processing of the APEG is quite efficient even when the overall graph is very large. Furthermore, dynamic use of the ready list eliminates the need for synchronization operations by worker threads, which can lead to significant reduction in contention for shared communication and memory resources. The control thread has exclusive access to the ready list so no

lock is needed to implement the list.

4.1.3 Scheduler Operation

The HMBE scheduler maps each WSDF actor onto a group of dedicated threads, as described in Section 4.1.1. The scheduler may activate more threads at a given time than N_C, the number of CPU cores. To help avoid excessive contention among active threads, HMBE limits the number of active threads at any given time to WTC, a parameter introduced in Section 4.1.1. More specifically, WTC is determined at setup time as $(WTC = (1+\varepsilon) \times N_C)$, where ε is a positive real-valued parameter that a designer can specify when launching an HMBE-based application. We envision that ε will typically be in the range of 0 to 0.5 and we provide a default setting of $\varepsilon = 0.2$ in case ε is not explicitly configured by the designer. As ε is increased, an increasing number of active threads is allowed to coexist. Excessively large values of ε lead to large amounts of contention and context switching among active threads, and degrade performance. Optimization of ε is intended to be part of the iterative performance tuning process that is supported by the HMBE framework.

Fig. 4.1 illustrates the operation of control and worker threads. The two boxes, *Read Message* and *Poll Messages*, indicate blocking operations (see Section 4.1.1). As such, processing on a thread will not continue until these operations complete. When a worker thread becomes blocked by the *Read Message* operation, the core running the thread is released to allow a new actor firing to execute on it. After spawning an initial set of worker threads at setup-time, the control thread enters the loop shown in Fig. 4.1(a).

Worker threads operate based on the loop shown in Fig. 4.1(b).

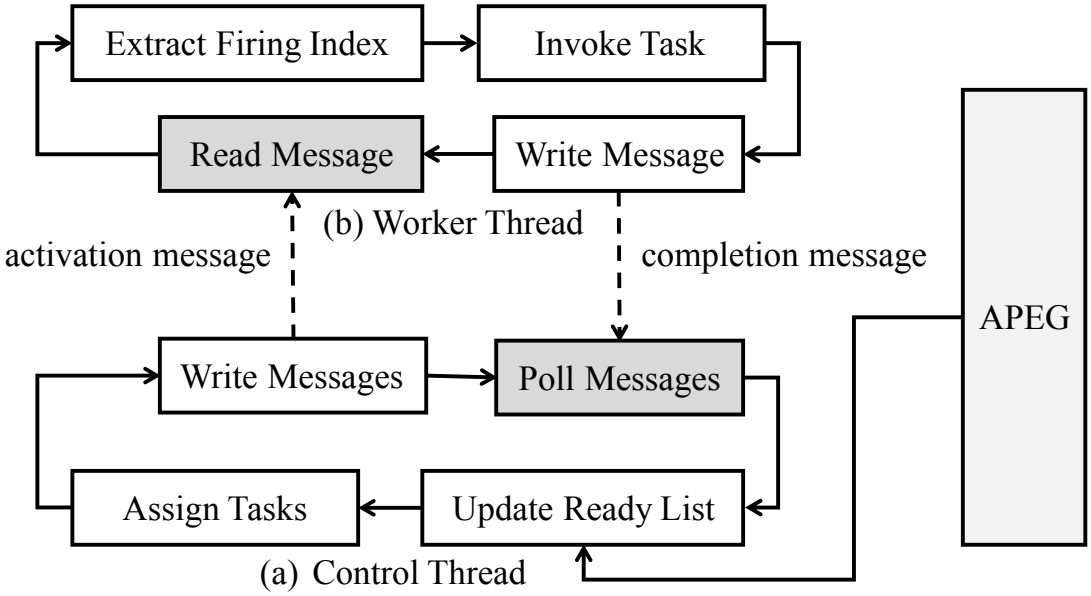


Figure 4.1: Operation of threads under the HMBE scheduler.

4.1.4 Limitations

A limitation of the current HMBE framework is that the default setting $N_T(A) = N_C$ is in general inefficient for actors that provide limited amounts of data parallelism, such as actors whose firings must be serialized because they maintain internal state. This limitation does not affect the experiments that we describe in Section 4.2.1 because the actors in these case studies provide relatively large amounts of data parallelism. Extending the HMBE scheduler to optimize the default configuration $N_T(A)$ of an actor A based on analysis of G_{apeg} is a useful direction for future work.

4.2 Experiments

4.2.1 Image Stitching

In this section, we evaluate the proposed HMBE framework with an image stitching application for large scale microscopy images [1]. Modern microscopes use a motorized stage to image a plate that is larger than a microscope's field of view and acquire a grid of overlapping images ("tiles"). Image stitching software then integrates these tiles into a single image mosaic for the plate under study. This process iterates over adjacent pairs of tiles and applies the dataflow graph shown in Fig. 4.2 to each adjacent pair.

4.2.1.1 Application Model

Fig. 4.2 shows the workflow for processing a given pair of adjacent tiles in a dataflow style. It first computes the Fast Fourier Transform (FFT) of each tile and then applies the Phase Correlation Image Alignment Method (PCIAM) [41] to the results of these FFT computations. The "CCF" block in the figure computes the Cross Correlation Factors (CCFs) of the PCIAM result, and extracts the maximum intensity point from the derived CCFs.

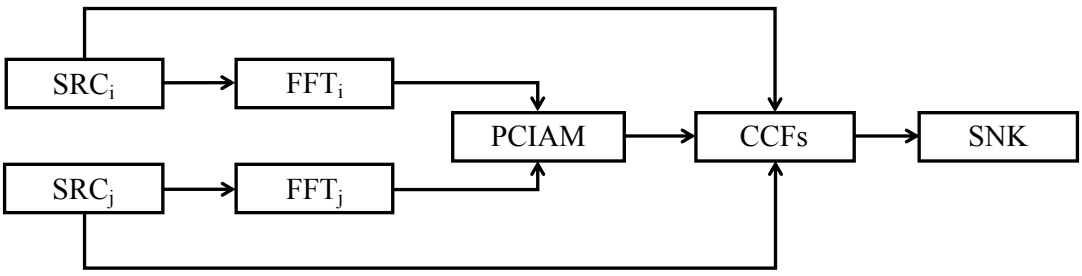


Figure 4.2: Workflow for processing an adjacent pair of tiles.

Fig. 4.3 depicts the WSDF model that we use to develop the image stitching application in HMBE. This model has two separate branches for processing tile pairs that are horizontally and vertically adjacent. The parameters $\{c_i\}$ and $\{d_i\}$ shown in the figure are inherited from WSDF. Intuitively, each c_i represents the dimensions of a sliding window associated with the communication of tokens across the given edge and each d_i represents how the sliding window associated with an edge is translated from one window instance to the next in the associated image grid. We refer to the parameters c_i and d_i as the WSDF *window size* and *window stride* parameters, respectively. For more details on the window size and window stride parameters, we refer the reader to Keinert et al. [7].

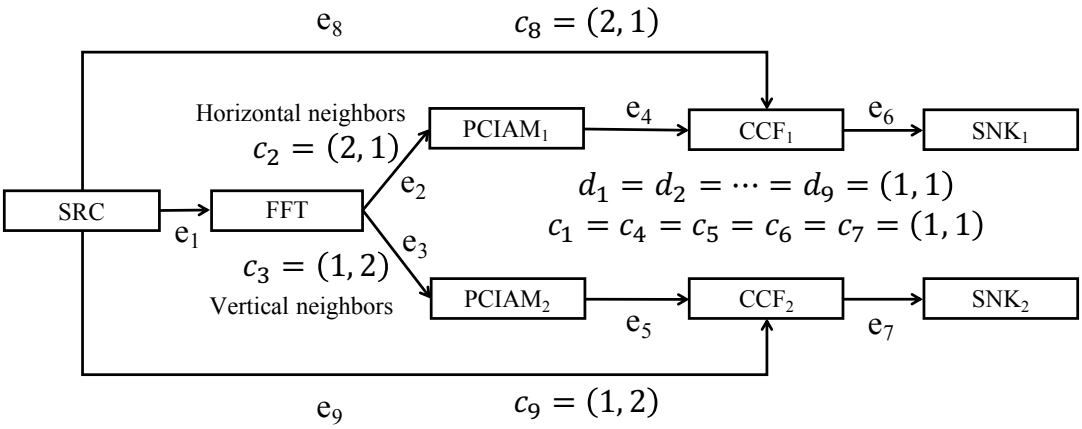


Figure 4.3: WSDF model of the image stitching application.

For our image stitching application, efficient memory management is critical. Each image tile is approximately 3 MB in size and its corresponding FFT result is nearly 12 MB in size. Thus, the total memory requirement of each image tile is about 15 MB. The input dataset, obtained from a real imaging microscope, is a 42×59 grid of 2478 tiles. If memory is not released while processing tiles, then the tiles and their transforms occupy nearly 37 GB of memory; this exceeds the capacity of many commercial platforms and

certainly most embedded signal processing devices. Indeed, one of the difficulties in using HTGS is configuring the appropriate rules in the task graph for releasing memory [1].

In contrast, HMBE uses a systematic and automated approach for memory management. This is achieved through the APEG-driven, ready list scheduling approach along with the ICP and SAD selection criteria defined in Section 4.1. Additionally, the configurability of the HMBE framework allows the designer to efficiently experiment with alternative selection criteria. For example, such experimentation can allow the designer to configure trade-offs between execution time and memory consumption that are most closely suited to the constraints of the given application and platform. Further study of alternative metrics to guide ready list scheduling in HMBE is a useful direction for future work.

4.2.1.2 Results

To demonstrate the scalability of the HMBE framework, we experimented with five platforms ranging from a high-end computing configuration to an embedded configuration based on a Raspberry Pi device. These platforms are summarized in Table 4.1. On all of the platforms, we used pthreads to implement threads through the interface provided by the C++ standard library.

We used the implementation generated by the HMBE framework to process a 42×59 grid of image tiles. For comparison, we applied the same dataset to a sequential implementation of image stitching and two different configurations of HTGS.

The two configurations of HTGS differ in that one activates the *memory pool* option

Platform	CPU	Cores	RAM	Read (MB/s)
High-end w/s	2x 2.4 GHz Intel E5-2640v4	40	512 GB DDR4	518.56
Medium w/s	3.4 GHz Intel i7-2600K	8	16 GB DDR3	116.31
MacBook Pro	2.7 GHz Intel i5-5257U	4	8 GB DDR3	1335.93
Old Desktop	2.8 GHz Intel Pentium D	2	2 GB DDR1	68.52
Raspberry Pi3 B	1.2 GHz ARMv7 rev 4	4	1 GB LPDDR2	21.37

Table 4.1: Configurations of the platforms used in our experiments.

of HTGS, while the other does not use this option. In this context, a memory pool is a fixed-size block of pre-allocated memory that HTGS manages internally rather than using external libraries for dynamically allocating memory on a system heap. The memory pool option in HTGS is designed to efficiently prevent fragmentation in memory and reduce the number of memory allocation operations that need to be carried out at runtime. In contrast to HTGS, HMBE does not provide a memory pool option. Integration of a memory pool into HMBE is a useful direction for future work.

Experimental results on execution time and peak memory usage are summarized in Table 4.2. The first and second columns in the table show the execution time in seconds and the peak memory usage in MB. For HTGS with the memory pool option disabled, we were only able to measure runtime (15.06 s) on the high-end workstation. This is because on the other platform configurations, the application could not execute to completion due to exhaustion of memory resources. The execution times and peak memory usages reported in Table 4.2 are averaged over 10 runs each with the 10th and 90th percentiles removed.

The results summarized in Table 4.2 demonstrate that for the Intel-based configu-

Platform	Sequential Implementation	HMBE	HTGS w/100 Tile Memory Pool
High-end w/s	182.40 s, 411 MB	15.03 s, 1234 MB	13.65 s, 1043 MB
Medium w/s	586.07 s, 412 MB	51.84 s, 756 MB	50.23 s, 960 MB
MacBook Pro	256.75 s, 380 MB	108.57 s, 544 MB	104.06 s, 896 MB
Old Desktop	863.60 s, 410 MB	406.30 s, 571 MB	398.28 s, 870 MB
Raspberry Pi3 B	2789.1 s, 378 MB	1025.4 s, 568 MB	1679.2 s, 857 MB

Table 4.2: Average execution time and peak memory usage comparison.

rations, HMBE achieves processing speeds that are close to that of HTGS while (a) providing a more systematic approach for exploiting parallelism and (b) automating complex steps that are needed for effectively exploiting such parallelism. In particular, generating and applying the APEG is fully automated within HMBE. Additionally, HMBE achieves peak memory usage levels that are sometimes significantly better than those measured for HTGS. HMBE also achieves significantly better execution time performance compared to HTGS on the Raspberry Pi platform. We expect that this is due to the limited amount of memory available on the Raspberry Pi, the larger memory utilization of HTGS compared to HMBE, and the resulting performance penalty due to HTGS's use of virtual memory.

After reducing the value of the memory pool size parameter in HTGS on the Raspberry Pi Platform, HTGS performed with an execution time that is approximately 300 seconds faster than the result shown in Table 4.2. This performance improvement highlights the need for fine tuning of parameters in HTGS in order to obtain performance. In contrast, HMBE obviates the need for such tuning.

4.2.2 Background Subtraction

In this section, we evaluate the HMBE framework on an application for the background subtraction of multispectral video streams [42]. This application significantly differs from the image stitching application (see Section 4.2.1) and helps to demonstrate the generality of the models and methods introduced in this thesis. Unlike image stitching, which involves complex data dependencies and abundant task-level parallelism, the background subtraction application has simpler data dependencies and involves a significant amount of sequential processing between tasks.

Multispectral imaging uses up to 10 distinct spectral bands and, as such, provides increased spectral discrimination compared to conventional imaging methods. For additional background on multispectral imaging, we refer the reader to Coffey [43]. The specific multispectral image processing application that we investigate in this section is the background subtraction application presented in LDspectral [42], a tool for multispectral image processing that is built on top of the Lightweight Dataflow (LD) environment [30].

In our experiments, we ported a background subtraction application developed as part of LDspectral to the HMBE framework and evaluated the efficiency of the new implementation. Based on the original application design in LDspectral, the process of background subtraction on a given multispectral image (video stream frame) involves performing background subtraction for each band separately and combining the results in a pixel-by-pixel fashion. More specifically, the output value $v(p)$ for a given pixel p is derived by taking a weighted sum of the corresponding pixel values $b_1(p), b_2(p), \ldots, b_m(p)$, where each $b_i(p)$ is the result for pixel p of the background subtraction process that is ap-

plied to band i of the input image. For details on how the weights for this combination operation are derived, we refer the reader to Li et al. [42].

In our experiments, we applied the multispectral video dataset published by Benezeth et al. [44]. Each video frame in this dataset has seven bands. Background subtraction accuracy is assessed by comparing the results compared to ground truth results that are provided as part of the dataset.

4.2.2.1 Application Model

Fig. 4.4 shows the application model of the background subtraction application for multispectral video streams developed in LDspectral [42]. The application model is designed to execute multispectral background subtraction over a sequence of image frames, compute statistics on background subtraction accuracy, output the accuracy results, and also output other diagnostic information to aid in understanding the performance of the overall system design. Thus, in the context of our experiments, the "application" applies background subtraction to a given video sequence and evaluates the resulting accuracy compared to existing ground truth results that are available for the video sequence.

The model shown in Fig. 4.4 includes seven source actors, labeled source 1–7. Each source actor reads pixel values associated with a separate spectral band from an input file associated with the band. The source actors then produce tokens that encapsulate the pixel values read from the respective input files.

The actors labeled bgsub 1–7 perform single-band background subtraction on spectral bands 1–7, respectively. The results from the background subtraction actors are pro-

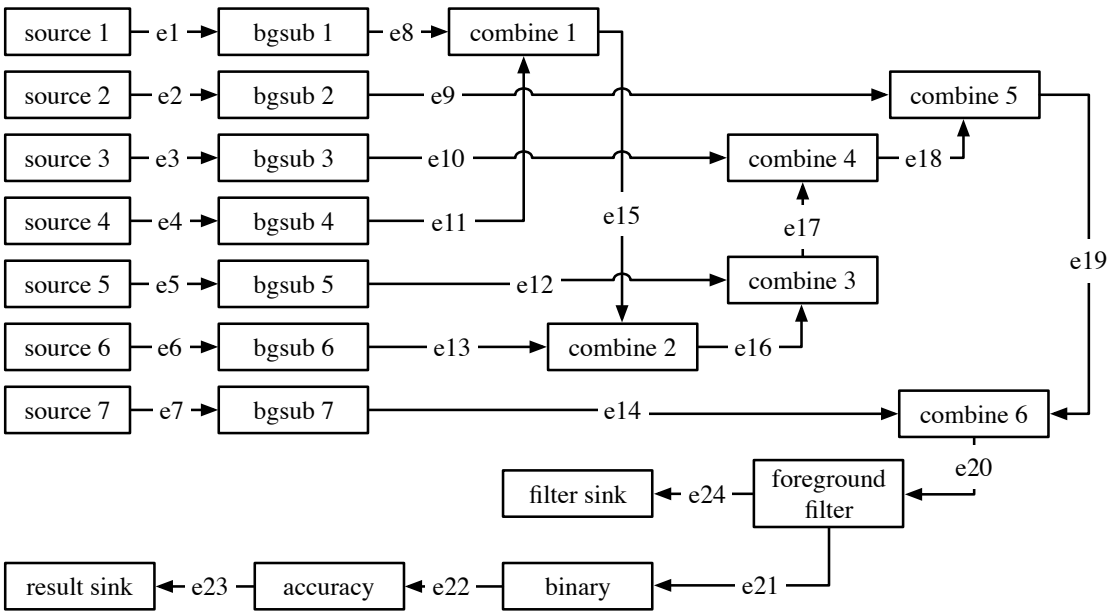


Figure 4.4: Application model of the background subtraction application for multispectral video streams.

cessed sequentially by the actors labeled combine 1–6. Each combine actor performs a weighted addition of two images using pre-defined weights that are optimized through a training process in LDspectral. The weight-combined result goes through a foreground filter, which is designed to remove noise from the combined foreground mask. Each pixel in the output of the foreground filter is a gray-scale value with higher values corresponding to higher likelihood of being a foreground pixel. The output of the foreground filter is written to disk for diagnostic purposes by the filter sink actor. It is also converted into a binary image, with each pixel labeled as foreground or background, by the binary actor; the binary actor uses a pre-defined threshold in this conversion.

The accuracy of the background subtraction result is then evaluated by comparing the result to the ground truth that corresponds to the input image. The accuracy actor has access to a database of ground truth images, as well as the order in which the input images are presented to the system (through the source actors). This actor evaluates the accuracy

of the background subtraction operation for the current image, combines this result with an accuracy value that has been averaged across all input frames processed previously and then outputs, for diagnostic purposes, the new average accuracy value while taking into account the result for the current frame. For more details on the accuracy evaluation, we refer the reader to Li et al. [42]. The accuracy actor produces a token that encapsulates a pointer to the binary image that is input to the actor. The result sink actor uses this pointer to write the image to an output file.

All the actors in Fig. 4.4 include one or more calls to functions from OpenCV, an open source library for computer vision (CV) [45]. Furthermore, some of these actors maintain internal state, which prevents different firings of the same actor from executing in parallel. Internal state can be modeled in dataflow graphs as self-loops (edges that have the same source and sink actors) [46]. For clarity, we omit these self-loop edges from the drawing in Fig. 4.4. For example, each background subtraction actor uses an internal state that is associated with the actor's use of Gaussian mixture model techniques [44]. The accuracy actor also uses state to compute summary statistics across the entire video stream on which the application is executed.

By examining Fig. 4.4, it is apparent that actors combine 1–6 must execute sequentially. This is a different form of serialization when compared to the one imposed by the presence of state in the background subtraction actors and the accuracy actor. In particular, the background subtraction and accuracy actors exhibit inter-firing serialization (different firings of the same actor must execute sequentially), whereas the combine actors exhibit inter-actor serialization (different actors must execute sequentially). These different forms of serialization constrain the exploitation of parallelism. However, we

demonstrate that using HMBE we are able to achieve significant speedup despite these constraints.

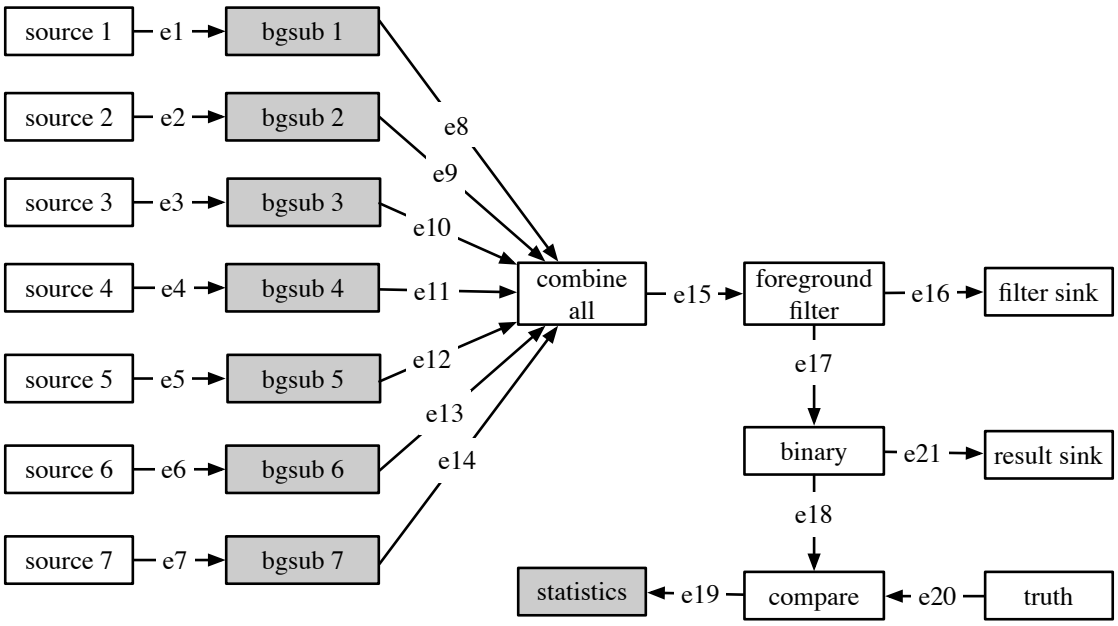


Figure 4.5: Dataflow model of the background subtraction application for multispectral video streams.

4.2.2.2 HMBE-based Model

Fig. 4.5 illustrates our modified dataflow model, which we apply as input to HMBE for our experiments. When porting the application model of Fig. 4.4 to HMBE, we modified the design so it conforms to the WSDF-based semantics of HMBE and it achieves more efficient inter-thread communication within the scheduler/worker framework of the HMBE runtime system. The modifications did not change the functionality (input/output behavior) of the application. They were made only to adapt the application for correct and efficient operation within HMBE. In the remainder of this section, we refer to the dataflow graph shown in Fig. 4.5 as the *HMBE application model*.

In the HMBE application model, the eight shaded actors of Fig. 4.5 have internal

state and, as such, exhibit inter-firing serialization. As discussed previously, this means that parallelism can be exploited across these actors for a given video frame, but cannot be exploited across different firings of the same actor (i.e., across successive frames). Furthermore, the HMBE application replaces the six combine actors from Fig. 4.4 with a single actor called combine all in order to reduce the amount of message passing between the scheduler and workers. This modification does not reduce parallelism because the combine actors execute sequentially due to their data dependencies.

The HMBE application also decomposes the accuracy actor of Fig. 4.4 into three actors: (1) the *statistics* actor maintains statistics about background subtraction accuracy based on the results produced by the compare actor; (2) the *truth* actor reads the ground truth image (corresponding to the current input image) from disk; (3) the *compare* actor compares each pixel of the binary image (produced by the binary actor) with the corresponding pixel of the ground truth image. Based on these comparisons, the compare actor determines (1) the number of pixels n_1 that are classified as foreground in the binary image, (2) the number of pixels n_2 that are foreground in the ground truth image, and (3) the number of pixels n_3 that are classified correctly in the binary image. The statistics actor then uses the values n_1, n_2, n_3 to update its record of accuracy statistics.

Decomposing the original accuracy actor into three separate actors reduces the amount of sequential processing and exposes more parallelism. Specifically, the statistics actor involves sequential operation, but is lightweight (low complexity). On the other hand, the compare and truth actors not only enable inter-firing parallelism, but also allow overlapped execution of computational tasks and disk IO tasks.

The HMBE application model is a WSDF model with one dimension and no overlap

between the sliding windows of the model. As such, the window size and stride parameters are trivially set to 1 pixel each. More precisely, for each edge e_i in the dataflow graph, the dimension vector of the sliding window is $c_i = (1,1)$, and the stride vector is $d_i = (1,1)$.

4.2.2.3 APEG Representation

A significant challenge in accelerating this application is exploiting task-level parallelism under the sequential processing constraints that are imposed by the actors that maintain an internal state across invocations. These sequential processing constraints are modeled naturally within the dataflow model through self-loop edges, as described in Section 4.2.2.1. When the APEG is constructed, these self-loop edges result in APEG edges between successive firings of the associated actor (each actor in the HMBE application model that has a self-loop edge).

Fig. 4.6 shows the APEG that is generated by HMBE for the dataflow model of Fig. 4.5 with an unfolding factor of 3. In our context, the unfolding factor gives the number of dataflow graph iterations (video frames) for which the APEG is constructed. This value of 3 is chosen as a simple (small-scale) example for illustration purposes. Larger unfolding factors for the APEG expose more inter-iteration (inter-frame) parallelism at the expense of larger design-time and runtime storage costs. This trade-off can be optimized as part of the iterative experimentation process that is supported by HMBE. Developing algorithms within HMBE to optimize the unfolding factor automatically is an interesting direction for future work.

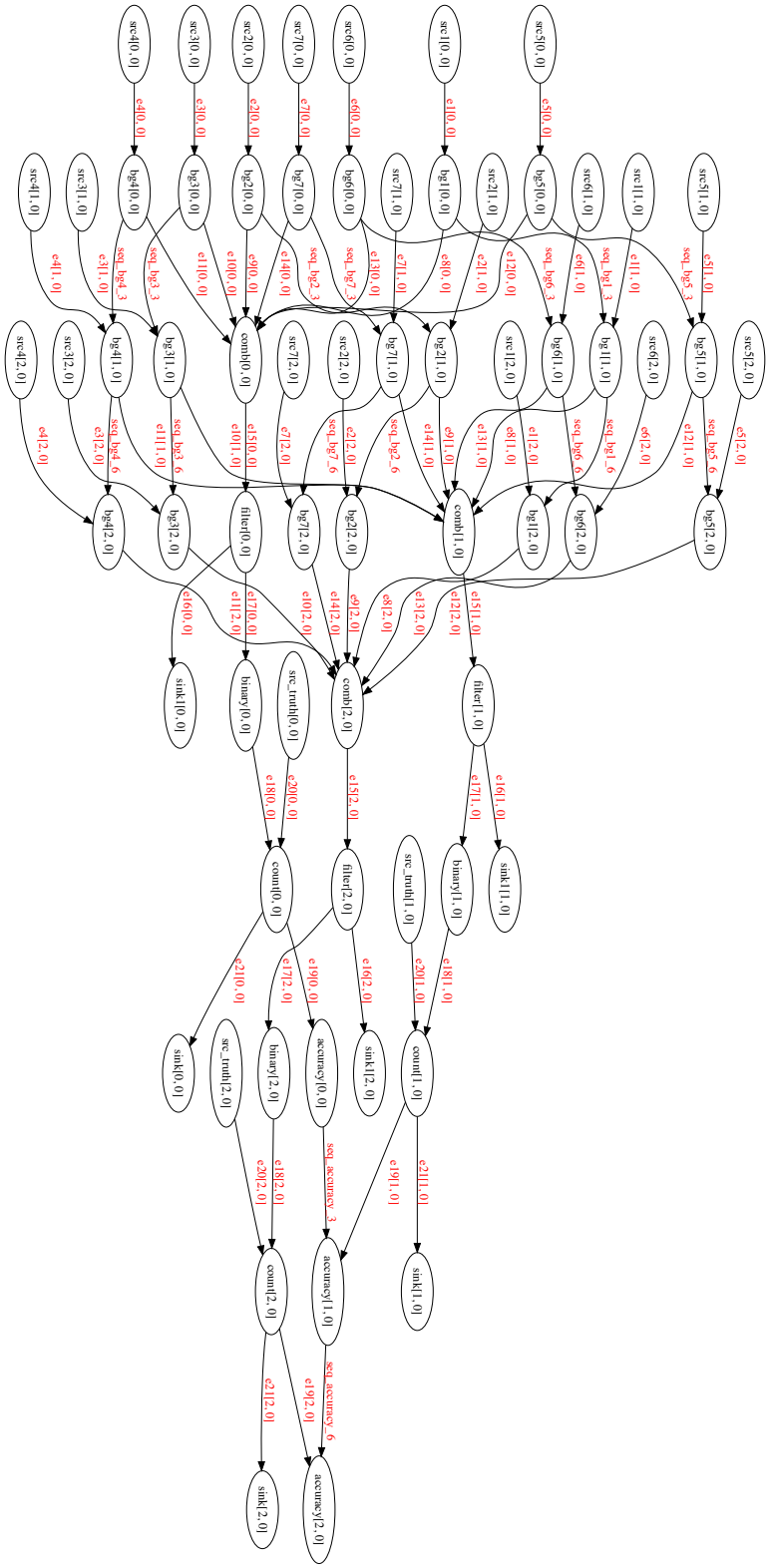


Figure 4.6: The APEG that is generated for the HMBE application model with an unfolding factor of 3.

4.2.2.4 Results

In our experiments with the Multispectral Background Subtraction (MBS) application, we used the three different computing platforms listed in Table 4.3. The first two platforms represent desktop configurations; the third is a widely used embedded computing platform.

Platform	CPU	Cores	RAM	Read (MB/s)
Medium w/s	3.4 GHz Intel i7-2600K	8	16 GB DDR3	116.31
Entry-Level Server	3.3 GHz Intel Pentium G4400	2	4 GB DDR4	200.44
Raspberry Pi3 B	1.2 GHz ARMv7 rev 4	4	1 GB LPDDR2	21.37

Table 4.3: Platforms used in our experiments on the MBS application.

The input data for our MBS experiments is a sequence of 172 multispectral video frames obtained from the dataset of Benezeth et al. [44]. These frames are stored in (172×7) distinct files with a separate file per spectral band for each frame. Each source actor, source 1–7, reads the video frames associated with its corresponding band. This experiment compares three implementations for each of the platforms of Table 4.3: *Sequential*, *Intra-Firing*, and *HMBE*. These three implementations use the same OpenCV kernels, but have different parallelization and scheduling strategies. The OpenCV library supports intra-firing parallelism by enabling multithreading within kernels.

1. *Sequential* sets all OpenCV kernels to single-thread execution. It uses a static scheduler based on the LIDE (LIghtweight Dataflow Environment) package [30].

2. *Intra-Firing* invokes actors sequentially using the same static scheduler as *Sequential*, but exploits the intra-firing parallelism within actors by enabling multithread-

ing in OpenCV kernels.

3. *HMBE* exploits both inter-actor and inter-firing parallelism. It explicitly sets the OpenCV kernels to single-threaded execution to avoid any interference with scheduling decisions made by the HMBE scheduler.

Table 4.4 summarizes the execution time and memory usage of each implementation on the three platforms.

	Sequential		Intra-firing Parallelism		HMBE	
Platform	Time (s)	RAM (MB)	Time (s)	RAM (MB)	Time (s)	RAM (MB)
Medium w/s	11.50 ± 0.14	143	5.22 ± 0.07	144	3.32 ± 0.02	266
Entry-Level Server	9.55 ± 0.01	146	6.80 ± 0.05	148	5.29 ± 0.02	187
Raspberry Pi3 B	117.61 ± 1.18	143	85.64 ± 0.10	142	34.11 ± 1.49	221

Table 4.4: Execution time & standard deviation, and peak memory usage comparison.

The results above show that, on the eight-core workstation, *HMBE* is nearly 70% and 36% faster than *Sequential* and *Intra-Firing*, respectively. *HMBE* also keeps the peak memory usage at quite a low level.

This experiment clearly demonstrates the ability of HMBE to exploit coarse grain parallelism (inter-actor and inter-firing parallelism) for a complex application graph. It also allows us to observe that intra-firing parallelism, embodied by the *Intra-Firing* implementation, cannot produce enough parallelism to saturate all CPU cores. In addition to limiting parallelism, frequently spawning and joining threads (from inside individual firings) poses considerable overhead to the runtime system as well.

More generally, this experiment highlights (1) the importance of considering paral-

lelism within an application as a whole instead of just within compute kernels in order to maximize performance and (2) the effectiveness of using HMBE to exploit precisely this whole-application parallelism.

4.3 Summary

In this chapter, we have introduced the HTGS Model-Based Engine (HMBE) for design optimization of multicore signal processing systems. HMBE incorporates the windowed synchronous dataflow (WSDF) model of computation and novel dynamic scheduling techniques that map application tasks onto threads using an expanded representation of the WSDF model to guide the mapping process. We have demonstrated the utility of HMBE in deriving efficient implementations across a diverse range of platforms and for two significantly different practical applications. Several useful directions for future work have been motivated from this first-version development of HMBE, including runtime configuration of the worker thread capacity, more extensive investigation of alternative ready-list selection criteria, integration of a memory pool into HMBE, optimization of unfolding factors within HMBE, and extension of HMBE to work with other forms of dataflow in addition to WSDF.

Chapter 5: Design Flow

In this chapter, we develop a design methodology and associated design tool support for systematically integrating the model-based methodology of HMBE into the HTGS runtime system. A key component of the tool is a new approach to hierarchical dataflow scheduling that integrates a global scheduler and multiple local schedulers. The local schedulers are lightweight modules that work independently. The global scheduler interacts with the local schedulers to minimize overall memory usage and execution time. The proposed design tool, called HMBE-Integrated-HTGS (HI-HTGS), is demonstrated through a case study involving an image stitching application for large-scale microscopy images.

Material in this chapter was published in partial, preliminary form in Wu et al. [47].

5.1 Related Work

HI-HTGS places a special emphasis on high-performance multicore implementation, and integrated optimization of memory management and task scheduling using a single actor, dynamic invocation (SADI) scheduling model [9]. HI-HTGS combines the abstract dataflow graph analysis features of DIF with the APIs of HTGS, which enable construction, integration, and iterative optimization of high-performance software com-

ponents and task graphs. While DIF focuses on high level dataflow analysis in which the detailed functionality of individual graph components (actors) is abstracted, HTGS provides extensive infrastructure for creating fully functional, high-performance task graph implementations. Thus, the features of DIF and HTGS are highly complementary, and their integration through HI-HTGS provides new capabilities for automated, model-based analysis, implementation, and optimization of multicore signal and information processing systems.

Like HMBE, HI-HTGS applies an adapted version of Keinert's windowed synchronous dataflow (WSDF) model of computation [7]. WSDF is well suited as a model of computation for smart vision system design due to its provisions for representing image processing functionality. In WSDF, dataflow tokens can be placed in correspondence with multidimensional windows of data. These windows are restricted in WSDF to have constant dimensions. The degree of overlap between the windows can be configured through the *sampling matrices* parameter.

We incorporate minor adaptations to WSDF in the model of computation that we use in HI-HTGS. First, since the sampling matrix is a diagonal matrix, its representation can be simplified to be a vector. We apply this simplification in the adapted form of WSDF that we apply. Also, our adapted form of WSDF eliminates use of the *effective token size* parameter; instead, we assume that the basic unit of token production is a single token (as in conventional forms of dataflow).

We employ WSDF semantics in HI-HTGS due to the orientation of HI-HTGS toward high-performance image processing and other areas of multidimensional signal processing. However, the scheduling techniques presented in this proposal can read-

ily be adapted to other forms of dataflow, such as synchronous dataflow, cyclo-static dataflow, and multidimensional synchronous dataflow, from which a certain type of expanded dataflow representation, called an acyclic precedence expansion graph (APEG) (or simply "acyclic precedence graph") [48], can be derived. In Section 5.2, we discuss the APEG representation in more detail.

5.2 Workflow

HI-HTGS is a design tool that enables high-performance image processing on multicore platforms. The tool is composed of two main parts: (a) the DIF-based analysis engine (*analysis engine*), which performs compile-time analysis for the application dataflow graph, and (b) the HTGS-based runtime system (*runtime system*), which applies the analysis engine and provides optimized multithreaded execution and memory management for the application. Fig. 5.1 illustrates the workflow associated with HI-HTGS.

To transfer dataflow graph analysis results from the analysis engine to the HTGS-based runtime system, and allow the HTGS-based runtime system to apply these results efficiently, we have developed a compact data exchange protocol using the Google Protobuf library [37]. Our protocol, called the *HI-HTGS exchange protocol*, ensures that the analysis engine and HTGS-based runtime system communicate reliably and efficiently even though they are developed in different languages (Java and C++, respectively). Due to the more compact binary format of the HI-HTGS exchange protocol compared to the XML or JSON formats, data can be exported and imported with lower runtime overhead and with smaller storage cost. These features are important in HI-HTGS since a relatively

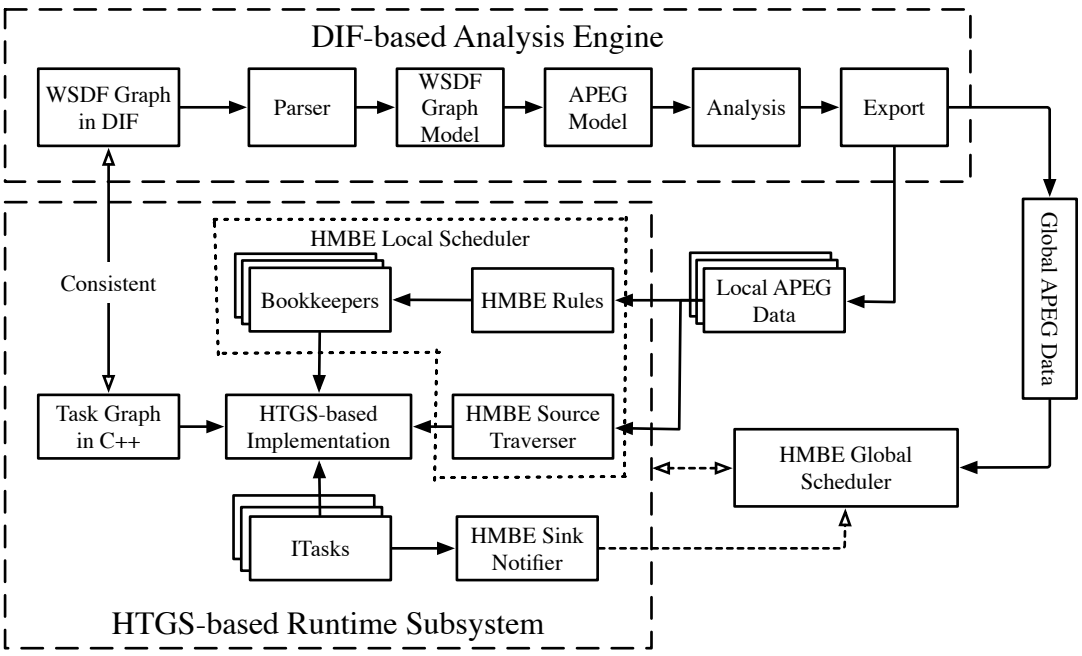


Figure 5.1: Workflow associated with HI-HTGS.

large amount of data is exchanged between the two parts of the tool, including the APEG representations that are constructed by the analysis engine.

As described in Section 5.1, HI-HTGS incorporates an adapted form of WSDF as the system designer's entry point to the tool. The WSDF-based application model (*application graph*) is written in the DIF Language, as shown in the top left corner of Fig. 5.1. Given a textual representation of the WSDF model in DIF, the analysis engine parses the representation and produces a graphical internal representation within the analysis engine. This internal representation is then expanded into an equivalent APEG representation using minor adaptations to WSDF of standard techniques for multirate to single rate synchronous dataflow (SDF) conversion [48].

To ensure functional correctness, the WSDF graph representation must be kept consistent with the C++ task graph representation that is developed by the designer in HTGS. This consistency requirement is represented in Fig. 5.1 by the two arrows labeled "consis-

tent". Consistency between the two parallel representations can be assessed as an integral part of the design process through component and subsystem-level testing that verifies functional equivalence. Well-known processes for automated execution of test suites can allow such testing to be integrated efficiently into the design process. Various authors have discussed this matter; we cite the work of Hamill as representative of this body of work [49].

Intuitively, HTGS and DIF provide complementary programming formats for applications in HI-HTGS. Software components in HTGS, called *tasks*, specify detailed intra-task functionality that is integrated with mechanisms for high-performance task execution. On the other hand, the WSDF actors specified in DIF expose formal properties associated with interactions among tasks. In general, the mapping between HTGS tasks and WSDF actors can be one-to-many, one-to-one, or many-to-one depending on the application and the designer's approach to model-based representation. The programming approach in HI-HTGS shifts programmer effort to that of task programming and model-based design (WSDF-based design in the current version of HI-HTGS), and alleviates the burden of inter-task scheduling and memory management. The result is a more productive and reliable form of design through the higher level of abstraction provided by the coupled task- and actor-based specification format.

The APEG represents the firings within a single iteration of a periodic schedule for the WSDF-based application graph. Each vertex v in the APEG has an associated actor $actor(v)$ in the application graph, and represents a distinct firing of $actor(v)$ within a periodic schedule. Thus, each actor in the WSDF graph is mapped to a set of one or more vertices in the APEG. The APEG is useful as an internal scheduling representation in part

because it exposes parallelism and inter-actor communication at the level of application firings [4]. HI-HTGS uses only APEG information without reference to any specific periodic schedule. Instead, the schedule evolves dynamically under the control of the runtime system.

After construction of the APEG, the analysis engine performs various forms of analysis that are used to annotate the APEG vertices. These annotations are used by the runtime system when the application executes. After finishing its compile-time analysis on the APEG, the analysis engine attaches its analysis results as attributes of the APEG vertices, and exports the annotated APEG as binary data using the HI-HTGS exchange protocol. This binary data is then read by the runtime system at the beginning of execution when the application graph is launched.

HI-HTGS augments HTGS with a novel hierarchical scheduling approach to bridge the semantic mismatch between the original HTGS framework and the HMBE scheduler. In our context, the problem of bridging the semantic mismatch is related in large part to the concept of task graph *bookkeepers* in HTGS [1]. Bookkeepers are task graph components that are specialized to maintain state. They are especially useful for efficiently managing communication among tasks that have complicated data dependency relationships.

While HTGS bookkeepers provide a modular abstraction for specifying powerful methods to optimize memory management and interprocessor communication, the design of bookkeepers is in general a complex task requiring considerable expertise, and effort. The hierarchical scheduling approach developed within HI-HTGS includes features to automate the design, implementation, and integration of HTGS bookkeepers.

To enable more powerful global optimization in HI-HTGS, we have developed a hierarchical dataflow scheduling approach that integrates a global scheduler and multiple local schedulers. The HMBE global scheduler maintains the global execution state of the underlying dataflow model. It retrieves information from the local schedulers periodically and uses this information to dynamically configure the execution priorities of the actors with the objective of optimizing thread contention.

In summary, HI-HTGS allows the system designer to work at a high level of abstraction and automates challenging and time consuming tasks that are required to optimize use of HTGS. Additionally, the models and methods employed in the analysis engine for HI-HTGS are based on WSDF semantics and APEG representations. They can be adapted readily to other design processes and tools that are compatible with these models.

5.3 Hierarchical Scheduling

In this section, we present the hierarchical scheduling approach that is employed in HI-HTGS. In this development, we refer to the input dataflow model as the *application graph* $G_m = (V_m, E_m)$, where V_m is the set of actors and E_m is the set of edges. We denote the APEG representation of G_m as $G_a = (V_a, E_a)$. As described in Section 5.2, each vertex $v_a \in V_a$ is expanded from an actor $actor(v_a) \in V_m$ of the application graph. We refer to vertex $v_a \in V_a$ as an *invocation* of its associated application graph actor $actor(v_a)$. At compile time, the analysis engine automatically derives G_a from G_m, derives information for optimized scheduling from G_a, and then exports this scheduling information along with the APEG to the runtime system using the HI-HTGS exchange protocol.

5.3.1 Local Schedulers for Bookkeepers

Hierarchical scheduling in HI-HTGS is designed to automate and systematically integrate two aspects of HTGS-based implementations that are especially time consuming and error prone. The first aspect is related to a specific type of HTGS task graph component called a *bookkeeper*. HTGS bookkeepers help to resolve data dependencies and coordinate buffer management between communicating tasks [1]. Each bookkeeper b is associated with a set $Q(b)$ of HTGS tasks that communicate with one another. A bookkeeper b is designed by specifying *rules* that coordinate the production and consumption of data by the tasks in $Q(b)$. While bookkeepers enable separation of concerns between efficient memory management and task implementation, the design of bookkeepers is time consuming and requires significant programmer expertise.

In HI-HTGS, the implementation of bookkeepers is automated through the use of relevant information from the APEG. In particular, each bookkeeper is implemented in HI-HTGS by a *local scheduler*. An HI-HTGS local scheduler is a scheduling subsystem within the runtime system that uses local APEG information — that is, graph information that is related only to tasks in $Q(b)$. A local scheduler associated with bookkeeper implementation uses this graph information to dynamically manage data production and consumption, and optimize memory management associated with communication among tasks in $Q(b)$.

5.3.2 Local Schedulers for Source Actors

The second aspect of HTGS-based implementations that is automated in HI-HTGS through hierarchical scheduling is the process of determining how input data in multidimensional data streams is traversed to supply input to sequences of task executions. The traversal order can have a significant impact on performance. The enforcement of efficient traversal orders is performed automatically by local schedulers that are associated with source actors in the WSDF model (actors that model the interface between the system inputs and the task graph). These local schedulers are called *source traversers*. The source traversers developed in the current version of HI-HTGS assume that dataflow graph input is read from files.

In HI-HTGS, the problem of optimizing the traversal order for a source actor σ is modeled as the problem of determining the order in which the APEG vertices associated with σ are executed. This *APEG vertex execution order* (*AVEO*) is determined statically in HI-HTGS and enforced at runtime through information that is exported from the analysis engine to the runtime system.

The AVEO is computed using the undirected version U of the APEG — that is, the undirected graph that results from ignoring the direction of the edges in the APEG. The computation starts by deriving a starting vertex v_s in U that is associated with σ. The starting vertex is taken as an APEG vertex that has minimum out-degree from among all vertices associated with σ. If multiple vertices have minimum out-degree, then one of these is selected arbitrarily.

From the starting vertex, shortest paths in U are computed to all other vertices

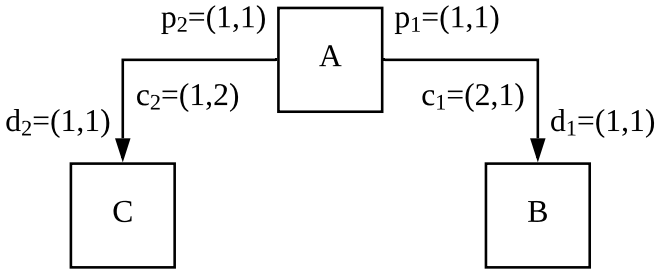
associated with σ. The path length used in this context is simply the number of vertices visited in a given path. After this process is complete, we have a path length $p(v)$ for every APEG vertex associated with σ. The vertices associated with σ are then ordered for execution by nondecreasing order of $p(v)$ with ties broken arbitrarily. The local scheduler associated with σ enforces the $p(v)$-sorted ordering at runtime.

Intuitively, all steps of our AVEO derivation heuristic are designed to minimize the lifetime of data and improve data locality as the graph is executed, which are identical to the objectives of the HTGS bookkeepers.

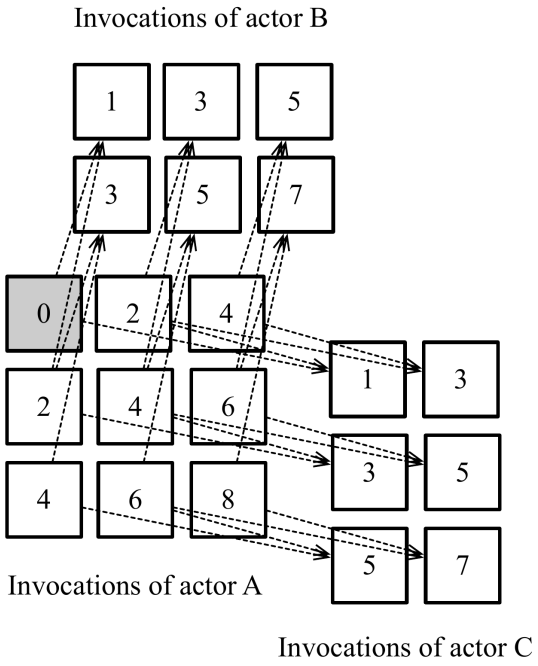
5.3.3 Source Traverser Example

Fig. 5.2 illustrates with a simple example the derivation of the execution order for a source traverser. Fig. 5.2(a) shows an example of a WSDF-based application graph that contains a source actor A. Here, each p_i represents the two-dimensional production rate associated with the ith edge, and similarly, each c_i represents the consumption rate. Each d_i specifies the movement of a sliding window on each edge from one invocation of the edge's sink actor to the next. For more details on the multidimensional production rates, consumption rates, and sliding window parameters associated with WSDF edges, we refer the reader to Keinert et al. [7].

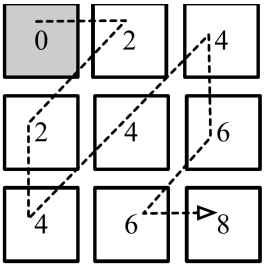
In WSDF, the APEG in general depends on the dimensions of each frame of input data on which the application graph operates. Fig. 5.2(b) shows the APEG that results from the application graph of Fig. 5.2(a) with a frame size of 3×3 tokens. Here, squares represent APEG vertices and dashed arrows correspond to APEG edges. The vertices are



(a) Application graph.



(b) APEG for the application graph.



(c) Derived execution order for the source traverser.

Figure 5.2: A simple example that illustrates derivation of the execution order for a source traverser in HI-HTGS.

laid out in the figure according to the two-dimensional index spaces associated with the corresponding WSDF actors [7]. For example, the APEG vertex in the second row and third column for actor A corresponds to invocation $(2,3)$ of the actor. The number inside each square in Fig. 5.2(b) gives the shortest path length between the corresponding APEG vertex and an invocation associated with the source actor A.

To determine the schedule for the source traverser associated with A, we sort the APEG vertices associated with A in nondecreasing order of their shortest path lengths. A schedule that results from such a sorting operation is illustrated in Fig. 5.2(c). The dashed path illustrated in the figure shows the order in which invocations are dispatched starting with invocation $(1,1)$ and ending with invocation $(3,3)$.

5.3.4 Hierarchical Scheduling Architecture

In HI-HTGS, a given application graph G_m is scheduled using a set $\lambda = L_1, L_2, \ldots, L_r$ of local schedulers, which are in one-to-one correspondence with the bookkeepers in the associated HTGS graph and the source actors in G_m. The actors V_m in G_m are partitioned into $(r+1)$ disjoint sets $\alpha(L_1), \alpha(L_2), \ldots \alpha(L_r), I$, where each $\alpha(L_i)$ represents the set of actors associated with local scheduler L_i, and I represents the (possibly empty) set of actors that are not associated with any local scheduler. In this context, we refer to I as the set of *independent actors*.

A global scheduler coordinates execution across the local schedulers along with their associated actors, and the independent actors. In our current implementation of HI-HTGS, we use a purely data-driven global scheduling strategy where actor invocations

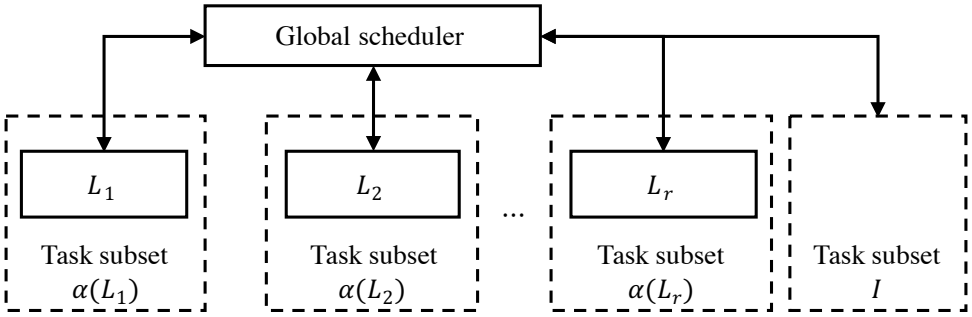


Figure 5.3: An illustration of interactions among the global scheduler, local schedulers, and independent tasks in HI-HTGS.

are dispatched for execution as soon as they have sufficient data and (if applicable) their local schedulers select them for execution. Investigation of more sophisticated global scheduling strategies in HI-HTGS is an interesting direction for future work.

Fig. 5.3 illustrates interactions among the global scheduler, local schedulers, and independent tasks in HI-HTGS. Each local scheduler L_i and each source actor in G_m is assigned a separate thread in the targeted multicore architecture. Each actor that is not a source actor is assigned N_c threads, where N_c is the number of processor cores. The total number of threads for G_m can therefore be expressed as $r+N_s+N_c \times (N_m-N_s)$, where N_m and N_s represent the total number of actors in G_m and the number of source actors in G_m, respectively. During runtime, an actor invocation executes when it has been dispatched and one of the threads allocated to the actor is available to execute the invocation.

5.3.5 Operation of Hierarchical Schedulers

At setup time, the system launches two kinds of threads: worker threads and scheduler threads. For each actor, a set of worker threads is allocated for it. For each actor that is not a source actor, the number of worker threads allocated for it is equal to the

platform-supported concurrency N_c. On the other hand, only one worker thread is allocated for each source actor. This is because source actors in HI-HTGS correspond to file-reading interfaces, and for such functionality, we anticipate that I/O access speed is the dominant concern. From our empirical analysis, using multiple threads for source actors typically does not help to improve performance.

Fig. 5.4 illustrates the operation of the hierarchical scheduler employed in HI-HTGS. To keep the scheduler lightweight and avoid busy waiting, the local schedulers and the global scheduler are all designed to have two states each — a *dormant* state, and an *active* state. The shaded boxes in Fig. 5.4 indicate blocking operations.

Initially, all local schedulers are in the dormant state as there is no data waiting to be processed. They are blocked at the component within Fig. 5.4 that is labeled Blocked Reading Token. When input data becomes available to an actor from a predecessor task, the corresponding local scheduler becomes active. At this point, the local scheduler attempts to read a control message from the global scheduler in a nonblocking manner. That is, the local scheduler processes an incoming control message if one is available; otherwise, if no control message is available at the current time, the local scheduler continues its operation.

The local scheduler then performs the following steps.

1. Sends a completion message to the global scheduler to indicate the completion of the predecessor task that produced the newly-read token.

2. Updates the execution state in the local APEG associated with A, where A is the actor associated with the local scheduler.

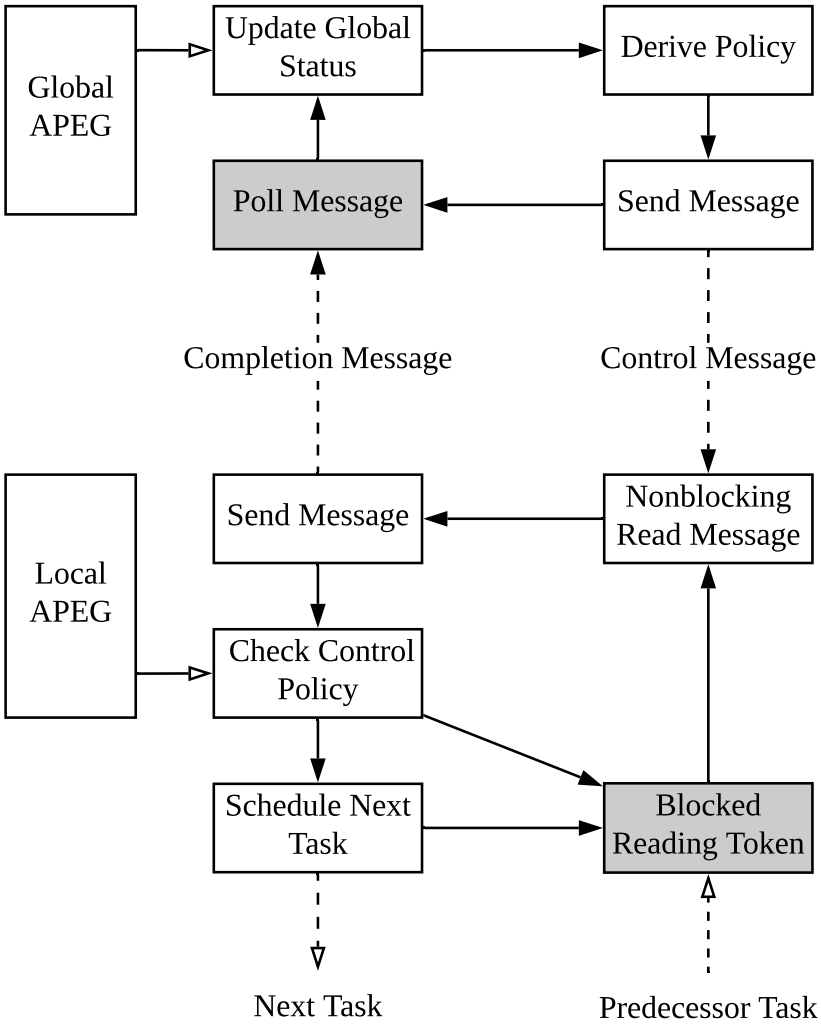


Figure 5.4: Operation of the hierarchical scheduler employed in HI-HTGS.

3. Determines whether the next firing of A has sufficient data based on the newly-received input data.

4. Appends this next firing at the end of a queue, called the *pending firing queue* of the local scheduler.

Depending on the control policy in the local scheduler, the scheduler may then hold all task references in the queue or dequeue some subset of the task references and dispatch the associated tasks.

The global scheduler is blocked by the poll message operation. The scheduler becomes active when a completion message is sent to it from one of the local schedulers. Upon becoming active, the global scheduler updates the global state of the global APEG and sends one or more new control messages if necessary. The control messages are used to change the control policies of the local schedulers, thereby providing a lightweight means for dynamic reconfiguration of scheduling strategies.

The hierarchical scheduler maintains a buffer in each local scheduler to store tokens until a sufficient set of tokens has accumulated to enable the next actor firing associated with the scheduler. The local scheduler associated with an actor A is equipped with one or more counters that keep track of the numbers of tokens that have been received from the predecessor(s) of A, but that have not yet been consumed by A. Using these counters, along with the firing rules of the actor, the local scheduler can determine (in Step 3 above) whether or not the next firing of A has become enabled.

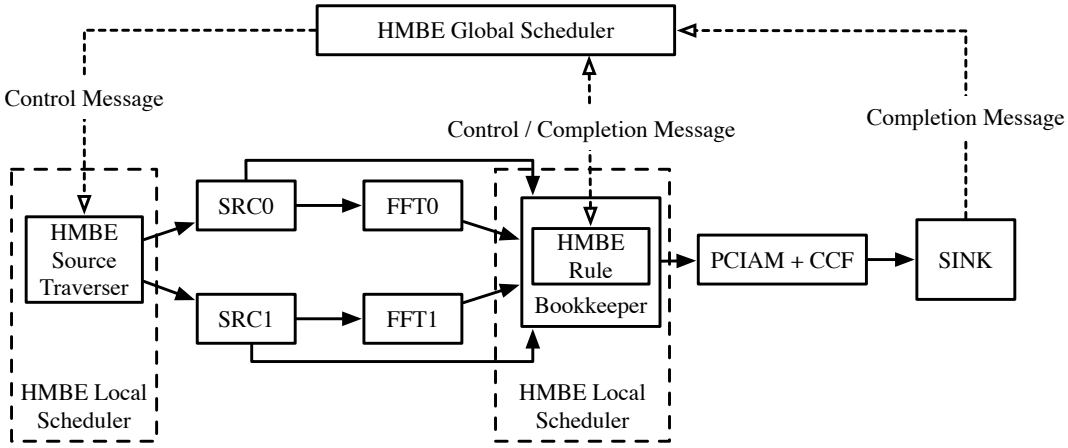


Figure 5.5: WSDF model of the image stitching application.

5.4 Case Study

In this section, we experimentally study the performance of HI-HTGS on the same image stitching application for large-scale microscopy that was presented in Section 4.2.1.

The HTGS task graph that we use to implement the image stitching application is shown in Fig. 4.2. The task graph iteratively loads adjacent pairs of tiles, computes their fast Fourier transforms (FFTs), applies the *phase correlation image alignment method* (*PCIAM*) [41] to the FFT results, and then extracts translation parameters from the results of the PCIAM operations.

This application is data intensive, as we have explained in Section 4.2.1. Due to this data intensive nature, if intermediate results are not released in a timely manner, the application can quickly run out of physical memory on the targeted computing platform.

To implement the image stitching application using HI-HTGS, we model the application as a WSDF graph and specify this graphical model using the DIF Language. Our WSDF model of the application is illustrated in Fig. 5.5.

We experimented with the HTGS- and HI-HTGS-based implementations of the im-

age stitching application on three different platforms, including a mid-range workstation, a low cost laptop computer and an outdated desktop computer. These platforms, summarized in Table 5.1, are selected to help demonstrate the scalability of HI-HTGS, as well as the applicability of HI-HTGS to diverse targets.

Platform	CPU	Cores	RAM	Read (MB/s)
Medium w/s	3.4 GHz Intel i7-2600K	8	16 GB DDR3	116.31
MacBook Pro	2.7 GHz Intel i5-5257U	4	8 GB DDR3	1335.93
Old Desktop	2.8 GHz Intel Pentium D	2	2 GB DDR1	68.52

Table 5.1: The platforms that we used in our experiments.

Our experimental results involving execution time comparison are summarized in Table 5.2. Here, we compare the measured execution times among our HTGS-, HMBE-, and HI-HTGS-based implementations of the image stitching application. The HMBE-based implementation is derived by automatically generating the APEG using a C++ extension of HTGS, and manually integrating the APEG with the HMBE scheduler, which is external to HTGS. Like the process of configuring bookkeepers in HTGS, this process of manually integrating the APEG is time consuming and error prone. Both of these tasks, bookkeeper configuration and APEG-to-scheduler integration, are fully automated in HI-HTGS. Thus, HI-HTGS greatly simplifies and accelerates the process of deriving implementations that are equipped with the powerful performance optimization features of HTGS. From the results shown in Table 5.2, we see that these improvements are delivered while maintaining performance levels that are similar to HMBE and HTGS.

To help assess the improvement in software development and maintenance cost provided by HI-HTGS, we compared the source code size of our HI-HTGS based imple-

Platform	Sequential Implementation	HMBE	HTGS	HI-HTGS
Medium w/s	586.07 s	50.51 s	49.01 s	48.94 s
MacBook Pro	256.75 s	108.57 s	104.06 s	103.86 s
Old Desktop	863.60 s	406.30 s	398.28 s	397.33 s

Table 5.2: Comparison of execution time (mean over 10 runs).

mentation of image stitching with that of the original HTGS based implementation. We measured the source code size as the lines of code in all relevant source files. In these measurements, we considered all of the application-specific source code used for application setup, scheduling, and memory management—that is, all source code excluding code that is part of the general HTGS and HI-HTGS frameworks, and outside of the task (kernel) implementations, which can be reused across different applications. The results of these measurements are summarized in Table 5.3, which shows a reduction in source code size of 63 % achieved by HI-HTGS.

Tool	C++ code	DIF code	Total code	Improvement
HTGS	909	0	909	—
HI-HTGS	273	64	337	63 %

Table 5.3: Comparison of source code size.

5.5 Summary

In this chapter, we have presented the software tool, HI-HTGS, for design and implementation of multicore image processing systems. This tool consists of two main parts: the DIF-based analysis engine, which applies the Dataflow Interchange Format (DIF) Package, and the HTGS-based runtime system, which builds on the Hybrid Task Graph

Scheduler (HTGS). The tool allows system designers to incorporate powerful techniques for performance optimization and memory management while specifying applications at a high level of abstraction and using significant levels of automation. Our experiments demonstrate the ability of our new design tool to provide this high level of abstraction and automation while generating efficient implementations on a diverse set of platforms.

Chapter 6: CPU-GPU Model-Based Engine

Graphics processing units (GPUs) are of increasing interest in a wide variety of high performance image processing applications. Processing large images in real-time requires effective image processing algorithms as well as efficient software design and implementation to take full advantage of all CPU cores and GPU resources on state of the art CPU/GPU platforms. In this chapter, we extend the capability of HMBE to heterogeneous platforms and develop a new runtime environment based on the executor pattern. This work results in a third model-based design tool, the CPU-GPU Model-Based Engine (CGMBE), which automates and optimizes critical design decisions involved in high-performance image processing implementation on CPU-GPU platforms. CGMBE contains a compile-time analyzer and a runtime scheduler. CGMBE allows data-dependent, dynamic dataflow behavior in the application model and automates the process of scheduling dataflow tasks (actors) and coordinating CPU-GPU data transfers in an integrated manner.

6.1 Application Modeling

In this section, we discuss the modeling techniques used in CGMBE to represent image processing applications. In Sections 6.2 and 6.3, we discuss how we analyze the

resulting models, schedule tasks and data transfers, and synthesize efficient software for CPU-GPU platforms. Then we demonstrate the effectiveness of CGMBE through a data-intensive, computation-intensive application found in the domain of hyperspectral image processing in Section 6.4.

6.1.1 Application Graph

The *application graph* is a dataflow model of a signal processing application being developed using CGMBE. In this model, vertices stand for actors (tasks) while edges represent data dependencies and data flows, and correspond to FIFO queues. Actor (task) invocations consume data values (tokens) from the actor's input edges, invoke the computation associated with the task, and produce tokens on the actor's output edges. In CGMBE, the application model is specified by the system designer using the Dataflow Interchange Format (DIF) language, a textual language for specifying signal processing oriented dataflow graphs [30]. The application graph is developed based on *Single-Rate Synchronous DataFlow* (*SRSDF*) semantics, a special case of the synchronous dataflow (SDF) model [6]. SDF and closely-related models, such as SRSDF, are widely used in the design and implementation of signal processing systems (see Bhattacharyya et al. [3]).

In SRSDF, as in SDF, actors consume and produce constant numbers of tokens on their input and output ports respectively. The number of tokens produced per invocation of an actor on a given output edge must be constant across all invocations of the actor. Similarly, the number of tokens consumed from any given input port must be constant. These numbers of tokens, produced or consumed, are referred to as the *production* and

consumption rates, respectively, of the associated edge e; they are denoted as $prd(e)$ and $cns(e)$. In SRSDF, we impose the additional constraint that $prd(e) = cns(e)$ for all edges e.

Systematic methods can convert general SDF representations into functionally equivalent SRSDF representations [6]. By using such transformations, it is possible to apply CGMBE to general SDF graphs. The equivalent SRSDF representations in general expose parallelism more explicitly at the expense of potentially higher graph complexity (i.e., larger numbers of actors and edges).

An important property of SDF and SRSDF representations is that they can be scheduled to execute in bounded memory regardless of how many times the graphs are iterated [3, 6]. This property is important for reliable operation when applications are deployed to process data streams that are very large or of indefinite length. Such deployment scenarios are common in the signal processing domain and a major motivation for our use of SRSDF as the core model of computation for application graph design in CGMBE.

6.1.2 Well-behaved Patterns of Dynamic Behavior

Many signal processing applications conform mostly to the constant production and consumption rate restriction of SRSDF with the exception of deviations that are localized to specific subgraphs within the dataflow representations. When such a localized subgraph can be encapsulated hierarchically as an SRSDF actor and the token populations of the edges inside the subgraph remain constant across firings of the hierarchical actor, we refer to the subgraph as a *Globally Static, Locally Dynamic* (*GSLD*) pattern of dataflow

behavior. The requirement that the token populations inside the subgraph remain constant is useful to ensure that there is no unbounded accumulation of data inside the subgraph even though its interface is SRSDF. We provide an example to demonstrate a GSLD pattern concretely in Section 6.1.3.

While we apply GSLD patterns in the context of SRSDF encapsulations, the concept can naturally be extended to cases where the dynamic subgraphs are encapsulated within hierarchical SDF representations. This use of GSLD patterns and their application to bounded memory scheduling in CGMBE is inspired by prior work on well-behaved dataflow graphs [50] and quasi-static scheduling of Boolean dataflow (BDF) graphs [51]. These prior works applied similar concepts to exploit higher-level predictability for certain kinds of "well-behaved" dataflow subgraphs that incorporate limited amounts of dataflow behavior at lower levels of abstraction. These prior works used GSLD-related concepts for the purpose of bounded memory verification and single-processor scheduling. In this work, we apply the concepts in the context of application modeling for mapping onto heterogeneous CPU-GPU platforms.

6.1.3 GSLD Pattern Example

Fig. 6.1 shows an example of a GSLD pattern, which corresponds to a dynamic dataflow if-then-else construct for conditional, data-dependent execution between two actors.

In this graph, nodes *A*–*F* represent *homogeneous SDF actors*, SRSDF actors that each produce or consume a single token per firing on each actor port. The Fork actor, also

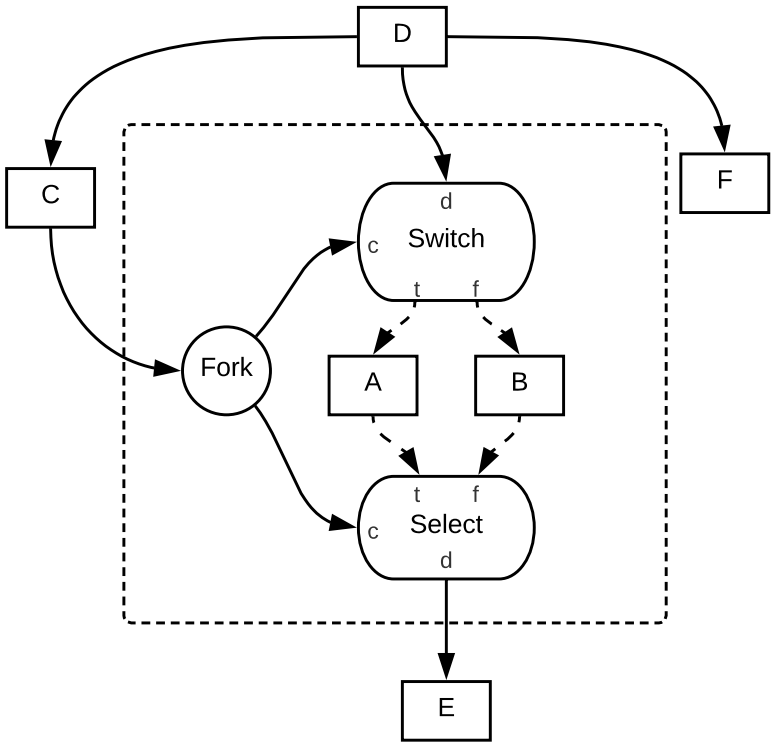


Figure 6.1: An example of a GSLD pattern.

a homogeneous SDF actor, corresponds to a simple broadcast operation. At each firing, it consumes one token on its input port, and copies the consumed data value into separate tokens that are produced on its two output ports.

The Switch and Select actors each exhibit dynamic dataflow behavior on two of their ports. They are called Boolean dataflow actors because the dynamic production or consumption of data is controlled by Boolean-valued tokens, called control tokens, that are consumed from designated control ports of the actors [51]. The control ports of these actors are labeled by the symbol "c" in Fig. 6.1.

At each firing, the Switch actor consumes a single token from its control port, and a single token (data token) from its other input port, and produces a copy of the data token on one of its two output ports depending on the value v of the control token consumed in

the same firing. If $v = true$ then the data token is copied onto the output edge connected to the port labeled t; otherwise, it is copied onto the output edge connected to the port labeled f.

Similarly, the Select actor consumes a single control token from its control port c, and consumes a single token from one of its two other input ports, but not from the other. These two other input ports, labeled t and f, are referred to as the data input ports of the Select actor, and the tokens that are consumed on these ports are referred to as data tokens. On each firing, the Select actor consumes one data token from its t port if the consumed control token is *true*-valued, and otherwise, it consumes one data token from its f port. The consumed data token is then copied to produce a single token on the actor's output port.

From these definitions of the Switch and Select actors, we see that the dashed edges in Fig. 6.1 correspond to edges for which the production and consumption rates are not both constant. For example, the production rate for each output edge of the Switch actor is 0 or 1 depending on the value of the control token.

The actors Switch, Select, Fork, A, and B in Fig. 6.1 constitute an example of a GSLD pattern, which we refer to as the *conditional pattern*. This pattern can be encapsulated within a hierarchical SRSDF actor H as shown in Fig. 6.2. A single firing of H includes a single execution each of the Fork, Switch, and Select actors, and also a single execution of A *or* B, depending on the value of the token consumed on the control input (labeled "c" in Fig. 6.2) to H. Ports x and y on actor H corresponds to ports d on actors Switch and Select in the encapsulated subgraph respectively.

If the firing of H is defined in this manner, it is easily seen that H will consume

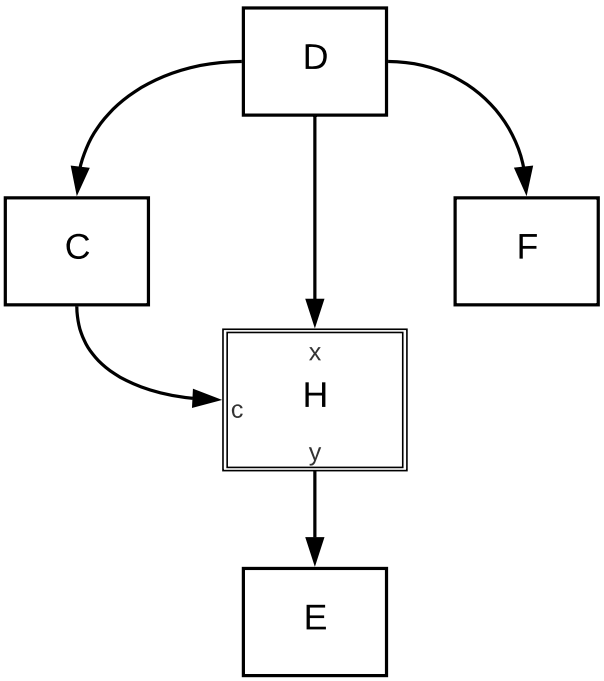


Figure 6.2: Hierarchical representation of the GSLD pattern shown in Fig. 6.1.

a single token on each input port, and produce a single token on its output port, and therefore, that it is an SRSDF actor. Furthermore, it can be verified that regardless of which "branch" is executed (*A* or *B*), the token population remains unchanged for each of the six edges that are encapsulated by *H*. In particular, each of these edges starts and ends each firing of *H* with a token population of zero since any token produced on any of the edges is consumed within the same firing of *H* (i.e., within the same iteration of the encapsulated subgraph).

6.1.4 Actor Types in CGMBE

In summary, the application modeling approach of CGMBE utilizes SRSDF modeling at the highest level of abstraction. Each SRSDF actor in the application graph is either a library actor or a hierarchical actor. A library actor corresponds to a computa-

tional function that is developed using C++ and/or CUDA, and available through *actor libraries* associated with the tool and the application that is being developed. On the other hand, a hierarchical actor in CGMBE is a GSLD pattern whose constituent actors are library actors. At present, CGMBE supports only one level of hierarchical nesting within an application graph. Extensions to handle recursive nesting of GSLDs is an interesting direction for future work.

The CGMBE tool provides an actor library that consists of a small, extensible set of actors that are relevant across a variety of applications. Additionally, an application developer can implement application-specific actors to integrate into a CGMBE application model for a CPU-GPU-targeted signal processing system under development.

As described above, a library actor in CGMBE can be developed using C++, CUDA, or both. Library actors with C++-only implementations are constrained to be mapped to the CPU, while those with CUDA-only implementations are constrained to the GPU. These types of actors are referred to as single-implementation actors, while actors with both C++ and CUDA implementations are referred to as dual-implementation actors. Dual-implementation actors can be mapped to either the CPU or GPU.

For dual-implementation actors in the application graph, the designer can specify the mapping information to enforce whether each actor should be executed on the CPU or the GPU. If such information is not provided by the designer, then CGMBE can use profiling information of actor execution times and inter-processor communication costs to automatically determine which devices the actors should be mapped onto. The automated mapping approach applies the HEFT algorithm with mapping results fixed beforehand for single-implementation actors, and for dual-implementation actors that have designer-

specified mappings. From the result produced by HEFT, the CGMBE automated mapping approach extracts the mapping information, while discarding the task ordering information.

CGMBE provides integrated optimization of scheduling, inter-processor communication, and memory management based on mapping information that is provided implicitly in the library (for single-implementation actors), provided explicitly by the designer (for dual-implementation actors with designer-specified mappings), and derived automatically (for any remaining actors). The optimized scheduling, communication, and memory management approaches employed in CGMBE are discussed further in Sections 6.2 and 6.3.

6.2 Design Flow

CGMBE automates the implementation process for mapping high-performance signal processing applications onto heterogeneous CPU-GPU platforms in a reliable and scalable way. It allows the designer to model the application dataflow (inter-kernel interactions) at a higher level of abstraction and to implement computational kernels independently of each other using platform-oriented programming languages, such as C++ and CUDA. CGMBE frees the designer from complicated and error-prone scheduling issues using an automated scheduling approach that jointly optimizes execution time, data transfers, and memory requirements.

Fig. 6.3 illustrates the design flow utilized in CGMBE. The signal processing system designer specifies the application flowgraph to be developed using the SRSDF model,

where (as described in Section 6.1) each SRSDF actor corresponds either to a function in an actor library or to a GSLD pattern.

Fig. 6.3 shows developer inputs as *rounded rectangles*, components of the CGMBE tool as *straight rectangles*, intermediate results produced by CGMBE as *ovals*, and the resulting implementation as a *hexagon*.

6.2.1 Unfolded SRSDF Graph

The Analysis Engine converts the input application model (SRSDF model G and GSLD patterns) into an expanded form called the Unfolded SRSDF Model, which exposes parallelism between graph iterations. Generating the new graph can be viewed as unrolling J iterations of the application graph G to yield a new SRSDF graph, the *unfolded graph*. The resulting graph represents J consecutive iterations of G and exposes parallelism across multiple graph iterations in addition to the intra-iteration parallelism that is already exposed in G. The designer specifies the *unfolding factor* J as a parameter to the Analysis Engine, as shown in Fig. 6.3.

Dependencies between graph iterations are generally caused by *delays* (initial tokens) associated with graph edges [6], including self-loop edges, which represent actor states. A self-loop edge in a dataflow graph is simply one that has the same source and sink actor. The unfolded graph simply consists of J copies of the application graph when there are no delays on any edge.

The unfolded graph is stored in memory for use by the CGMBE runtime system (the *Runtime Engine*), which allocates a separate dataflow buffer for each edge in the un-

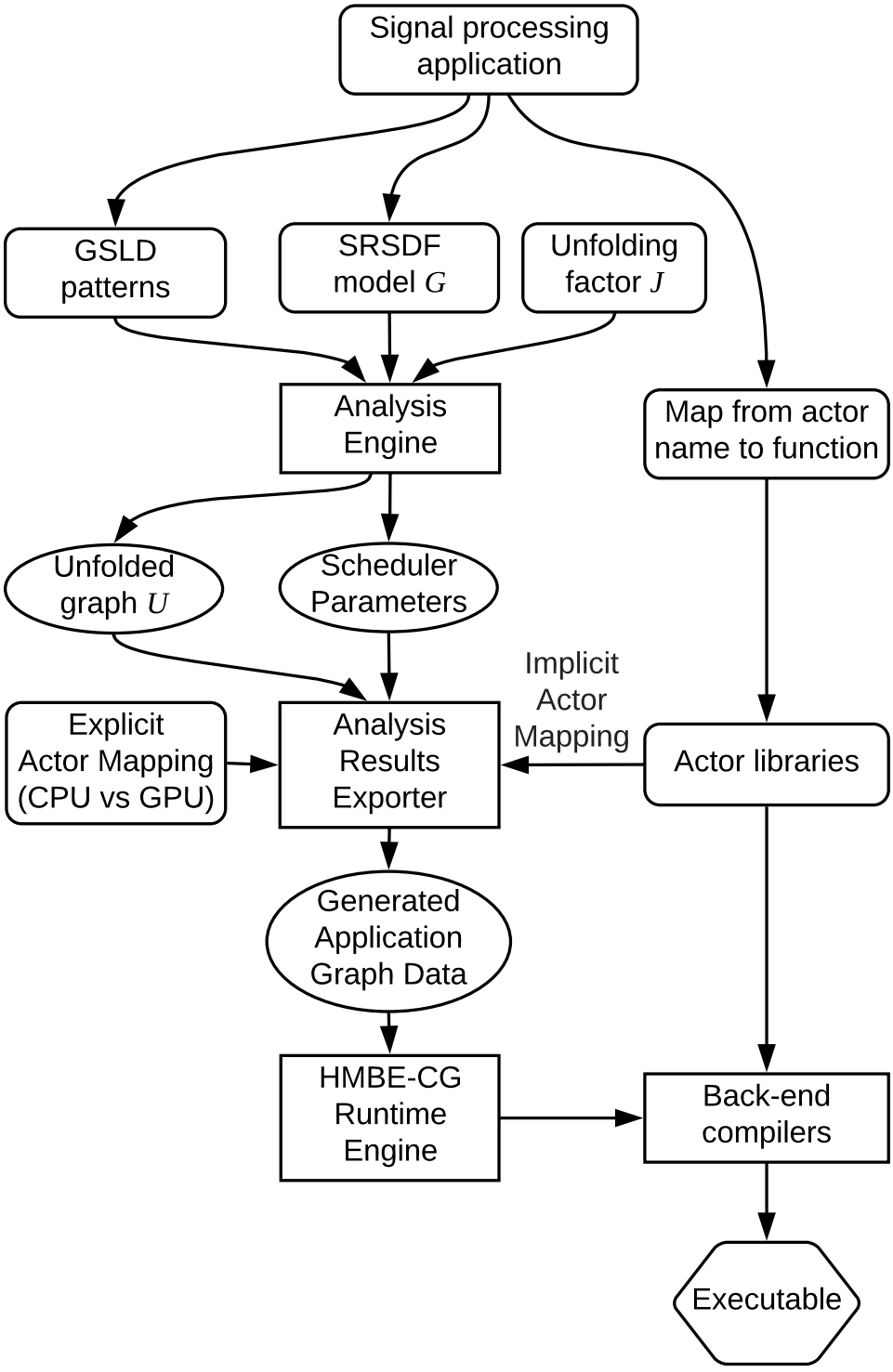
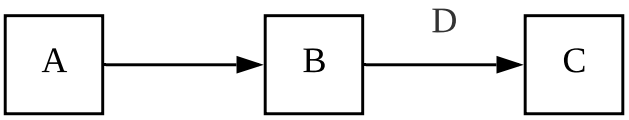


Figure 6.3: Design flow of CGMBE.

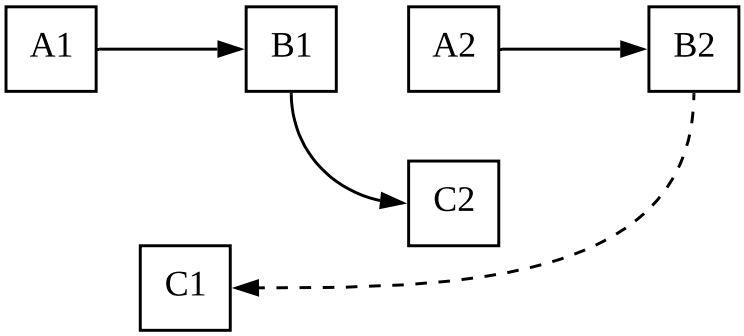
folded graph. As such, the unfolding factor J needs to be selected carefully to effectively expose inter-iteration parallelism without requiring excessive memory consumption. Optimal selection of the unfolding factor involves complex relationships among the application graph, target platform, and scheduling strategy. In Section 6.4, we illustrate how performance varies for different values of J in an application case study that we use in our experimentation with CGMBE.

Rather than automating the selection of J, CGMBE uses J as an optional input, as illustrated in Fig. 6.3, with a default value of $J = 1$. CGMBE automates the derivation of implementations for a given application graph together with arbitrary values of J. As such, the designer can efficiently experiment with different values of J using CGMBE to determine an unfolding factor that best matches the optimization constraints and objectives of the system being developed.

Fig. 6.4 illustrates construction of the unfolded graph U from an SRSDF graph G with an unfolding factor of $J = 2$. Parts (a) and (b) of Fig. 6.4 show G and U, respectively. Each actor X in G is expanded in U into J actors $X_1, X_2, \ldots, X_J$, where each X_i is associated with the same library component(s) or GSLD pattern as the corresponding application graph actor X. The "D" symbol on edge (B, C) represents a single unit of delay. Because of this delay, each invocation of actor C consumes a token produced in the previous application graph iteration by actor B. The dashed edge (B_2, C_1) represents a dependence between two successive iterations of U.



(a) SRSDF graph ***G***



(b) Unfolded graph ***U***

Figure 6.4: A simple example of an unfolded SRSDF graph.

6.2.2 Scheduling Parameters

In addition to the unfolded graph described in Section 6.2.1, the Analysis Engine determines values for various scheduling parameters that are associated with each actor in the unfolded graph U and used for efficient dynamic scheduling by the CGMBE Runtime Engine.

The CGMBE dynamic scheduler is implemented using checks on input data availability (fireability) to determine if an actor should be invoked. The fireability condition for a given actor Z_i in the unfolded graph is formulated in terms of a count $\delta(Z_i)$ of the number of input edges on which data must arrive before Z_i can execute. This count is referred to as the *pending token count* of Z_i. When $\delta(Z_i) = 0$, the actor Z_i can fire. Ad-

ditionally, firing of Z_i can be disabled by setting $\delta(Z_i) = -1$. Such disabling is used by the CGMBE Runtime Engine to implement the dynamic scheduling logic associated with GSLD patterns. For example, for the conditional pattern illustrated in Section 6.1.3, the dynamic scheduler assigns $\delta(A_i) = -1$ or $\delta(B_i) = -1$ depending on the value of the corresponding control token value.

Updating a pending token count relies on *update rules* that are selected by the Analysis Engine. In particular, when an actor Z_i in U fires, the dynamic scheduler executes the update rules for all successors of Z_i in U. If an actor in U is not part of a GSLD pattern, then the actor's pending token count is initialized to its in-degree (number of input edges); its update rule is to decrement the pending token counts of its successors.

On the other hand, update rules associated with patterns are based on the semantics of the pattern. More specifically, the local dynamic dataflow properties of GSLD patterns are implemented in CGMBE through their implementation as δ-update rules for the actors that are part of the patterns. To add a new pattern type to the library of GSLD patterns recognized by CGMBE, the design tool developer or application designer must therefore provide a set of update rules that can be used to implement the pattern.

This association of update rules with GSLD pattern types provides a concrete and modular approach to extend CGMBE with new GSLD patterns that can be used in application models. It also constrains the patterns that can be supported in CGMBE: the patterns must satisfy the constraints associated with GSLD patterns in Section 6.1.2; they must also have local scheduling behaviors that are amenable to implementations based on suitably formulated update rules.

For the conditional GSLD pattern type defined in Section 6.1.3, the update rule

for the Switch actor sets the pending token count value of one of its successors to -1, as described above. Additionally, the pending token count value of the Select actor is initialized to 2 rather than to its in-degree value, which is 3. This allows the Select actor to fire when the single data token arrives on one of its data inputs and the control token is present. The update rule of the Switch actor ensures that on any given firing of the enclosing GSLD pattern, only one data token will arrive at an input of the Select actor.

In Fig. 6.3, the scheduling parameters generated by the Analysis Engine, including the update rules for the actors, are represented by the *Scheduling Parameters* block. Update rules are already pre-defined for different actor and pattern types. The role of the Analysis Engine in this context is to simply package the appropriate analysis rules together with the corresponding graph vertices so they can be retrieved efficiently at execution-time. Other Scheduling Parameter settings generated by the Analysis Engine include the initial value of $\delta(A_i)$ for each actor A_i in the unfolded graph. The pending token count of each A_i is reset to this initial value at every new iteration of the unfolded graph.

6.2.3 Analysis Results Exporter

The results produced by the Analysis Engine together with implicitly- and explicitly-specified actor mapping information (see Section 6.1.4) are exported for use by the CGMBE Runtime Engine in binary form. We refer to this binary output as the *generated application graph data*. This exporting functionality is represented in Fig. 6.3 by the *Analysis Results Exporter* block.

The application graph data generated is stored in a compact binary form using a data exchange protocol based on the Google Protobuf library [37]. In addition to facilitating compact storage, this protocol provides reliable and efficient communication between the CGMBE Analysis Engine and Runtime Engine. Moreover, in contrast with human-readable data serialization formats such as the XML or JSON formats, the binary format used in CGMBE can be imported and exported with lower execution-time overhead.

6.2.4 Executable Generation

To generate the executable associated with a given application graph, unfolding factor, and explicit actor mapping, CGMBE uses back-end compilers for the target platform—the CPU- and GPU-targeted C++ and CUDA compilers—to compile and link the Runtime Engine together with the generated application graph data and the functions for all of the library actors that are used in the application graph. This phase of executable generation is represented by the *Back-end Compilers* and *Executable* blocks in Fig. 6.3.

The CGMBE Runtime Engine uses the generated application graph data to configure just-in-time scheduling, and dynamically optimize host & device utilizations as well as memory management on both host & device. Furthermore, the Runtime Engine uses the generated data to efficiently access relevant information about the global execution state of the heterogeneous computing system. This information allows the dynamic scheduler to optimize execution more effectively. Section 6.3 provides more details on the Runtime Engine and its associated dynamic scheduler.

6.2.5 Limitations

Two current limitations of CGMBE are that it only supports utilizing a single GPU, and all firings of the same actor are constrained to execute on the same device (CPU or GPU). When targeting multiple GPUs, it may be especially useful to allow different subsets of firings for the same actor to execute on different devices—in particular, on different GPUs. Studying efficient algorithms to partition actor firings in this more flexible way, and to support multiple GPUs are useful directions for future work in extending CGMBE.

6.3 Runtime Engine

Before discussing components of CGMBE, we distinguish among three phases of system development and execution: compile-time, setup-time, and execution-time. In CGMBE, compile-time tasks include analyzing the application graph using the Analysis Engine, and exporting derived scheduler parameters and the unfolded graph in a binary form that can be loaded efficiently by the Runtime Engine. Setup-time tasks include loading the Generated Application Graph Data (see Fig. 6.3) into memory, initializing components of the Runtime Engine, and initializing system resources. The initialization processes include allocating memory pools, spawning thread pools, and spawning GPU stream pools.

Execution-time starts when the Runtime Engine fires an actor from the unfolded graph for the first time. Execution-time continues for a fixed number of application graph iterations when the number of application graph iterations, J, is specified by the developer. Alternatively, execution-time continues until the application terminates through external

control when the application is invoked on one or more input streams whose lengths are not known a priori. During execution, the Runtime Engine interprets the n_ath firing of an SRSDF actor A as the n_uth firing in the kth iteration of the unfolded graph, where $k = \lfloor n_a/J \rfloor$, $n_u = (n_a \pmod J) + 1$, J is the unfolding factor, and indexing for firing counts and graph iterations start at 1.

In total, the CGMBE dynamic scheduler iterates through the unfolded graph $\lceil N/J \rceil$ times, where N is the total number of application graph iterations executed, and J is the unfolding factor. The last iteration through the unfolded graph is a partial one if N is not evenly divisible by J.

In the remainder of this section, we present the design of the CGMBE Runtime Engine in a bottom-up fashion, with discussions about its executors, dynamic scheduler, and overall architecture, respectively.

6.3.1 Executors

The CGMBE Runtime Engine includes a single controller thread, which executes on the CPU, and three subsystems, called *executors*, which provide an interface between the controller thread and pools of threads on the CPU and GPU. More specifically, an executor allows the controller thread to submit actor firings through a thread-safe queue to executors so they run asynchronously. An executor allows the controller thread to hide the details of when and where (i.e., on which CPU thread or GPU stream) to run a submitted firing. In this manner, an executor decouples dataflow scheduling from the management of CPU threads and GPU streams.

Each CGMBE executor has a separate thread pool associated with it. These thread pools contain fixed numbers of threads that are spawned at setup-time, and destroyed only when the application completes execution. Throughout an application's execution-time, an executor assigns submitted firings to idle threads. This use of thread pools has three major advantages: thread pools avoid the execution time overhead due to spawning and deallocation of threads; they allow the maximum number of active threads to be controlled by the Runtime Engine, which in turn helps to avoid performance degradation caused by excessive inter-thread contention; and they allow actor invocations to execute asynchronously and reduce idle CPU cycles that are spent on waiting for synchronization conditions.

CGMBE employs three executors: I/O, CPU, and GPU. Each executor has an independent thread pool and receives firing requests from a separate queue, called the *firing request queue*, which is a Single Producer Multiple Consumer (SPMC) queue, where the producer is the controller thread and the consumers are workers in the thread pool. CGMBE provides two types of firing request queues, one is an ordinary thread-safe queue and the other is a thread-safe priority queue. By default, CGMBE uses a simple queue without priority. It also gives the designer the flexibility to control firing request orders by using different types of priority functions in conjunction with the firing request queue. When a priority function is used with the priority request queue, we refer to the queue as a *priority firing request queue*. An executor processes requests from its firing request queue using idle threads in its thread pool. When all threads are busy, an executor will delay processing a request until a thread becomes idle.

When a firing finishes execution, the associated executor writes a "firing complete"

message into a queue called the *notification queue*, which is a thread-safe Multiple Producer Single Consumer (MPSC) queue shared by all executors in the Runtime Engine. Messages in the notification queues are consumed by the controller thread. Upon writing a firing complete message to the notification queue, an executor releases the associated thread so it can be used by a subsequent request from the executor's firing request queue. This design ensures that the actor kernel designer (algorithm developer) can focus on image processing algorithm design rather than worrying about the difficulties of parallel programming, as described in Chapter 1. Thus, algorithm developers can focus on improving the quality of their kernels, while CGMBE takes care of optimizing the efficiency of schedules for executing those kernels on the targeted CPU-GPU platform.

CGMBE provides three different executors because I/O, CPU, and GPU tasks use significantly different sets of system resources on a heterogeneous platform. I/O tasks involve functionality such as reading from or writing to a hard disk, and sending or receiving data on a network. The performance bottlenecks of such tasks are related to the bandwidth of the disk or network. The effective bandwidth is shared among all of the threads, which plateaus the execution time and means that giving more threads or computational resources to such tasks does not necessarily improve a system's performance. For example, if multiple threads read data from a hard disk concurrently, the overall disk read speed may be slower than letting a single thread read data sequentially. Therefore, if the I/O Executor is combined with the CPU Executor, threads occupied by I/O tasks will not be able to efficiently utilize the computational resources allocated for those threads.

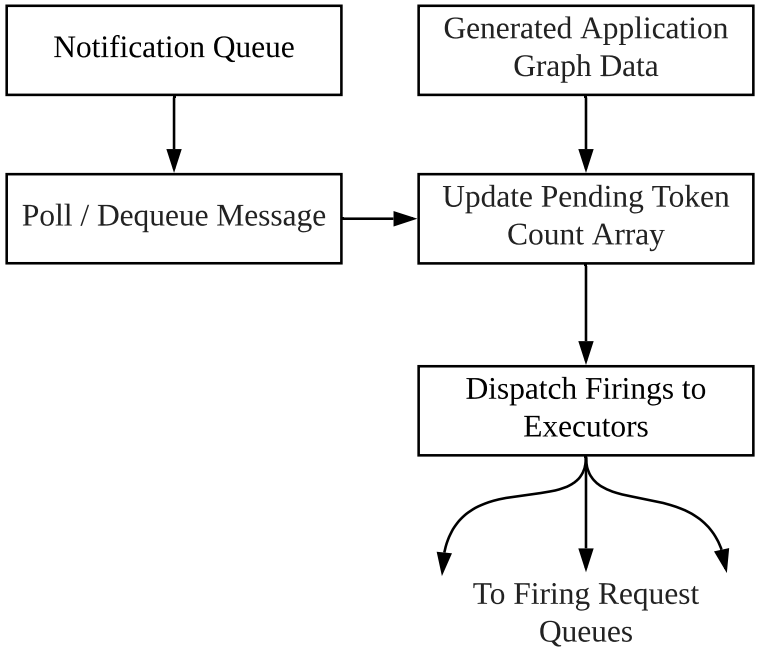


Figure 6.5: Workflow of CGMBE Scheduler.

6.3.2 CGMBE Scheduler

Fig. 6.5 provides an overview of the dynamic scheduler in CGMBE, which we refer to as the *CGMBE Scheduler*. The CGMBE Scheduler corresponds to the controller thread, introduced in Section 6.3.1, of the Runtime Engine. At setup-time, the CGMBE Scheduler loads binary data (generated application graph data) pertaining to the unfolded graph U and initializes the pending token counts for all actors in U based on initialization information provided in the generated application graph data. The pending token count data is maintained at runtime in an integer array that is indexed by unique-integer actor IDs; these IDs are associated with the actors in U at compile time.

The pending token count array together with the graph connectivity information in the generated application graph data allow the CGMBE Scheduler to efficiently extract

ready actors, which are actors in U that have zero-valued pending token counts. As described in Section 6.2.2, ready actors have sufficient data on their inputs to be fired. They are dispatched as they are identified by the CGMBE Scheduler to their corresponding executors. The mapping from actors to executors occurs at compile time and is based on information available in the actor libraries. This mapping information is stored as part of the generated application graph data.

The actions of polling messages from the notification queue, updating the pending token count array and dispatching ready firings all happen within a single thread, namely the controller thread of the Runtime Engine. Since reading and writing of the pending token count array are performed in the same thread, there is no potential for data races between these operations.

The processes of updating the pending token count array and dispatching firings to the firing request queues are non-blocking. If the Notification Queue is empty when it is polled by the CGMBE Scheduler, the controller thread blocks until at least one message arrives in the Notification Queue. Otherwise, if the queue is non-empty, the controller thread iteratively dequeues and processes messages, one by one, as long as the queue is non-empty. The processing of a message in this context involves, as illustrated in Fig. 6.5, executing update rules for the successor actors associated with the completed firing, which in turn results in updates to the pending token count array. Each successor is then examined to determine if it is fireable as a result of these updates and, when it is, a firing request is dispatched to invoke the successor using the appropriate executor.

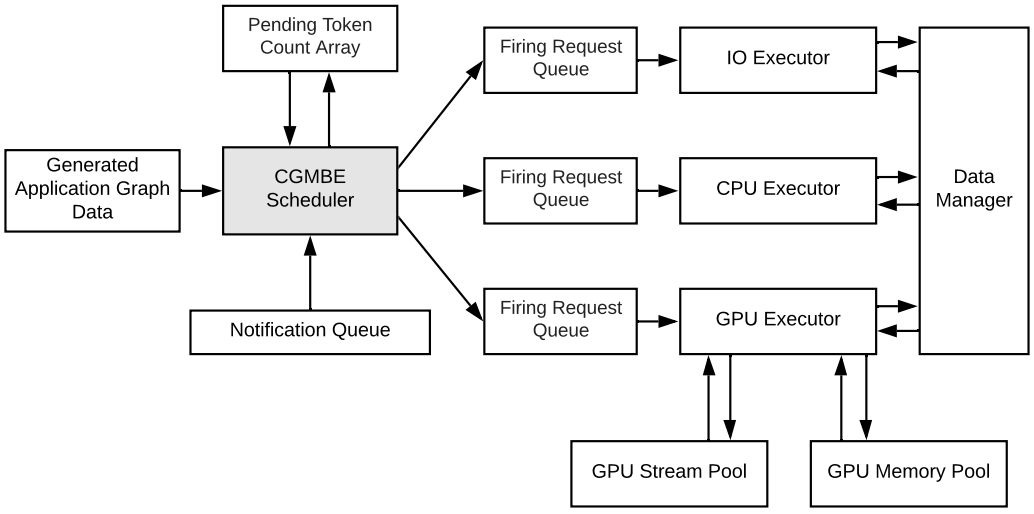


Figure 6.6: Structure of the CGMBE Runtime Engine.

6.3.3 Top-Level Workflow

Fig. 6.6 shows the overall workflow of the Runtime Engine. Previous sections have already presented several aspects of this workflow: Section 6.3.2 discussed the interactions among the CGMBE Scheduler, Generated Application Graph Data, Pending Token Count Array, Notification Queue, and the three firing request queues; Section 6.3.1 presented the interactions between each firing request queue and its corresponding executor. This section completes the workflow's description by discussing the interactions involving the Data Manager, GPU Stream Pool, and GPU Memory Pool.

The Data Manager provides a unified location to manage all application specific data, which is composed of two components: (1) a read-only component that contains variables that will be read by different actor invocations, such as configuration parameters; (2) a lock-free component that manages memory for dataflow edges in the unfolded graph. The Data Manager provides a simple, efficient, and reliable interface for actors to acquire

and release buffers to read from and write to.

The GPU Executor has an independent CPU thread pool for launching GPU kernels on the device. It also interacts with a GPU Stream Pool and GPU Memory Pool. The GPU Executor is designed differently from the other two executors to manage GPU streams and handle device-specific memory interfacing requirements. The GPU Stream Pool contains multiple CUDA streams with each stream ensuring that tasks assigned to it are executed on the GPU in a given order. The GPU Executor relies on the CUDA scheduler to launch CUDA kernels on streams. Multiple CUDA streams can execute concurrently on the same device whereas the kernels within a given stream execute sequentially. The GPU Memory Pool encapsulates blocks of device memory that are allocated during setup-time and that can be reused among GPU tasks during execution-time with low overhead. The GPU Memory Pool is used to avoid device synchronization caused by dynamically allocating memory during execution-time.

Two possible concerns for the CGMBE scheduler are the overhead and scalability of the controller thread as the number of concurrent worker threads becomes large; this is a serious concern as multicore CPUs with 64 cores and 128 hardware threads have been announced and these numbers are trending higher [52].

The controller thread has low overhead for two reasons. First, it operates on a small subset of the graph, the execution boundary of the unfolded SRSDF graph; its workload is proportional to the number of invocations that have not yet completed their execution but whose predecessors have all completed. Second, there are only a few simple operations per actor invocation and firing of a complete message as a result of using the cached compile-time analysis. This minimal constant overhead is negligible when compared to

the cost of complicated image processing kernels.

Furthermore, the controller thread enables dynamic scheduling, which compares very favorably to static scheduling; the resulting performance improvements far exceed the overhead of the controller thread. According to our measurements in Table 6.3, it is common for the standard deviation of actor execution times to exceed 10 %, which makes it difficult for static schedules to compete with low-overhead dynamic scheduling, as enabled by the controller thread.

In the class of high-performance signal processing applications targeted by CGMBE, most computations involve actors of significant complexity. Typical actors include filtering operations or domain transformations on single- or multi-dimensional signals. The dataflow modeling foundation of CGMBE is well-matched to this class of image processing applications (e.g., see Bhattacharyya et al. [3]). Since the typical actor complexity is significant and the CGMBE controller thread involves only lightweight operations, as described above, the controller thread introduces minimal overhead in overall system execution.

6.4 Experiments

In this section, we evaluate the effectiveness of CGMBE through a data- and computation-intensive application from the domain of hyperspectral image processing. This application is *Vertex Component Analysis* (*VCA*) for linear unmixing for hyperspectral images [53].

6.4.1 Hyperspectral Imaging and Vertex Component Analysis

Hyperspectral image processing is used in a variety of applications, with extracting information from remote sensing being one of the most common applications. Photographic images typically have 3 or 4 color channels per pixel: Red, Green, and Blue (RGB) channels, and possibly an additional "alpha" channel. In contrast, hyperspectral images typically contain hundreds of channels. These additional channels provide much greater spectral resolution and enable much more powerful forms of knowledge extraction, such as classification of different types of objects or materials. However, this increased spectral content makes hyperspectral image processing much more data-intensive and computationally challenging when compared to conventional image processing tasks.

Linear unmixing of hyperspectral images is used to identify different types of substances present in captured image scenes and to determine the abundances of the substances identified. The method is designed to exploit differences in reflectance spectra between different substances. Many linear unmixing algorithms have been proposed in the literature. These include the Pixel Purity Index (PPI) [54], N-FINDR [55], Independent Component Analysis (ICA) [56], and Vertex Component Analysis (VCA) [53]. Many researchers have worked on improving hyperspectral linear unmixing algorithms in terms of considerations such as unmixing accuracy, computational complexity, and available parallelism. In the experiments that we develop in this section, we apply CGMBE to the well-known VCA algorithm, which is an unsupervised linear unmixing algorithm that has been demonstrated to have similar or better accuracy than alternative algorithms, while having lower computational complexity [53].

6.4.2 Baseline Sequential Implementation

As a starting point for our experimentation with the VCA application, we obtained an open source MATLAB-based reference implementation [57]. By referring to this MATLAB-based version, we developed a C++-based sequential implementation, which we refer to as the *Sequential Version* of VCA in the remainder of this section. The Sequential Version uses the Eigen library [58], a C++ library for linear algebra.

Fig. 6.7 illustrates the workflow of the Sequential Version. The application is designed to process hyperspectral images stored in HDF5 format [59]; HDF5 is built for storing and organizing large amounts of data and for efficient I/O processing. We refer to the vertices in Fig. 6.7 as *tasks* instead of as actors since the Sequential Version is not designed to adhere strictly to dataflow semantics.

The computation proceeds as follows. The Source task produces the directory path for the next HDF5-format image I to be accessed by the Read task. This latter task then loads the entire data for I into host memory from disk. The Convert task converts the raw image data to a matrix data type in Eigen; we denote this converted data by R. The Zero Mean task computes the average value of each spectral band and stores its results in a vector R_m. This task also shifts the pixel values in each band so that each row has zero mean; the result after these shifting operations is denoted by R_0. The Orth task takes matrices as input and outputs the results of orthogonalizing these matrices. The SVD task computes the Singular Value Decompositions (SVDs) of the orthogonal matrices computed by the Orth actors.

The task graph in Fig. 6.7 includes a data-dependent branching operation, repre-

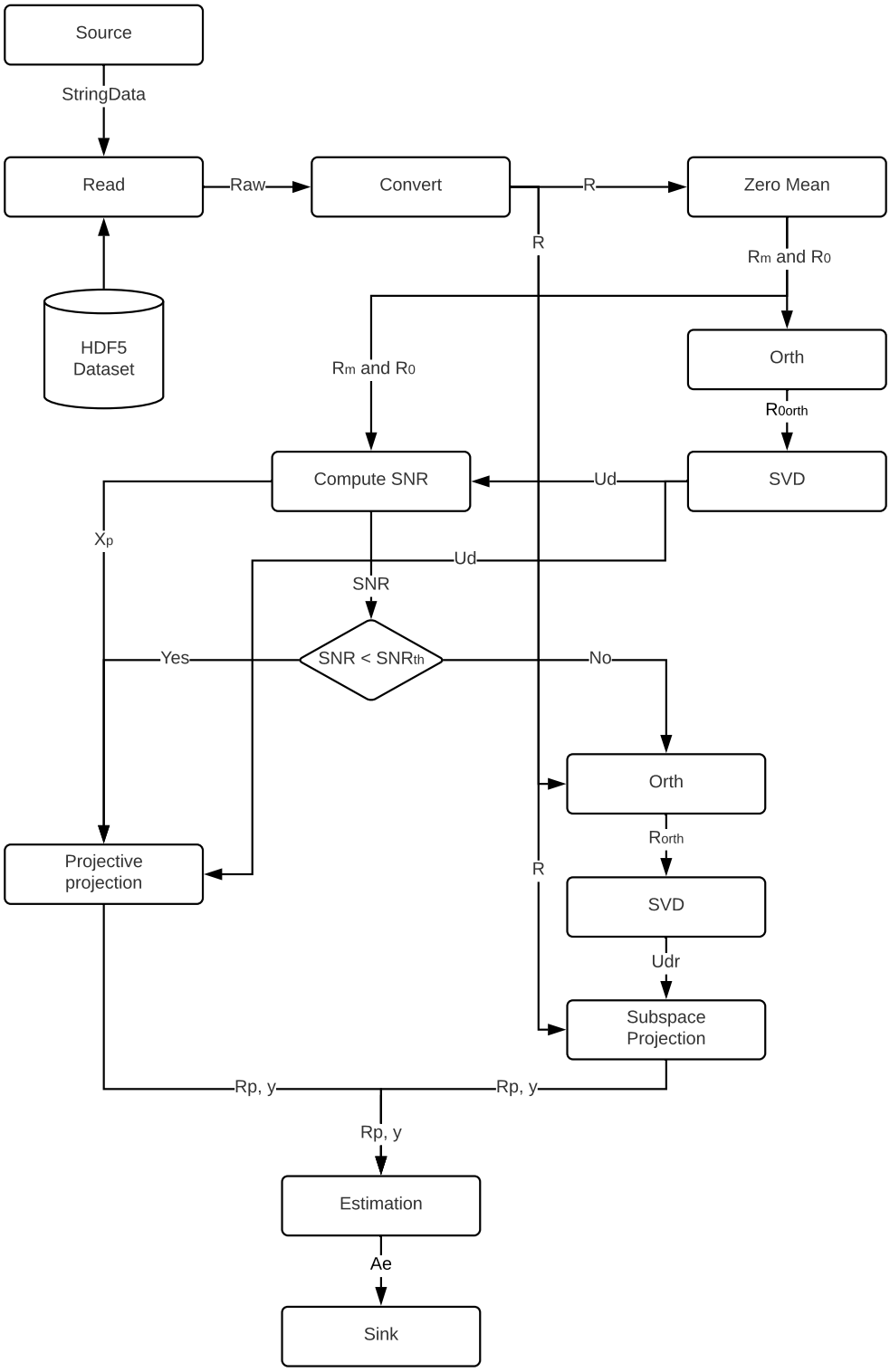


Figure 6.7: Sequential Version of VCA application (C++, Eigen, and HDF5).

sented by the task labeled SNR $<$ SNR_{th}. This branching operation is based on the result of the Compute SNR task, which computes the signal-to-noise ratio (SNR) of its input image. If the SNR is less than a predefined threshold, then the branch with the Projective Projection task is selected. Otherwise, the branch with the Subspace Projection task is selected. The selected branch will produce an ordered pair (R_p, y), where both R_p and y are two-dimensional matrices. These two matrices are then processed by the Estimation task, which applies an iterative algorithm to generate the estimated substance types and abundances for the input image I.

We profiled the Sequential Version and determined that the Orth task is the most computationally intensive task, which operates on R and R_0. This result makes sense because the R and R_0 images are relatively large compared to other intermediate results. For example, the hyperspectral images used in our experiments consist of 224 bands with each band having $128 \times 128 = 16\,384$ pixels. For this input image size, the R and R_0 matrices each have dimensions $224 \times 16\,384$ and, as such, each contain over 3.6 million pixels. Deeper analysis of the Orth task shows that most of its time is spent on matrix multiplication. This motivates us to migrate the matrix multiplication operations from the CPU to the GPU for acceleration of the VCA application.

6.4.3 CGMBE Application Model

Fig. 6.8 shows the dataflow-based application model that we developed to implement and experiment with the VCA application using CGMBE.

The formal model of the VCA application in CGMBE uses three Fork actors to

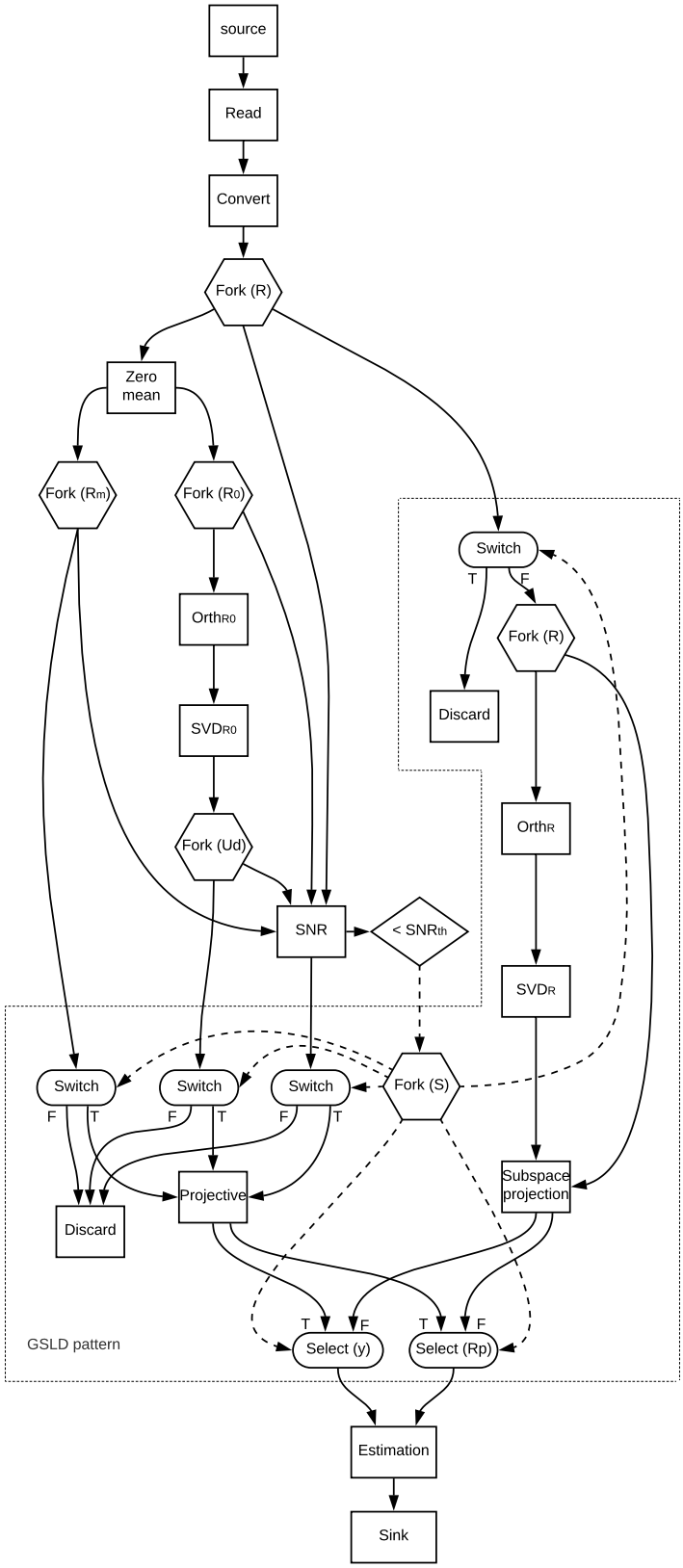


Figure 6.8: CGMBE application model for the VCA application.

represent data broadcasting functionality among subsets of actors. The CGMBE Analysis Engine and Data Manager have special provisions to "optimize away" Fork actors given that the Fork functionality is commonly needed in signal processing applications to duplicate data that will be used in multiple different actors in dataflow semantics. In particular, Fork actors are implemented by simply having all receiving actors share a common buffer so that there is no need for data duplication. This is a specialized application to Fork actors of the concept of passive component implementation for dataflow graphs [60].

In the application model, the subgraph enclosed within the dashed borders in Fig. 6.8 corresponds to an application-specific GSLD pattern that has been incorporated into the CGMBE-based VCA system design. From the "outside", this pattern behaves as an SRSDF actor and update rules for the pattern can be readily derived by extending the update rule formulations for the conditional pattern example discussed in Section 6.1.3. We omit the details of these rules here for brevity.

6.4.4 Experimental Setup

To evaluate the effectiveness of the CGMBE-based system design for the VCA application, we performed extensive experiments with the design using two different CPU-GPU platforms and a variety of software configurations on each platform. We generated a synthetic dataset of 200 hyperspectral images using a randomized image generation model [53]. Each generated image has 224 bands of 128×128 pixels; 98 of the 200 images have an SNR below the threshold and use the "Projective" threshold, while the remaining 102 images have an SNR above the threshold and use a "Subspace Projec-

Platform	CPU	NVIDIA GPU	RAM
Desktop PC	AMD Ryzen 1700X 8 cores, 3.4 GHz	GTX 1060 6 GB	16 GB DDR4 3000 MHz
High-end w/s	2x Intel E5-2699 v4 44 cores, 2.2 GHz	GTX 1080 Ti 11 GB	512 GB DDR4 2400 MHz

Table 6.1: Heterogeneous platforms used in our experiments.

tion". We use the same randomly-generated dataset for all experiments reported in this section to guarantee that the computational workloads are the same for all experiments. To validate functional correctness, we compared the output produced by the Sequential Version and the CGMBE-based implementation to the open-source MATLAB reference code [57].

As motivated in Section 6.4.2, we mapped the Orth actor to the GPU and all other actors to the CPU. These mapping decisions were given to CGMBE through the Explicit Actor Mapping and Implicit Actor Mapping components shown in Fig. 6.3. Additionally, we compared this mapping with the automated actor-to-device mapping generated from the HEFT algorithm in the Analysis Engine (see Section 6.1.4). Our implementation of HEFT in the Analysis Engine utilizes profiling results for actor execution times and communication costs that are exported from the lightweight CGMBE task profiler.

Table 6.1 summarizes the heterogeneous platforms that we used. The operating systems on the platforms are different versions of Linux (Desktop PC: Ubuntu 18.04, High-end workstation: RedHat 7.6).

6.4.5 Measurements

Table 6.2 summarizes experimental results on execution time and peak memory usage for CGMBE, HTGS, and the Sequential Version; these results are averaged over 100 runs. The results show that HTGS and CGMBE significantly boost application performance on the targeted platforms. However, CGMBE outperforms HTGS by $\approx$9 % in execution time, while also using less host memory. The HTGS-based implementation is already very effective because it uses all CPU cores and takes full advantage of the GPU given no other bottlenecks. The memory usage of HTGS might be further reduced by a more experienced user through attaching additional memory pools to tasks. We anticipate that the better performance of CGMBE can be attributed to the more efficient scheduler in CGMBE. HTGS spawns more threads than the platform can effectively utilize and leaves thread scheduling to the operating system. By contrast, CGMBE employs thread pools and actively controls the schedule using its dynamic scheduler so there is less thread contention in the CGMBE Runtime Engine at the cost of work stealing within thread pool. The memory utilization difference between the HTGS and CGMBE versions can be attributed to the use of dynamic memory allocation by the two implementations. HTGS limits memory utilization by controlling the degree of parallelism in each task and the size of memory pool attached to some tasks. By contrast, CGMBE controls memory explicitly using a Data Manager, which limits the maximum number of tokens (data items) in the system.

For the experimental results reported in Table 6.2, we used the OpenBLAS-based back-end for the Eigen library. To set the thread count T_{OB} for OpenBLAS when used with

Platform	Sequential	HTGS	CGMBE
Desktop PC	21.09 ± 0.13 s 82 MB	2.93 ± 0.03 s 6873 MB	2.63 ± 0.03 s 1842 MB
High-end w/s	34.76 ± 0.22 s 80 MB	2.68 ± 0.07 s 8891 MB	2.47 ± 0.08 s 4966 MB

Table 6.2: Execution time (mean & standard deviation over 100 runs) and peak CPU memory usage comparison.

HTGS and CGMBE, we evaluated the VCA application system for $T_{\text{OB}} = 1, 2, \ldots, 16$, and determined that $T_{\text{OB}} = 1$ leads to the best system-level performance. Therefore, we fixed $T_{\text{OB}} = 1$ in our reported experiments with HTGS and CGMBE. For the Sequential Version results, we used the best evaluated OpenBLAS thread configuration for each platform. For example, when testing the execution time for the Sequential Version on the desktop PC, we set $T_{\text{OB}} = 7$. The difference between the optimal thread counts for the Sequential Version versus the HTGS and CGMBE versions demonstrates that intra-actor parallelism is not sufficient to fully exploit the targeted heterogeneous platforms for the VCA application because intra-actor parallelism in the Sequential Version cause much more explicit synchronization among threads, which wastes more CPU cycles, than HTGS or CGMBE. One additional note, the Sequential Version is slower on the high-end workstation than on the Desktop PC because the workstation's CPU consists of lower frequency cores than its Desktop PC counterpart.

Next, we experimented with different values of the unfolding factor J using CGMBE. Fig. 6.9 shows measured results on the variation of execution time and memory usage on the high-end workstation for different values of J. The solid and dashed curves correspond to execution time and memory usage respectively. As the unfolding factor gets

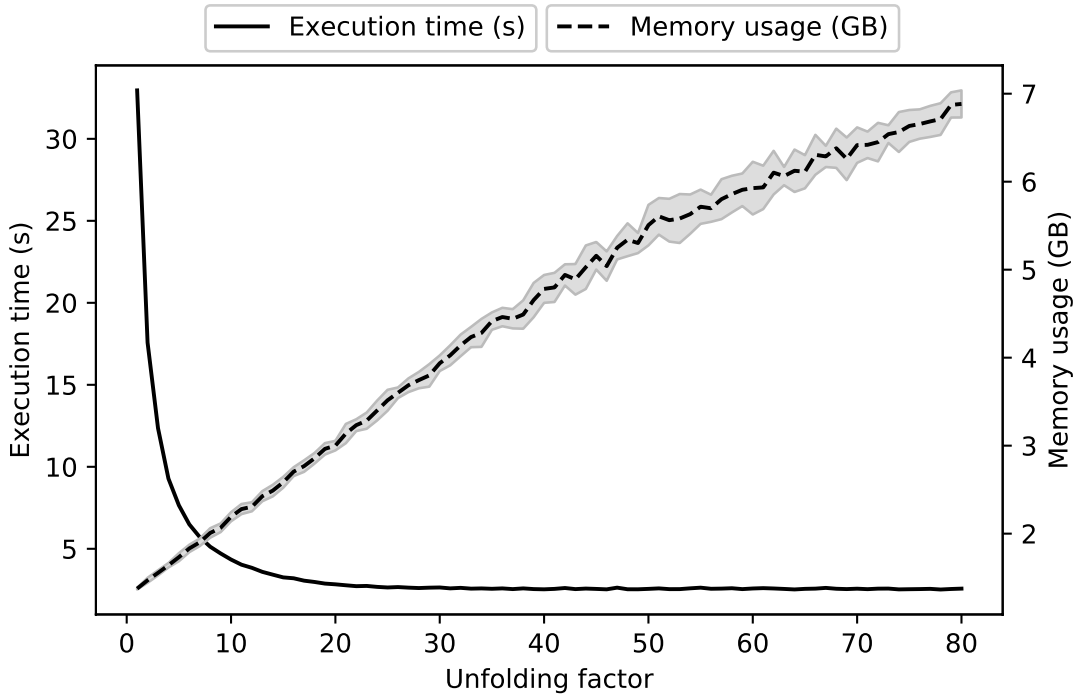


Figure 6.9: Execution time and memory usage (mean & standard deviation over 10 runs) versus unfolding factor for the VCA application on the high-end workstation.

larger from its default value $J = 1$, the execution time exhibits significant improvement until saturating at around $J = 30$. We anticipate that after this point of saturation, the system bottleneck is no longer computational power, but rather memory bandwidth. We also find that the memory usage increases almost linearly as J increases due to larger memory costs for storing the unfolded graph and its associated buffers. These results demonstrate the utility of CGMBE in supporting experimentation to explore trade-offs between memory cost and throughput associated with the important system design parameter J. Generating trade-off curves, such as those shown in Fig. 6.9, is straightforward using CGMBE since the process of generating an implementation for a given value of J is fully automated once the mapping is determined. Similar automation can be applied to experiment with trade-offs across different mappings for fixed J.

Both CGMBE and HTGS require platform-specific configurations. However, the configuration for CGMBE is simpler than its HTGS counterpart as the designer only needs to provide the unfolding factor, thread pool sizes for the CPU, I/O and GPU executors, and mapping of actors between the host and device. Additionally, the high degree of automation in CGMBE allows the designer to experiment with trade-offs among these settings through a highly automated and optimized process. In contrast, HTGS requires the designer to determine the number of threads $N(T)$ allocated for each CPU-targeted task T based on factors such as its computational intensity and the target platform. This is in addition to requiring a designer-specified mapping of actors to the host or device. The set of all combinations $\{N(T_1), N(T_2), \ldots, N(T_n)\}$ for a task graph containing n CPU-targeted tasks $T_1, T_2, \ldots, T_n$ grows very rapidly with n and the complexity of the target platform. As such, the selection of these values can be a highly complex and time-consuming task. Furthermore, when porting an HTGS-based application from one platform to another, or when experimenting with a different actor mapping, the values of the $N(T_i)$s may need to be modified again.

On the other hand, a limitation of CGMBE compared to HTGS is that its programming model is restricted to SRSDF with optional use of GSLD patterns. However, the developments and results presented in this chapter about CGMBE can help to inform specialized models of computation and optimization techniques that may be useful for adaptation into future versions of HTGS. This is an interesting direction for future work.

Static scheduling approaches, such as HEFT, that apply constant-valued estimates of single-threaded actor execution times may be highly inefficient for implementing the VCA application on multithreaded runtime environments. We anticipate that this incom-

patibility arises for two reasons. First, from our experiments with the VCA application, the execution time of a single-threaded actor implementation may be significantly less (e.g., more 50 % less for some actors) than the execution time of a multithreaded implementation of the same actor. In such cases, the overhead of multithreaded processing dominates any potential performance enhancement due to concurrent execution of multiple actor firings. Second, the standard deviation of execution times is more than 10 %.

To provide more concrete insight into these points, Table 6.3 shows profile data for actor execution times derived from CGMBE on the Desktop PC platform. For the single-threading environment in Table 6.3, the thread pool of the corresponding executor only has a single worker thread. In contrast, actors in the multi-threading environment run in the executor with a thread pool of size 16, which represents the maximum CPU-supported concurrency for the target platform.

The results in Table 6.3 demonstrate how the actor profiles can help the designer understand which actors are the most computationally intensive and dominate the computing time. Such profiling results are useful in deciding which actors the designer should focus on when performing actor-level optimization. However, due to the large amount of variation for actor execution times among different thread-based computing environments, static scheduling algorithms that are based on profiled execution times may be highly inefficient in such environments.

CGMBE provides a lightweight tracing tool that provides users with information about the detailed working status of all CPU threads, GPU streams and IO devices. The performance information provided is related to abstractions in the dataflow graph so as to give the designer higher-level insight into relevant performance issues. The data is pro-

Actor Name	Single-threading Environment	Multi-threading Environment
source	0.01 ± 0.00 ms	0.01 ± 0.00 ms
Read	3.35 ± 0.48 ms	7.50 ± 1.61 ms
Convert	4.36 ± 0.84 ms	11.12 ± 3.58 ms
Zero mean	6.09 ± 0.46 ms	15.55 ± 3.85 ms
Orth R_0	31.70 ± 0.55 ms	76.08 ± 6.92 ms
SVD R_0	29.75 ± 1.47 ms	62.55 ± 5.05 ms
Orth R	31.50 ± 0.52 ms	75.08 ± 9.31 ms
SVD R	2.24 ± 0.43 ms	4.90 ± 0.96 ms
Projective	4.09 ± 0.43 ms	10.15 ± 2.40 ms
Subspace	28.05 ± 1.38 ms	65.15 ± 9.26 ms
Estimation	0.01 ± 0.00 ms	0.02 ± 0.16 ms
Sink	0.01 ± 0.00 ms	0.01 ± 0.00 ms

Table 6.3: Profiles of actor execution times (mean & standard deviation over 100 runs) in different threading environments.

vided across the entire duration of application execution. For example, the information exposed from the tracing tool includes the times at which each actor invocation starts and ends, when a computing resource is idle, and which set of resources presents a performance bottleneck at any given time.

Fig. 6.10 presents information derived from the CGMBE tracing tool for the VCA application. The illustration of the performance data is constructed by first exporting the output of the tracing tool from CGMBE as a JSON file in Trace Event Format. The JSON file is then provided as input to Chrome Tracing [61,62] to generate the illustration shown in Fig. 6.10.

For example, from Fig. 6.10 we can see that I/O speed is the performance bottleneck during the first 100 ms of execution. Then CPU resources are mostly occupied by actor

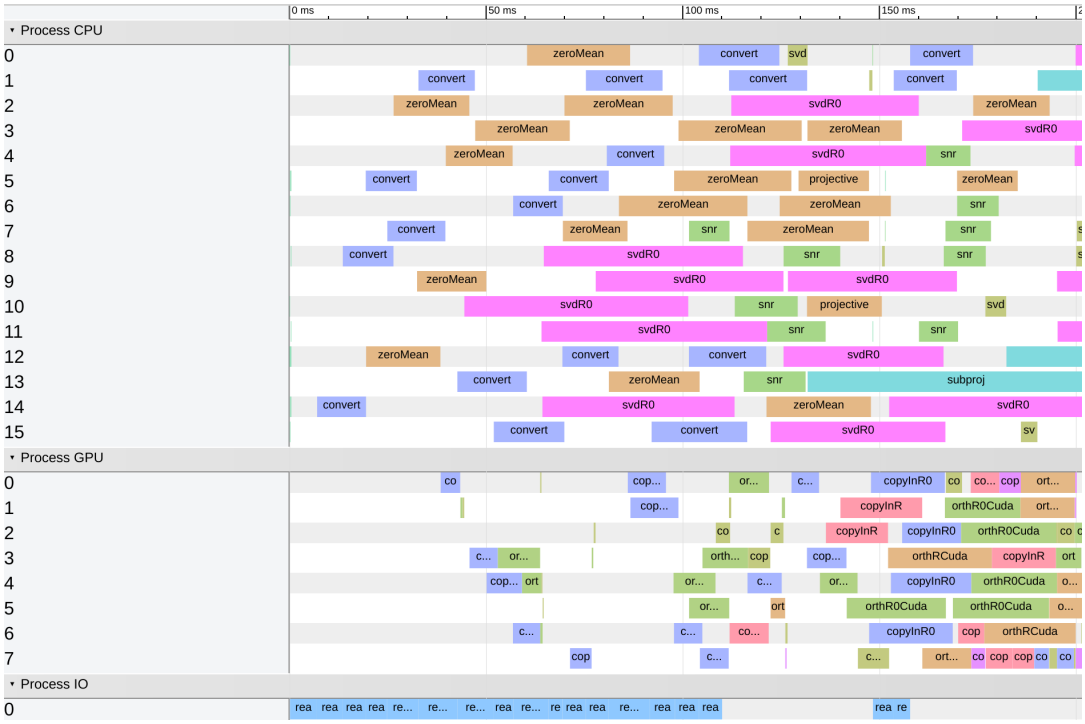


Figure 6.10: An illustration of information derived from the CGMBE tracing tool.

invocations related to the zero mean and SVD actors. CPU computing capacity is the bottleneck during the time interval from 100 ms to 150 ms after the start of execution. Finally, GPU streams are fully utilized starting from around the 150 ms point.

In contrast to traditional performance analysis tools, such as GNU gprof [63], which collects data for each function call and introduces considerably higher execution time overhead, the tracing tool in CGMBE uses a lightweight approach that simply records the time before and after each task invocation in the executor, and only computes and exports the tracing results at the end the application execution. Additionally, the CGMBE tracing tool gathers detailed information about execution status for I/O and computation resources throughout the entire execution process, and provides a high-level (actor-level) view of the execution process.

Next, we experimented with different software configurations on the Desktop PC.

Table 6.4 summarizes execution times and memory usage levels. All results shown in Table 6.4 use the same computation kernels for the CPU and GPU and their differences are attributed to differences in the scheduling approaches that are employed. All of the average execution times and standard deviations are computed from 100 runs.

We first compare two sequential implementations. The CPU-GPU Sequential implementation requires approximately the same execution time as the CPU Only Sequential implementation. This is because the CPU-GPU Sequential implementation does not overlap computations between the CPU and GPU, nor does it overlap computation with data motion. Although HEFT is a static scheduling algorithm, the HEFT implementation evaluated in Table 6.4 still has better performance than the two sequential implementations. However, because the HEFT scheduling algorithm does not exploit parallelism across different application graph iterations, and its schedule cannot be adapted dynamically based on execution-time variations, the speedup achieved by HEFT is very limited in our experiments.

In Table 6.4, CGMBE Basic refers to a version of CGMBE that utilizes both CPU and GPU devices, but does not include profiling or priority firing request queue features. On the other hand, the last three rows of the table correspond to versions of CGMBE that augment CGMBE Basic with priority firing request queues, where the queues are configured with three different priority functions. CGMBE with Profiling augments CGMBE Basic with profiling, and without use of priority firing request queues.

Because CGMBE Basic can overlap computation between different devices, and also overlap computation with data motion, the difference in execution time due to use of the GPU is much more significant than that in the sequential implementations. Specif-

ically, CGMBE Basic is over 25 % faster than the CPU Only CGMBE implementation. Furthermore, the task-level profiling capability of CGMBE is so lightweight that the run-time overhead of the CGMBE with Profiling version is only about 0.3 % of the execution time of CGMBE Basic for the VCA application.

CGMBE with Profiling augments CGMBE Basic with profiling capabilities that collect actor-level execution data throughout application execution, and display values for average actor execution times after application execution has completed. CGMBE with Profiling is provided as an option that allows designers to derive useful information about actor-level performance.

In the last three rows of Table 6.4, we utilized the priority firing request queue in CGMBE and compared the performance resulting from different priority functions. There is no significant difference in execution time among the three implementations represented in these last three rows. We anticipate that this is because the CGMBE Runtime Engine is able to keep all worker threads busy for this application regardless of how the tasks are ordered in the firing request queue. Here, the HEFT heuristic gives the firing order extracted from the conventional HEFT algorithm; "CP" stands for the Critical Path heuristic, where actors with larger critical path lengths are executed earlier; and "ICP" stands for the Inverse Critical Path heuristic, where actors with smaller critical path values are executed earlier. Comparing the results for HEFT, CP, and ICP with the result of CGMBE Basic, we find that the use of priority firing request queues does not add significant overhead to the system. Second, the ICP Heuristic uses less memory than the other CGMBE implementations; we anticipate that this is because it allows CGMBE to release memory earlier compared to the CP and HEFT heuristics. However, CGMBE with ICP

Configuration	Execution Time	Memory Usage
CPU Only Sequential	21.09 ± 0.13 s	82 MB
CPU-GPU Sequential	21.08 ± 0.25 s	82 MB
HEFT	15.33 ± 0.04 s	664 MB
CPU Only CGMBE	3.53 ± 0.02 s	1478 MB
CGMBE Basic	2.63 ± 0.03 s	1842 MB
CGMBE with Profiling	2.64 ± 0.03 s	1855 MB
CGMBE with HEFT Heuristic	2.64 ± 0.03 s	1838 MB
CGMBE with CP Heuristic	2.63 ± 0.02 s	1859 MB
CGMBE with ICP Heuristic	2.68 ± 0.03 s	1799 MB

Table 6.4: Execution time (mean & standard deviation over 100 runs) and memory usage for different software configurations.

Heuristic runs a little slower than the other two CGMBE implementations with priority firing request queues.

From the results summarized in Table 6.4, we find that the trade-off between execution time and memory usage is favorable for CGMBE Basic. Thus, we select CGMBE Basic as the default configuration in the overall CGMBE design tool. However, the CGMBE tool makes it easy for the designer to select among the other options listed in Table 6.4, and also to incorporate and experiment with new priority functions for coordinating priority firing request queues.

If the hyperspectral images that we experimented with are input to CGMBE Basic as a video stream, then the system can process this stream at approximately 75 frames per second. This demonstrates efficient, real-time processing of hyperspectral image data on a low cost, off-the-shelf computing platform. Moreover, the VCA application is mapped by CGMBE into this real-time implementation using a systematic and highly automated

process.

6.5 Conclusion

In this chapter, we have presented CGMBE, a new design tool for model-based design and implementation of real-time image processing applications targeted to heterogeneous CPU-GPU platforms. We introduced in detail the application modeling methods and design flow employed in CGMBE, and presented details on the CGMBE Runtime Engine, which applies dynamic scheduling techniques to optimize system performance. We demonstrated the effectiveness of CGMBE through experiments involving a complex data-intensive application for hyperspectral image processing and a variety of CPU-GPU platforms and software configurations. Interesting directions for future work include extending CGMBE to support multiple GPUs, developing hierarchical controller strategies within the Runtime Engine to support multi-node image processing environments, developing support for nesting of globally static locally dynamic patterns, and adapting relevant methods from CGMBE into HTGS.

Chapter 7: Conclusion and Future Work

In this chapter, we first summarize the contribution of the thesis. We then explore interesting directions for future work.

7.1 Conclusion

In this thesis, we have introduced the HMBE scheduler, the HI-HTGS design tool and the CPU-GPU Model-based Engine. In this section, we summarize the key contributions resulting from our investigation into each of these tools.

First, we introduced a novel model-based dataflow scheduler, the HTGS Model-Based Engine (HMBE), for design optimization of signal processing applications on multicore platforms. HMBE models applications using an adapted form of the windowed synchronous dataflow (WSDF) model of computation, and schedules these application models onto the given multicore platform in a lock-free and race-condition-free manner. HMBE provides an automated design process that does not require the designer to have detailed knowledge of parallel programming.

In HMBE, we also separated the runtime model, called the acyclic precedence expansion graph (APEG) representation, from the high-level WSDF-based application model. This separation allows the designer to work with a more compact and intuitive

model based on WSDF semantics, while the runtime engine operates on a more detailed model that supports efficient, automated dynamic scheduling analysis. The process of converting from the designer-oriented WSDF model to the runtime-oriented APEG model is fully automated.

We demonstrated the effectiveness of HMBE using two distinct application case studies: (1) a large-scale image stitching application, which is data- and compute-intensive and has abundant data parallelism along with complicated data dependencies, and (2) a background subtraction application, which contains actors with internal states and many sequential operations. In both sets of experiments, our results showed that the HMBE scheduler effectively utilized the available CPU resources and significantly boosted application performance.

Second, we developed HMBE-Integrated-HTGS (HI-HTGS), a software tool that implements and optimizes multicore signal processing systems using the runtime system of HTGS. In this work, we integrated the model-based design capability of HMBE with the application programming interface (API) of HTGS. HI-HTGS utilizes the powerful optimization techniques developed in the HMBE analysis engine and automates the complicated steps involved in design and implementation of bookkeeper rules for HTGS.

We resolved the mismatches between the execution environments of HMBE and HTGS though a novel hierarchical scheduler architecture. The hierarchical scheduler is composed of a group of local schedulers and a single global scheduler. We compared the execution time performance and program size (lines of code) between HI-HTGS-based and HTGS-based implementations of the same application. The comparison was performed using a case study involving a data- and computation-intensive image stitching

application. The experimental results demonstrated that the HI-HTGS version achieved similar performance as the HTGS version on a diverse set of hardware platforms. However, the HI-HTGS version required substantially fewer lines of code compared to the HTGS version, and could therefore be developed and maintained more efficiently.

Finally, we developed the CPU-GPU Model-Based Engine (CGMBE), which provides several important enhancements that go beyond the capabilities of both HMBE and HI-HTGS:

- CGMBE extends the supported class of target architectures from multicore platforms to hybrid CPU-GPU platforms.
- CGMBE generalizes the runtime APEG representation to the form of an unfolded single-rate synchronous dataflow (SRSDF) model, and replaces the HMBE runtime engine with an executor pattern, which provides greater flexibility and efficiency compared to the HMBE runtime engine.
- CGMBE generalizes the application modeling framework to include globally-static, locally-dynamic patterns (GSLD) patterns.
- CGMBE automates the tedious step of coordinating data motion between host (CPU) memory and device (GPU) memory by automatically inserting memory copy actors between adjacent actors that are mapped to different computation devices.
- The executor pattern introduced in CGMBE frees designers from making difficult choices involving the number of CPU threads or GPU streams to be assigned to each dataflow actor. This aspect of CGMBE makes it possible to extend CGMBE

to more complicated computation platforms in a modular fashion by deriving and plugging in new executors.

- CGMBE includes a useful tool for tracing execution time performance and relating the extracted performance information to the high-level, model-based application representation. This tracing tool is useful to designers in iterating on designs and tuning performance, and results in minimal execution time overhead for the instrumented implementations.

We demonstrated the utility of CGMBE through extensive experiments involving a Vertex Component Analysis (VCA) application for linear unmixing for hyper-spectral images. The experimental results demonstrated that CGMBE achieves significant improvements in both execution time and memory usage.

7.2 Future Work

In this section, we explore directions for future work from four aspects: (1) extending the supported application modeling methods to enable greater system-level modeling flexibility; (2) extending the supported back-end executor to support a more diverse variety of hardware platforms; (3) developing automated task partitioning capabilities for platforms with multiple, possibly distributed CPU-GPU combinations; and (4) enabling work stealing between different executors to enhance dynamic load balancing across computation resources.

Fig. 7.1 illustrates the architecture of the CGMBE framework with mature components represented as grey blocks and components for future work represented as white

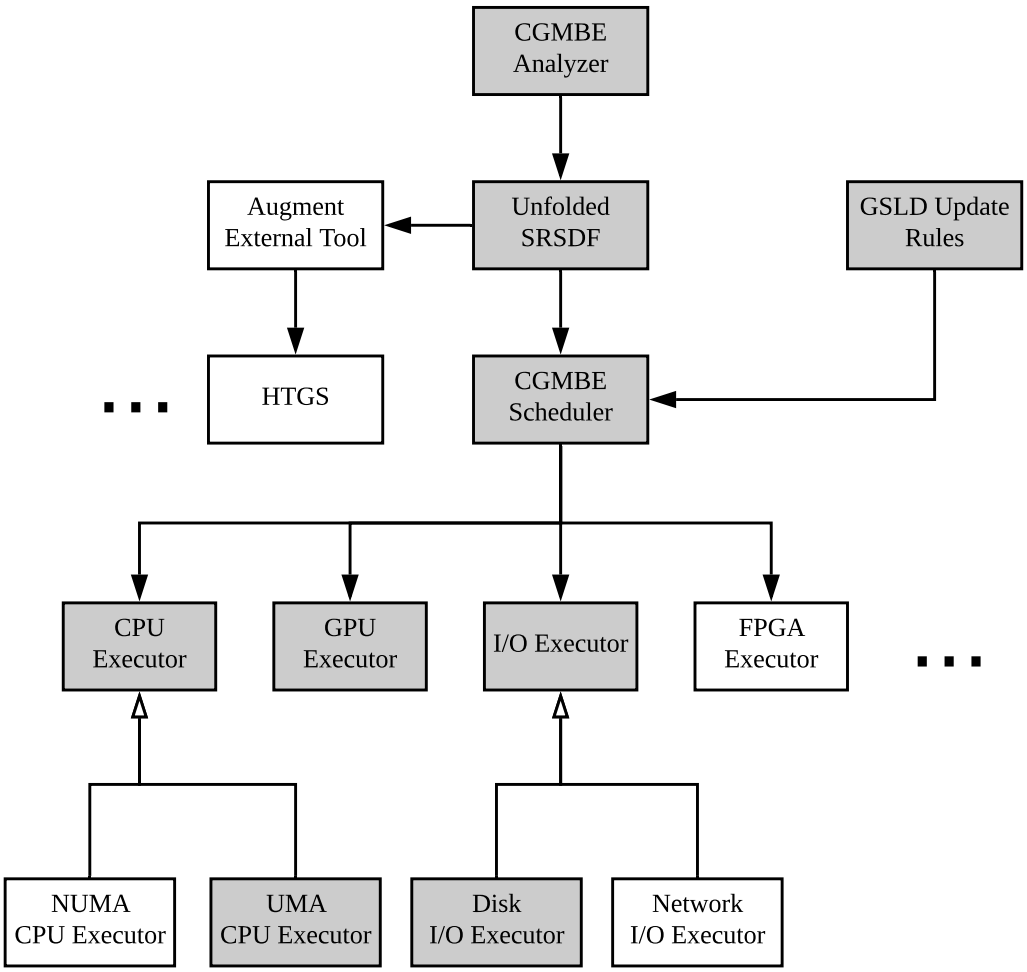


Figure 7.1: An illustration of future work to extend the CGMBE framework.

blocks.

7.2.1 Hardware Platforms

Currently supported devices in the CGMBE framework include CPUs, GPUs, and I/O devices. Each of these three categories of devices corresponds to a distinct executor type in CGMBE. Additional categories of hardware platforms and devices can be supported in CGMBE by developing corresponding new types of executors if the new type of component is not well-supported by any of the existing executor types. In addition to identifying an existing executor type or defining a new type, research is also in general needed to implement and optimize support for the new component type within the selected type of executor.

The library of supported executors in CGMBE is grouped into different classes of executors. Different executor types having the class share the same interface but may have different implementations. For example, there is currently one CPU executor type in the CGMBE framework, which is the Uniform Memory Access (UMA) model CPU executor. An interesting direction for future research is to add an executor type for a Nonuniform Memory Access (NUMA) model, which is useful to improve performance on platforms in which multiple CPU sockets do not uniformly share the available memory [64]. In this case, we envision that the UMA CPU executor and NUMA executor would provide different implementations of the CPU executor class.

The addition of a new executor type in this way involves implementing the new executor as a first step, and then as a more complex step, examining how different aspects

of the design tool components within CGMBE can be extended to optimize the integrated use of the new executor type.

In addition to the NUMA CPU executor type, other examples of useful new executor types to investigate are an FPGA executor (probably through the definition of a new executor class), and a Network I/O executor (probably as a new implementation of the existing I/O executor class).

7.2.2 Task Partitioning

A current limitation of the GPU executor in the CGMBE framework is that it only supports a single GPU device. To use multiple GPUs in CGMBE, the designer must manually partition GPU-targeted actors in the application model across the different GPUs and instantiate a separate executor for each GPU. This approach has a number of drawbacks. First, it may be very difficult for designers to derive efficient partitioning solutions by hand, especially when the complexity of the application graph becomes high. Second, designers are constrained by CGMBE to mapping each GPU-targeted actor to a single GPU. Different firings of the same actor cannot be spread across different GPUs. As a result, when the number of GPU-targeted actors is less than the number of available GPU devices, it is impossible to make full use of the GPUs. Third, the need for hand-developed partitioning of GPU-actors makes it time consuming and error-prone to retarget designs across different platforms.

This motivates investigating new methods in the CGMBE analyzer for automated partitioning of actors across multiple GPUs, and for allowing different firings of a given

actor to be distributed across multiple GPUs. The partitioning approach could operate on the intermediate representation, the unfolded SRSDF model, in the CGMBE analyzer. Many partitioning techniques have been developed for dataflow graphs and task graphs. Examples of this body of work are Bhattacharyya et al. [3], Shen et al. [65], and Wang et al. [66]. However, the models and design flow of CGMBE result in novel constraints under which partitioning solutions must operate, as well as novel capabilities in the compile-time and run-time engine that can be exploited. As such, maximizing the effectiveness of task partitioning for CGMBE is an interesting direction for further investigation.

7.2.3 Work Stealing

In the experiments involving the VCA application presented in Section 6.4, we found that the performance bottleneck of the CGMBE-based, CPU-GPU implementation was alternating back and forth between the CPU- and GPU-targeted subsystems. CPU and GPU workloads that remain more balanced throughout application execution can in general lead to significant improvements in performance. However, due to uncertainties in actor execution times and interprocessor communication times for the targeted class of platforms, it is in general not possible to keep execution times balanced through purely compile-time analysis.

Therefore, another interesting direction for future work is to investigate runtime load balancing mechanisms for executors in CGMBE. One promising approach in this direction is the application of work stealing concepts [67] among different executor types and executor classes to perform runtime load balancing between different types of de-

vices, including CPUs and GPUs. Maintaining data locality is a major challenge when applying work stealing techniques to CGMBE. The locality-guided work stealing algorithm introduced in [68] provides an interesting starting point for addressing this problem.

Bibliography

[1] T. Blattner, W. Keyrouz, M. Halem, M. Brady, and S. S. Bhattacharyya, "A hybrid task graph scheduler for high performance image processing workflows," in *Proceedings of the IEEE Global Conference on Signal and Information Processing*, 2015, pp. 634–637.

[2] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, pp. 773–799, May 1995.

[3] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*, Springer, third edition, 2019.

[4] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, CRC Press, second edition, 2009, ISBN:1420048015.

[5] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, May 2006.

[6] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[7] J. Keinert, C. Haubelt, and J. Teich, "Modeling and analysis of windowed synchronous algorithms," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, May 2006.

[8] C. Hsu, M. Ko, and S. S. Bhattacharyya, "Software synthesis from the dataflow interchange format," in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, Dallas, Texas, September 2005, pp. 37–49.

[9] J. Wu, T. Blattner, W. Keyrouz, and S. S. Bhattacharyya, "Model-based dynamic scheduling for multicore implementation of image processing systems," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, Lorient, France, October 2017, pp. 1–6.

[10] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, second edition, 1999.

[11] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S. Nikolopoulos, "A taxonomy of task-based parallel programming technologies for high-performance computing," *The Journal of Supercomputing*, vol. 74, no. 4, pp. 1422–1434, Apr 2018.

[12] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[13] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[14] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, et al., "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[15] Jairo Balart, Alejandro Duran, Marc Gonzàlez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta, "Nanos Mercurium: a research compiler for OpenMP," in *Proceedings of the European Workshop on OpenMP*, 2004, vol. 8, p. 56.

[16] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov 2012, pp. 1–11.

[17] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, Nov 2013.

[18] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julien Langou, Piotr Luszczek, and Stanimire Tomov, "The impact of multicore on math software," in *Applied Parallel Computing. State of the Art in Scientific Computing*, Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Waśniewski, Eds., Berlin, Heidelberg, 2007, pp. 1–10, Springer Berlin Heidelberg.

[19] Iain Duff and Florent Lopez, "Experiments with sparse Cholesky using a parametrized task graph implementation," in *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Jack Dongarra, Ewa Deelman, and Konrad Karczewski, Eds., Cham, 2018, pp. 197–206, Springer International Publishing.

[20] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen, "LAPACK: A portable linear algebra library for high-performance computers," in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Los Alamitos, CA, USA, 1990, Supercomputing '90, pp. 2–11, IEEE Computer Society Press.

[21] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers," in *[Proceedings 1992] The Fourth Symposium on the Frontiers of Massively Parallel Computation*, Oct 1992, pp. 120–127.

[22] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," *Journal of Physics: Conference Series*, vol. 180, pp. 012037, jul 2009.

[23] Horizon 2020 FET-HPC project, "Parallel numerical linear algebra for extreme scale systems," `http://www.nlafet.eu`, visited on July 31, 2019.

[24] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J. Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17 – 35, 2015.

[25] J. Eker and J. W. Janneck, "Dataflow programming in CAL — balancing expressiveness, analyzability, and implementability," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, 2012, pp. 1120–1124.

[26] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet, "Orcc: multimedia development made easy," in *Proceedings of the ACM International Conference on Multimedia*, 2013, pp. 863–866.

[27] M. Pelcat, P. Menuet, S. Aridhi, and J.-F. Nezan, "Scalable compile-time scheduler for multi-core architectures," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2009, pp. 1552–1555.

[28] 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," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2013, pp. 519–530.

[29] F. Palumbo, N. Carta, and L. Raffo, "The multi-dataflow composer tool: A runtime reconfigurable HDL platform composer," in *Proceedings of the Conference on Design and Architectures for Signal and Image Processing*, 2011.

[30] S. Lin, Y. Liu, K. Lee, L. Li, W. Plishker, and S. S. Bhattacharyya, "The DSPCAD framework for modeling and synthesis of signal processing systems," in *Handbook of Hardware/Software Codesign*, S. Ha and J. Teich, Eds., pp. 1–35. Springer, 2017.

[31] E. A. Lee and S. Ha, "Scheduling strategies for multiprocessor real time DSP," in *Proceedings of the Global Telecommunications Conference*, 1989, vol. 2, pp. 1279–1283.

[32] Y. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *Journal of the Association for Computing Machinery*, vol. 31, no. 4, pp. 406–471, December 1999.

[33] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[34] Jiahao Wu, Timothy Blattner, Walid Keyrouz, and Shuvra S. Bhattacharyya, "Model-based dynamic scheduling for multicore signal processing," *Journal of Signal Processing Systems*, Oct 2018.

[35] P. K. Murthy and E. A. Lee, "Multidimensional synchronous dataflow," *IEEE Transactions on Signal Processing*, vol. 50, no. 8, pp. 2064–2079, August 2002.

[36] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, February 1996.

[37] Google, Inc., "Protocol buffers," `https://developers.google.com/protocol-buffers/`, visited on April 26, 2017.

[38] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual PAP/CDR Edition*, Addison-Wesley, 2001.

[39] J. Siek, L.-Q. Lee, and A. Lumsdaine, "The Boost Graph Library (BGL)," `http://www.boost.org/doc/libs/1_66_0/libs/graph/doc/index.html`, Last access: 2018-03-09.

[40] H. Printz, *Automatic Mapping of Large Signal Processing Systems to a Parallel Machine*, Ph.D. thesis, School of Computer Science, Carnegie Mellon University, May 1991.

[41] C. D. Kuglin and D. C. Hines, "The phase correlation image alignment method," in *Proceedings of the International Conference on Cybernetics and Society*, 1975, pp. 163–165.

[42] H. Li, K. Sudusinghe, Y. Liu, J. Yoon, M. van der Schaar, E. Blasch, and S. S. Bhattacharyya, "Dynamic, data-driven processing of multispectral video streams," *IEEE Aerospace & Electronic Systems Magazine*, vol. 32, no. 7, pp. 50–57, 2017.

[43] V. C. Coffey, "Multispectral imaging moves into the mainstream," *Optics & Photonics News*, pp. 18–24, apr 2012.

[44] Y. Benezeth, D. Sidibé, and J. B. Thomas, "Background subtraction with multispectral video sequences," in *Proceedings of the Workshop on Non-classical Cameras, Camera Networks and Omnidirectional Vision*, 2014.

[45] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, "Real-time computer vision with OpenCV," *Communications of the ACM*, vol. 55, no. 6, 2012.

[46] E. A. Lee, "Recurrences, iteration, and conditionals in statically scheduled block diagram languages," in *Proceedings of the International Workshop on VLSI Signal Processing*, 1988.

[47] J. Wu, T. Blattner, W. Keyrouz, and S. S. Bhattacharyya, "A design tool for high performance image processing on multicore platforms," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Dresden, Germany, March 2018, pp. 1304–1309.

[48] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. 36, no. 1, pp. 24–35, 1987.

[49] Paul Hamill, *Unit Test Frameworks*, O'Reilly & Associates, Inc., 2004.

[50] G. R. Gao, R. Govindarajan, and P. Panangaden, "Well-behaved programs for DSP computation," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, March 1992.

[51] J. T. Buck and E. A. Lee, "Scheduling dynamic dataflow graphs using the token flow model," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1993.

[52] Advanced Micro Devices, Inc., "AMD EPYC 7002 series datasheet," `https://www.amd.com/system/files/documents/AMD-EPYC-7002-Series-Datasheet.pdf`, 2019, Last-access: 2019-09-06.

[53] José MP Nascimento and José MB Dias, "Vertex component analysis: A fast algorithm to unmix hyperspectral data," *IEEE transactions on Geoscience and Remote Sensing*, vol. 43, no. 4, pp. 898–910, 2005.

[54] A. Plaza, P. Martinez, R. Perez, and J. Plaza, "A quantitative and comparative analysis of endmember extraction algorithms from hyperspectral data," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 42, no. 3, pp. 650–663, March 2004.

[55] Michael E. Winter, "N-findr: an algorithm for fast autonomous spectral end-member determination in hyperspectral data," 1999, vol. 3753, pp. 3753 – 3753 – 10.

[56] Kailash C. Tiwari, Manoj Kumar Arora, Dharmendra P. Singh, and Deepti Soni Yadav, "Military target detection using spectrally modeled algorithms and independent component analysis," *Optical Engineering*, vol. 52, pp. 52 – 52 – 12, 2013.

[57] "VCA algorithm (unmix hyperspectral data)," 2019, http://www.lx.it.pt/~bioucas/code.htm.

[58] Gaël Guennebaud, Benoît Jacob, et al., "Eigen v3," http://eigen.tuxfamily.org, 2010, Last access: 2019-02-02.

[59] "High level introduction to HDF5," Tech. Rep., The HDF Group, September 2016, Last access: 2019-09-06.

[60] Y. Liu, L. Barford, and S. S. Bhattacharyya, "Generalized graph connections for dataflow modeling of DSP applications," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, Cape Town, South Africa, October 2018, pp. 275–280.

[61] Google, Inc., "The trace event profiling tool," https://www.chromium.org/developers/how-tos/trace-event-profiling-tool, visited on August 15, 2019.

[62] Google, Inc., "Trace event format," https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview#, visited on August 15, 2019.

[63] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, June 1982.

[64] Zoltan Majo and Thomas R. Gross, "Memory system performance in a NUMA multicore multiprocessor," in *Proceedings of the 4th Annual International Conference on Systems and Storage*, New York, NY, USA, 2011, SYSTOR '11, pp. 12:1–12:10, ACM.

[65] C. Shen, W. L. Plishker, D. Ko, S. S. Bhattacharyya, and N. Goldsman, "Energy-driven distribution of signal processing applications across wireless sensor networks," *ACM Transactions on Sensor Networks*, vol. 6, no. 3, 2010, Article No. 24, 32 pages.

[66] Minjie Wang, Chien-chin Huang, and Jinyang Li, "Supporting very large models using automatic dataflow graph partitioning," in *Proceedings of the Fourteenth EuroSys Conference 2019*, New York, NY, USA, 2019, EuroSys '19, pp. 26:1–26:17, ACM.

[67] Robert D. Blumofe and Charles E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sept. 1999.

[68] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe, "The data locality of work stealing," in *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, New York, NY, USA, 2000, SPAA '00, pp. 1–12, ACM.