AFRL-RI-RS-TR-2014-092

DESIGN TOOLS FOR ACCELERATING DEVELOPMENT AND USAGE OF MULTI-CORE COMPUTING PLATFORMS

UNIVERSITY OF MARYLAND

APRIL 2014

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2014-092 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
STANLEY LIS
Work Unit Manager
Intelligence

/ S /
MARK H. LINDERMAN,Technical Advisor
Computing & Communications Division
Information Directorate

REPORT DOCUMENTATION PAGE			*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* APRIL 2014	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* NOV 2010 – NOV 2013

4. TITLE AND SUBTITLE DESIGN TOOLS FOR ACCELERATING DEVELOPMENT AND USAGE OF MULTI-CORE COMPUTING PLATFORMS	**5a. CONTRACT NUMBER** FA8750-11-1-0062
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 62788F
6. AUTHOR(S) Shuvra S. Bhattacharyya	**5d. PROJECT NUMBER** T2MC
	5e. TASK NUMBER UN
	5f. WORK UNIT NUMBER ML
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Maryland, College Park Office of Research Administration & Advancement 3112 Lee Building MD-005 College Park, MD 20742	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITB 525 Brooks Road Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2014-092

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Multicore computing technologies are critical to high performance embedded systems. Such technologies are advancing rapidly in terms of the diversity of available multicore platforms, and the scale and heterogeneity of computing resources available on multicore-equipped devices. However, development of high performance signal processing software for multicore computing platforms is a complex process. Due to this complexity, designers face major limitations in effectively deploying high performance embedded solutions based on current design methodologies and tools. Key factors that complicate this process include challenges in exposing and exploiting application parallelism; heterogeneity and complex trade-offs among available multicore platforms; and the large scale of modern embedded software applications. To help designers experiment more effectively with alternative multicore software strategies, and to develop efficient and reliable embedded software implementations, this project has contributed systematic design methods for formal description of multiprocessor platforms, and optimized mapping of signal processing code blocks onto multiprocessor architectures.

15. SUBJECT TERMS
Multicore processors, signal processing, embedded systems, dataflow graphs.

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** STANLEY LIS
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	32	**19b. TELEPHONE NUMBER** (*Include area code*) N/A

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. SUMMARY

Multicore computing technologies are critical to high performance embedded systems. Such technologies are advancing rapidly in terms of the diversity of available multicore platforms, and the scale and heterogeneity of computing resources available on multicore-equipped devices. However, development of high performance signal processing software for multicore computing platforms is a complex process. Due to this complexity, designers face major limitations in effectively deploying high performance embedded solutions based on current design methodologies and tools. Key factors that complicate this process include challenges in exposing and exploiting application parallelism; heterogeneity and complex trade-offs among available multicore platforms; and the large scale of modern embedded software applications. To help designers experiment more effectively with alternative multicore software strategies, and to develop efficient and reliable embedded software implementations, this project has contributed systematic design methods for formal description of multiprocessor platforms, and optimized mapping of signal processing code blocks onto multiprocessor architectures.

The project has involved the following inter-related thrusts.

1. *Hardware description of multiprocessor systems*. We have developed a formal framework for efficiently capturing and analyzing the underlying hardware structures in multicore platforms at a high level of abstraction.
2. *Systematic mapping of code blocks onto hardware*. Building on our novel hardware description approach, we have developed algorithms and tools for systematically mapping signal processing code blocks onto multicore platforms.
3. *Customization and demonstration based on video surveillance and pattern recognition computations*. To validate, refine, and demonstrate our description and mapping approaches, we have experimented with key algorithms and application subsystems in the domains of video surveillance and pattern recognition.

2. INTRODUCTION

Modeling DSP applications through coarse-grain dataflow graphs is widespread in the digital signal processing (DSP) design community, and a variety of dataflow models have been developed for dataflow-based design. A growing set of DSP design tools support such dataflow semantics. Furthermore, Turing-complete DSP-oriented dataflow modeling approaches are available to provide for full expressibility within the dataflow framework.

In this project, we have advanced dataflow-based design methodologies for design and implementation of high-performance, multicore signal processing systems. We have applied dataflow concepts and methods to develop new methods for hardware description of multicore systems, and systematic mapping of code blocks onto hardware. We have demonstrated these new methods through extensive experimentation with key algorithms and application subsystems in the domains of video surveillance and pattern recognition.

In this project, we have built on our previous work on the dataflow interchange format (DIF) [Hsu 2005]. The DIF toolset provides a foundation for exploring techniques that bridge heterogeneous dataflow models and architectures. A critical need in transferring technology across these different dataflow-based tools and their targeted platforms is that of a standard, vendorindependent language and an associated package with intermediate representations and efficient implementations of dataflow analysis and optimization algorithms. DIF is designed for this purpose and is proposed to be a standard language for specifying and working with dataflow-based DSP applications across all relevant dataflow modeling approaches that are related to DSP system design.

In order to provide the DSP design industry with a convenient front-end to use DIF and the DIF package, automating the exporting and importing processes between DIF and design tools is an essential feature. Although problems related to exporting and importing are design-tool-specific, many practical implementation issues are quite common among different design tools. DIF and the associated DIF package have been designed to help reuse effort that is related to these common issues so that developers and users of design tools can focus on the novel features and unique constraints associated with their design problems.

The DIF framework therefore provides a useful foundation for developing novel high performance computing capabilities that address the performance and productivity challenges of high performance signal processing applications.

3. METHODS, ASSUMPTIONS, AND PROCEDURES

3.1. Dataflow Interchange Format

Development of high performance embedded signal processing software for multicore computing platforms is a complex process. Key factors that complicate this process include challenges in exposing and exploiting application parallelism; heterogeneity and complex trade-offs among available multicore platforms; and the large scale of modern embedded software applications [Bhattacharyya 2013]. To enable designers to experiment more effectively with alternative multicore software strategies, and to develop efficient and reliable embedded software implementations, development of new design tools is critical for capturing and accurately targeting the constraints of state-of-the-art multicore systems.

In this work, we have built on the Dataflow Interchange Format (DIF) Project [Hsu 2005], which is a core research focus of the Maryland DSPCAD Research Group [DSPCAD 2013] at the University of Maryland. The DIF Project provides a valuable infrastructure for developing, experimenting with, and integrating computer-aided design techniques for embedded signal processing systems. In this project, we have developed capabilities in the DIF package to demonstrate the techniques developed in this research, and provide a basis for integrating the techniques into practical design flows for optimized implementation of multicore signal processing software.

3.2. Dataflow Modeling for Signal Processing Systems

In recent years, dataflow models have become increasingly popular for design and implementation of signal processing systems (e.g., see [Bhattacharyya 2013]). Through their connections to computation graphs [Karp 1966] and Kahn process networks [Kahn 1974, Lee 1995], signal-processing-oriented dataflow models of computation (SDMs) build on a strong theoretical foundation. Additionally, through their natural correspondence to signal flow graphs, which are used widely by signal processing algorithm designers, SDMs provide an intuitive framework for high-level application modeling and programming. Dataflow methods and tools have been developed previously by the DoD. For example, the processing graph method (PGM) has been developed extensively at NRL [Stevens 1997], and has influenced industrial tools for high-performance embedded signal processing, such as the Autocoding Toolset from MCCI [Robbins 2002].

In SDMs, as in other forms of dataflow, applications are represented by directed graphs in which vertices, called *actors*, correspond to computational tasks, and each edge corresponds to a logical buffer that stores data values as they pass from the output of one actor to the input of another. In SDMs, dataflow actors can represent computations of arbitrary complexity as long as the interfaces of the computations conform to dataflow semantics. Intuitively, dataflow semantics in this context means that actors produce and consume data from their input and output edges, respectively, and each dataflow actor executes as a sequence of discrete units of computation, called *firings*, where each firing depends on some well-defined amount of data from the input edges of the associated actor. The restrictions and mathematical characterizations associated with this general notion of well-defined amount of data are primary aspects that distinguish alternative SDMs from one another.

Unlike dataflow architectures [Dennis 1975], which embed dataflow graphs directly in hardware, SDMs apply dataflow purely as a programming model. Because of the concurrency and other forms of application structure exposed by signal processing dataflow graph representations, SDM tools have significant potential for high-level optimization, and efficient retargetability across diverse target platforms. For these reasons, along with their portability and intuitive appeal, SDMs have been applied to many application areas and a variety of target platforms (e.g., see [Bhattacharyya 2013]. As system complexity and the diversity of components in embedded signal processing platforms increases, designers are expressing more and more types of behavior in dataflow languages, and even combining different dataflow models to describe individual applications.

3.3. The DIF Language (TDL)

The dataflow interchange format (DIF) is proposed as a standard approach for specifying and integrating arbitrary dataflow-based semantics for signal processing system design [Hsu 2005]. The DIF Package (TDP) [Hsu 2005, Shen 2012] is a software tool, developed in conjunction with DIF, for modeling and analyzing signal processing oriented dataflow graphs. The DIF Language (TDL) is an accompanying textual design language for high-level specification of signal-processing-oriented dataflow graphs. The TDL syntax for dataflow graph specification is designed based on dataflow theory and is independent of any specific design tool.

Because dataflow-oriented design tools in the signal processing domain are fundamentally based on actor-oriented design, TDL provides a syntax to specify tool-specific actor information, which ensures that TDP can extract all relevant information from a given design tool.

TDL is designed as a standard approach for specifying signal processing dataflow graphs at a high level of abstraction that is suitable for both programming and interchange. TDL provides a unique set of semantic features for specifying graph topologies, hierarchical design structure, dataflow-related design properties, and actor-specific information. TDP accompanies TDL, and provides a variety of intermediate representations, analysis techniques, and graph transformations that are useful for working with dataflow graphs that have been captured by TDL. For example, TDP includes a transformation that converts SDF (synchronous dataflow) representations into equivalent homogeneous SDF (HSDF) representations based on the algorithm introduced in [Lee 1987]. Such a transformation can in general expose additional concurrency that is not represented explicitly in the original SDF graph.

Compared to other design tools for representation and transformation of dataflow graphs — such as SysteMoC [Haubelt 2007], PeaCE [Kwon 2004], and stream-based functions [Kienhuis

2001] — a distinguishing feature of TDP is its support for representing and manipulating different specialized forms of dataflow semantics. This arises from the emphasis in TDL on recognizing a wide variety of important forms of dataflow semantics along with relevant modeling details that are required to meaningfully analyze those semantics. Due to this feature of TDP, its capabilities are highly complementary to those of existing dataflow-based frameworks. In particular, TDL and TDP can be used to capture and analyze, respectively, representations from many of these frameworks.

3.4. Synchronization Graphs

In this project, we have also leveraged our work on the synchronization graph modeling methodology (e.g., see [Sriram 2009]. Synchronization graphs provide formal methods for integrated representation and analysis of dataflow graph application behavior together with schedules (software structures for coordinating the execution of computational tasks across shared processors) that carry out execution of dataflow behaviors on multiprocessor hardware.

Intuitively, a synchronization graph can be viewed as a graph-theoretic representation of a self-timed multiprocessor schedule for a synchronous dataflow graph. Here, by synchronous dataflow, we mean a specialized variant of dataflow in which the number of tokens produced and consumed on each actor port is constant [Lee 1987], and by a self-timed multiprocessor schedule, we mean a multiprocessor schedule in which the assignment of actors to processors and the execution ordering of actors that are mapped to the same processor are fixed at compile time [Lee 1989]. Self-timed scheduling differs from static scheduling in that in self-timed scheduling, the actual time at which an actor invocation executes is determined at run time through appropriate synchronization. Such a scheduling model provides a framework for exploiting statically known application structure (through the compile time assignment and ordering), while providing for robustness when execution times are not known precisely or exhibit some amount of run-time variation.

Self-timed execution of synchronous dataflow specifications is widely used in parallel execution of signal processing applications (e.g., see [Bhattacharyya 2013]), and has provided a valuable starting point for our work in this project. Synchronization graphs provide a formal mathematical framework for analyzing, optimizing, and implementing this class of parallel signal processing systems.

4. RESULTS AND DISCUSSION

4.1. Objectives

Our project has centered on the following three key objectives, which involve the development of high level tools for formal description of multiprocessor platforms, systematic mapping of signal processing code blocks onto multiprocessor architectures, and in-depth application case studies in the domains of video surveillance and pattern recognition based on these tools.

1. Hardware Description of Multiprocessor Systems.
2. Systematic Mapping of Code Blocks onto Hardware.
3. Customization and Demonstration based on Video Surveillance and Pattern Recognition Computations.

4.2. Hardware Description of Multiprocessor Systems

4.2.1. CLT Design Flow

In this project, we have developed a novel design flow, based on core functional dataflow (CFDF) graphs [Plishker 2008a] for integrated simulation and implementation of signal, image, and video processing applications on state-of-the-art multicore platforms. Our design flow builds on our previously developed work on the lightweight dataflow (LWDF) [Shen 2010] and targeted dataflow interchange format (TDIF) tools [Shen 2012]. Because of its strong connections to CFDF, LWDF, and TDIF concepts, we refer to our new design flow as the *CFDF-LWDF-TDIF design flow* or *CLT design flow*. The CLT design flow provides a structured design process, based on formal dataflow-based models of computation, for efficient, high-confidence mapping of signal, image, and video processing systems onto multicore platforms. Our development of the CLT design flow has involved progress on all of the directions outlined in Section 4.1 (e.g., see Section 4.3.1 and Section 4.4.1 for more details).

4.2.2. DEIPS Methodology

In this project, we have developed a methodology, based on our recently developed dataflow schedule graph (DSG) model [Wu 2011], for design and implementation of embedded image processing systems. We refer to this as the DEIPS (DSG-based design and implementation of Embedded Image Processing Systems) methodology. Our developments on the DEIPS methodology are reported in [Wu 2013].

We have developed the DEIPS methodology in the context of a state-of-the-art multiprocessor system-on-chip (MPSoC) platform that is relevant in the embedded image processing domain — the Texas Instruments (TI) TMS320C6678L embedded multicore digital signal processor platform, using the TI TMS320C6678L Evaluation Module [TI 2012].

The underlying multicore processor of the targeted TMS320C6678L device contains eight cores that can run at 1 Gigahertz (GHz) each. Each core has L1 cache and L2 cache. The L1 cache is made up of separate parts for program and data, while the L2 cache provides unified space for program and data. The memory subsystem includes 512 Megabyte (MB) memory double data rate 3 (DDR3), which we employ as local memory, and 4 MB SRAM (MSMCSRAM), which we employ as shared memory among processors. Programmers can allocate memory space in the L2SRAM, MSMCSRAM or DDR3 through heaps that handle them. This platform provides significant flexibility to programmers and high level design tools to manage thread definitions, memory partitioning for threads, and inter-processor communication.

In the DEIPS methodology, we map each thread to a Sequential Dataflow Schedule Graph (SDSG) [Wu 2011]. The memory usage of each thread, which can be analyzed or simulated efficiently using the underlying SDSG model, is then used to determine the size of the corresponding block of partitioned memory. Additionally, we implement two pairs of special actors to provide more accurate DSG representations targeted to the multicore TI platform. These actors implement data synchronization and control synchronization, respectively, on the TI platform.

Inter-thread interactions are modeled as communication and synchronization actors between pairs of communicating SDSGs, and the resulting system-level schedules are modeled as CDSGs. When more than one processor is employed in the schedule, the CDSG model includes more than one SDSG and provides the nexus of the different SDSGs to coordinate and synchronize their concurrent execution.

The schedule control actors *snd* and *rec* are used to synchronize pairs of communicating SDSGs. Such implementation of interprocessor communication is complicated on the targeted TI platform since it requires handshaking involving heaps in shared memory, and creation of correct heap-based communication mechanisms. To simplify interprocessor communication from the designer's point of view, and to make such communication more reliable, we integrate the handshaking functionality into pre-defined, reusable, TI-targeted *snd* and *rec* actor components. Designers can then integrate such interprocessor communication components as needed in their DSG structures without having to bother with the low level implementation details associated with interprocessor communication on the targeted device. Each time a *snd* or *rec* is instantiated in a CDSG, the associated inter-processor communication is effectively instantiated and appropriately configured based on the surrounding CDSG context. Similarly, data synchronization is modeled in the CDSG through appropriate actors that implement the required communication functionality on the targeted TI device.

For more details on the DEIPS methodology and its application to Texas Instruments multicore digital signal processors, we refer the reader to [Wu 2013].

4.3. Systematic Mapping of Code Blocks onto Hardware

4.3.1. Mapping Methods in the CLT Design Flow

In the *system optimization* step of the CLT design flow (See Section 4.2.1), we enter the implementation phase, which is where TDIF comes into play. There are two main kinds of optimization techniques supported in the TDIF framework. One is *cross-platform implementation* for actor-level optimization, and the other is *scheduling and mapping* for system or subsystem optimization.

After we identify the bottleneck actors, cross-platform implementation of actors allows designers to efficiently experiment with alternative actor realizations on different kinds of platforms, such as graphic processing units (GPUs), multicore programmable digital signal processors (PDSPs), and field programmable gate arrays (FPGAs), to help derive a platform or mix of platforms that will be strategic in terms of the given design constraints (e.g., constraints involving cost, performance, and energy consumption). During this process, much of the code from the simulation phase can be reused. Only the functionality associated with selected actor modes (e.g., bottleneck modes of bottleneck actors) needs to be rewritten or selected from available platform-specific libraries.

The TDIF environment currently supports C-like programming languages — e.g., languages that are targeted to Central Processing Unit (CPU), GPU and multicore PDSP platforms. The GPU-based capabilities of TDIF are currently oriented towards NVIDIA GPUs, based on the Compute Unified Device Architecture (CUDA) programming language [NVIDIA 2007], which can be viewed as an extension of C. The multicore PDSP capabilities currently in TDIF are oriented towards Texas Instruments (TI) PDSP devices, and are interoperable with the multithreading libraries provided by TI [TI 2012].

TDIF also provides a library of First-In-First-Out (FIFO) implementations that are optimized for different platforms. These FIFOs all adhere to a common set of CFDF-based application programming interfaces (APIs) so that they can be integrated in a manner that is consistent with the CFDF graph model from the simulation phase. After simulation-mode FIFOs are mapped into platform-specific FIFOs, optimized actors can communicate in a manner that is efficient, and consistent with the designer's simulation model.

The scheduling strategy employed determines the execution order of the actors while the mapping process, which is typically coupled closely with scheduling, determines which resource each actor is executed on. TDIF provides the *generalized schedule tree* (*GST*) [Ko 2007] representation to facilitate implementation of and experimentation with alternative scheduling and mapping schemes for system optimization. GSTs are ordered trees with leaf nodes and internal nodes. An internal node of a GST in TDIF represents iteration control (e.g., a loop count) for an iteration construct that is to be applied when executing the associated subtree. On the other hand, a GST leaf node includes two pieces of information that are used to carry out individual actor firings — one is an actor of the associated dataflow graph, and the other is mapping information associated with the actor. The GST representation provides designers with a common interface through with topological information and algorithms for ordered trees can be applied to access and manipulate schedule elements.

Execution of a GST involves traversing the tree to iteratively enable (and then execute, if appropriate) actors that correspond to the schedule tree leaf nodes. Note that if actors are not enabled, the GST traversal simply skips their invocation. Subsequent schedule rounds (and thus subsequent traversals of the schedule tree) will generally revisit actors that were unable to execute in the current round.

For schedule construction in the implementation phase, the *CFDF graph decomposition* approach of [Plishker 2009] is integrated in the TDIF framework. This approach allows designers to decompose a CFDF graph into a set of SDF subgraphs. Each SDF subgraph can be scheduled by existing static scheduling algorithms, such as an APGAN-based scheduler [Bhattacharyya 1997] (APGAN stands for "acyclic pairwise grouping of adjacent nodes" — for more details, we refer the reader to [Bhattacharyya 1997]). The GST schedule trees that result from scheduling these SDF subgraphs are then systematically combined into a single, "execution-rate-balanced" GST using profiling and instrumentation techniques.

4.3.2. Hierarchical Mapping of Multidimensional Dataflow Specifications

In this project, we have developed a structured design method based on multidimensional synchronous dataflow (MDSDF) graphs for hierarchical mapping of DSP systems onto parallel architectures. Our developments on this new design method are reported in [Wang 2012].

MDSDF is a generalization of synchronous dataflow to multiple dimensions and provides an effective model for a variety of multidimensional DSP systems that have statically structured dataflow characteristics [Murthy 2002]. Our methods apply dataflow transformations to exploit data parallelism hierarchically for multidimensional dataflow graphs. Our methods provide a systematic approach for exposing and exploiting parallelism from multidimensional dataflow specifications across different levels of the specification hierarchy. We demonstrate our proposed new modeling techniques and design methods by applying them to optimize implementations on the NVIDIA graphics programming unit (GPU) programming model [NVIDIA 2012].

Recently-introduced data parallel programming environments emphasize support for exploiting multi-level or hierarchical parallelism, where parallelism is exploited programmatically at multiple levels of granularity. For example, CUDA [NVIDIA 2012] provides a two-level thread hierarchy, where a set of threads makes up a thread block, and multiple thread blocks form a grid. Such hierarchical support for representing parallelism is important for multidimensional signal processing applications, where parallelism exists in different forms at different levels of the design hierarchy (DH) (e.g., interframe, inter-block, and inter-pixel parallelism in video processing).

In this project, we have built on the MDSDF model of computation, and develop a design method to represent and apply parallelism hierarchically for multidimensional dataflow graphs.

Suppose that we have an N -level hierarchical parallel programming model (platform hierarchy) P, which we want to use to implement a given MDSDF graph G. For example, such a parallel programming model could be used as a target for code generation or could be used for an implementation that is derived from hand based on a functional reference (“golden model”) that is developed in terms of the MDSDF specification. We develop an N-level hierarchical dataflow graph transformation approach to achieve such a mapping from MDSDF to P. We refer to N in this context as the *platform depth*.

First, we introduce some definitions and notation related to hierarchical dataflow graphs. A *supernode* s in a dataflow graph G = (V, E) is an actor (i.e., $s \in V$) that is associated with a “nested dataflow graph” H(s), where execution of s in G corresponds to execution of H(s). In general, not all actor ports in H(s) are connected in H(s) (i.e., not all of them connect to edges within H(s)). The “unconnected actor ports” are referred to as the interface ports of H(s), and these ports are in one-to-one correspondence with the ports of actor s.

If G is the “top” of the DH (i.e., G is not encapsulated by a supernode in another graph), then we say that the nesting level (or simply level) of G, denoted λ(G), is 1. Similarly, for each supernode s in G, λ(H(s)) = 2; for each supernode t in any of these H(s)’s, λ(H(t)) = 3, and so on.

The DHs in our model are non-overlapping, which means that for all supernodes within a DH (i.e., across all levels), their corresponding nested dataflow graphs do not share any actors or edges. Furthermore, we assume that these DHs are finite, which means that the levels (λ values) are all bounded.

We refer to the maximum λ value in a DH D as the depth δ of D. For each $i \in \{1, 2, \cdots, \delta\}$, we denote by Li the set of all actors that are “at level i”. That is, L1 = V, and for i = 2, 3, . . . , δ,

$$Li = \cup\{Vh(s) | \lambda(H(s)) = i\}, \qquad (1)$$

where Vh(s) denotes the set of actors in the nested dataflow graph H(s).

DHs in our decomposition approach can be constructed by designers as they explore alternative methods to structure the hierarchies such that they map efficiently into the parallelism hierarchy supported by the targeted platform. The key constraint in construction of a DH D is that the depth of each candidate DH should equal the platform depth.

We develop a systematic method, called multidimensional DH mapping, to specify and map these DHs into hierarchies of smaller graphs, which can be mapped to successively lower levels of the targeted platform hierarchy. Figure 1 illustrates this approach for an MDSDF graph. The designer can construct the DHs bottom-up or top-down. At each *i*th level (*i* > 1) of the DH, one or more groups (clusters) of connected actors are combined into units that are viewed as individual supernodes from level (*i*−1). Groups of actors, including supernodes, which are contained within such clusters are then scheduled together by adapting techniques for SDF-based and MDSDF-based clustered graph analysis and scheduling [Bhattacharyya 1996, Murthy 2002].

Use of these techniques to systematically derive production and consumption tuples associated with actors at different levels of the design hierarchy, as well as firing vectors, which determine the relative rates at which different actors in a cluster execute, is illustrated in Figure 1.

4.4. Customization and Demonstration based on Video Surveillance and Pattern Recognition Computations

4.4.1. Gaussian Filtering

In this section, we present a discussion of customizations, demonstrations and associated experimental findings in connection with the CFDF-LWDF-TDIF (CLT) design flow, which is discussed in Section 4.2.1. To demonstrate the CLT design flow, we experiment with an image processing application centered on Gaussian filtering. Two-dimensional Gaussian filtering is a common kernel in image processing that is used for preprocessing. Gaussian filtering can be used to denoise an image or to prepare for multiresolution processing. A Gaussian filter is a filter whose impulse response is a Gaussian curve, which in two dimensions resembles a bell.

For filtering in digital systems, the continuous Gaussian filter is sampled in a window and stored as coefficients in a matrix. The filter is convolved with the input image by centering the matrix on each pixel, multiplying the value of each entry in the matrix with the appropriate pixel, and then summing the results to produce the value of the new pixel. This operation is repeated until the entire output image has been created.

The size of the matrix and the width of the filter may be customized according to the application. A wide filter will remove noise more aggressively but will smoothen sharp features. A narrow filter will have less of an impact on the quality of the image, but will be correspondingly less effective against noise.

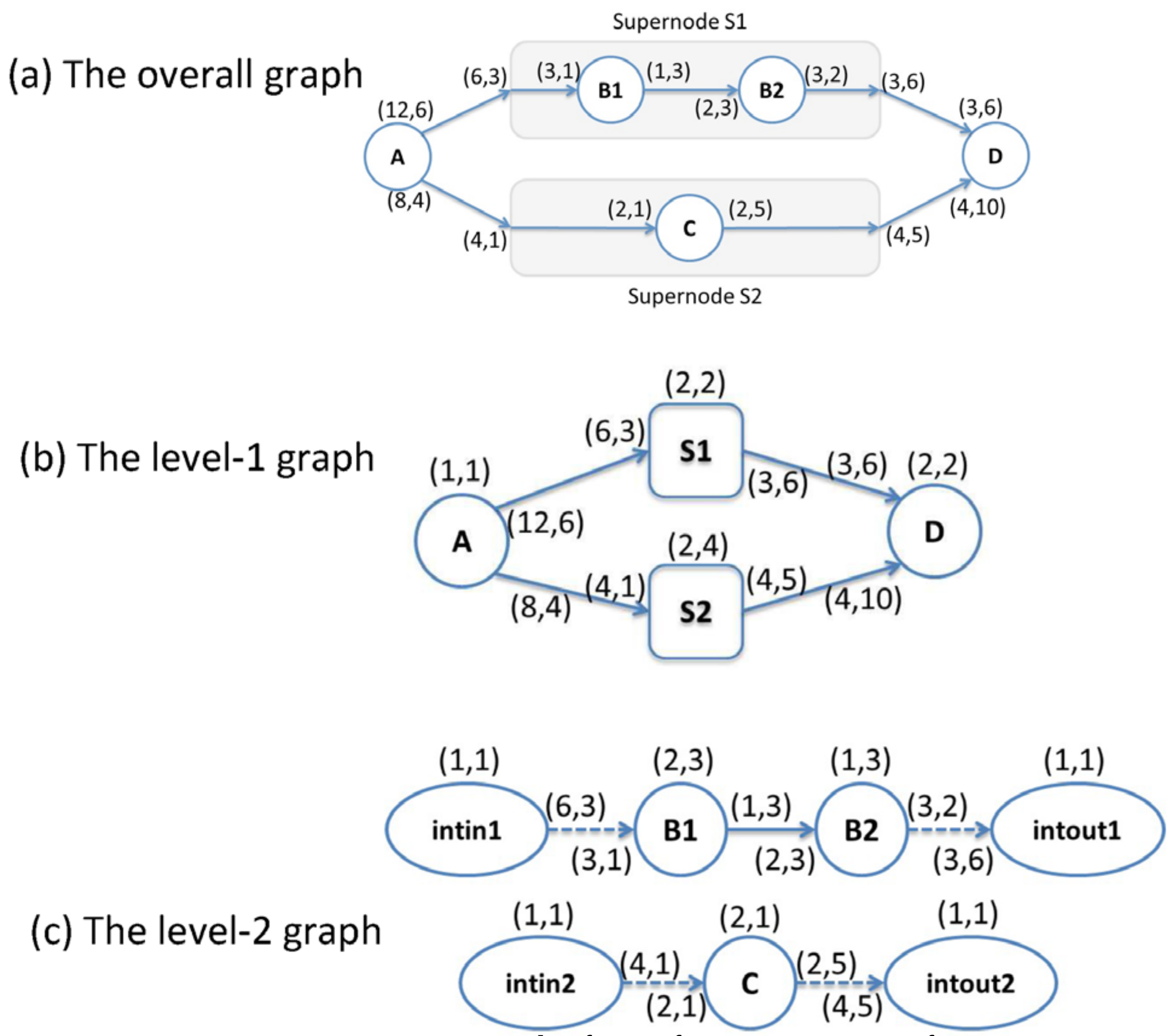

Figure 1. An example of a DH for an MDSDF specification.

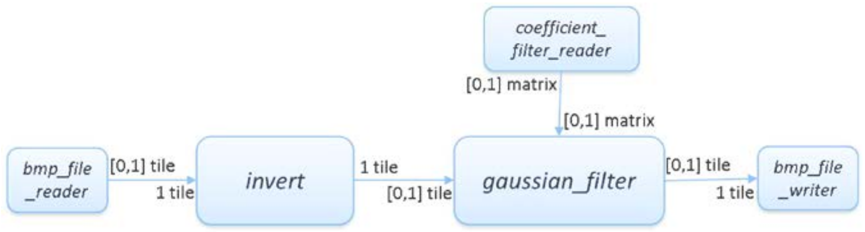

Figure 2. Dataflow graph of an image processing application for Gaussian filtering

Figure 2 shows a simple application based on Gaussian filtering. It reads bitmap files in tile chunks, inverts the values of the pixels of each tile, runs Gaussian filtering on each inverted tile, and then writes the results to an output bitmap file. The main processing pipeline is single-rate in terms of tiles, and can be statically scheduled, but after initialization and end-of-file behavior is modeled, there is conditional dataflow behavior in the application graph, which is represented by square brackets in the figure.

Such conditional behavior arises, first, because the Gaussian filter coefficients are programmable to allow for different standard deviations. The coefficients are set once per image — `coefficient_filter_reader` produces a coefficient matrix for only the first firing. To correspond to this behavior, the `gaussian_filter` actor consumes the coefficient matrix only once, and each subsequent firing processes tiles. Such conditional firing also applies to `bmp_file_reader`, which produces tiles until the end of the associated file is reached.

As shown in Figure 2 our dataflow graph of the image processing application for Gaussian filtering is specified as a CFDF graph. The graph includes five actors:

- `bmp_file_reader`,
- `coefficient_filter_reader`,
- `invert`,
- `gaussian_filter`, and
- `bmp_file_writer`.

We first use the LWDF programming methodology, integrated with the C language, to construct the system for simulation.

It should also be noted that the tiles indicated in Figure 2 do vary somewhat between edges. Gaussian filtering applied to tiles must consider a limited neighborhood around each tile (called a *halo*) for correct results. Therefore, tiles produced by `bmp_file_reader` overlap, while the halo is discarded after Gaussian filtering. As a result, non-overlapping tiles form the input to `bmp_file_writer`.

4.4.1.1. Simulation

In our design, the `bmp_file_reader` actor is specified using two CFDF modes, and one output FIFO. The two modes are the `process` mode and the `inactive` mode. It is the actor programmer's responsibility to implement the functionality of each mode. In the `process` mode, the `bmp_file_reader` reads image pixels of a given tile and the corresponding header information from a given bitmap file, and produces them to its output FIFOs. Then the actor returns the `process` mode as the mode for its next firing. This continues for each firing until all of the data has been read from the given bitmap file. After that, the actor returns the `inactive` mode, which is a *terminal mode*. Arrival at a terminal mode indicates that the actor cannot be fired anymore until its current mode is first reset externally (e.g., by the enclosing scheduler).

The `coefficient_filter_reader` actor is also specified in terms of two modes and one output FIFO. The two modes are again labeled as the `process` mode and the `inactive` mode, and again, the `inactive` mode is a terminal mode. On each firing when it is not in the `inactive` mode, the `coefficient_filter_reader` actor reads filter coefficients from a given file, stores them into a *filter coefficient vector* (*FCV*) array, and produces the coefficients onto its output FIFO. The FCV *V* has the form

$$V=(sizeX,sizeY,c_0,c_1,\ldots,cn-1), \quad (4)$$

where *sizeX* and *sizeY* denote the size of the FCV represented in two dimensional format; each c_i represents a coefficient value; and $n = sizeX \times sizeY$. After firing, the actor returns the `process` mode if there is data remaining in the input file; otherwise, the actor returns the `inactive` mode.

The `bmp_file_writer` actor contains only a single mode and one input FIFO. The single mode is called the `process` mode. Thus, the actor behaves as an SDF actor. On each firing, the `bmp_file_writer` actor reads the processed image pixels of the given tile and the corresponding header information from its input FIFOs, and writes them to a bitmap file, which can later be used to display the processed results. The actor returns the `process` mode as the next mode for firing.

The `gaussian_filter` actor contains one input FIFO, one output FIFO and two modes: the store coefficients (`STC`) mode and the `process` mode. On each firing in the `STC` mode, the `gaussian_filter` actor consumes filter coefficients from its coefficient input FIFO, caches them inside the actor for further reference, and then returns the `process` mode as the next mode for firing. In the `process` mode, image pixels of a single tile will be consumed from the tile input FIFO of the actor, and the cached filter coefficients will be applied to these pixels. The results will be produced onto the tile output FIFO. The actor then returns the `process` mode as the next mode for firing. To activate a new set of coefficients, the actor must first be reset, through external control, back to the `STC` mode.

The `invert` actor also contains a single mode called the `process` mode, and contains one input FIFO and one output FIFO. Because it has only one mode, it can also be viewed as an SDF actor. On each firing, the `invert` actor reads the image pixels of the given tile from its input FIFOs, inverts the color of the image pixels, and writes the processed result to its output FIFO. The actor always returns the `process` mode as the next mode for firing.

After designing the actors, as described above, we connect the actors with the appropriate FIFOs. For our simulation setup, we use 256x256 images decomposed into 128x128 tiles, and filtered with different sizes of matrices for Gaussian filter coefficients. The CFDF canonical scheduler [Plishker 2008b] is used to run the simulation on 3 GHz Intel Xeon processors. The profiling results are reported in Table 1. As can be observed from this table, increases in the matrix size lead to increases in the processing time for the Gaussian filter and the overall application. Furthermore, the Gaussian filter actor accounts for most of the processing time in the application in all cases. Thus, if we can optimize the Gaussian filter actor, the performance of the overall application will be enhanced.

Table 1. Execution time of the Gaussian filter actor (GF) and the Gaussian filtering application (App) during simulation.

Filter size	5X5	11X11	21X21	25X25	37X37
GF. CPU (ms)	50	280	1080	1540	3310
App. CPU (ms)	70	295	1100	1550	3340
Percentage	**71.4%**	**95%**	**98.2%**	**99.3%**	**99.1%**

4.4.1.2. Implementation

From the experiments discussed in the previous section, we identified the bottleneck actor to be the Gaussian filtering actor. To improve the performance of this actor, we apply the cross-platform implementation features of TDIF. In particular, we use TDIF to experiment with a new version of the implementation in which the Gaussian filtering actor is executed on a graphics processing unit (GPU).

GPUs provide a class of high performance computing platforms that provide high peak throughput processing for certain kinds of regularly structured computations [Owens 2008]. Typically, a GPU architecture is structured as an array of hierarchically connected cores. Cores tend to be lightweight as the GPU will instantiate many of them to support massively parallel graphics computations. Some of the memories are small and scoped for access to small numbers of cores, but can be read or written in one or just a few cycles. Other memories are larger and accessible by more cores, but at the cost of longer read and write latencies.

Using TDIF, we explore the use of GPUs to accelerate the `gaussian_filter` actor. We employ an NVIDIA GTX 285 GPU and employ the CUDA programming environment to specify the internal functionality of the `gaussian_filter` actor for GPU acceleration. This CUDA-based actor implementation is integrated systematically into the overall application-level CFDF graph through the TDIF design environment. We apply actor-level vectorization to exploit data parallelism within the actor on the targeted GPU.

Fundamentals of vectorized execution for dataflow actors have been developed by Ritz [Ritz 1993] and explored further by Zivojnovic [Zivojnovic 1994], Lalgudi [Lalgudi 2000], and Ko [Ko 2008]. In such vectorization, multiple firings of the same actor are grouped together for execution to reduce the rate of context switching, enhance locality, and improve processor pipeline utilization. On GPUs, groups of vectorized firings can be executed concurrently to achieve parallel processing across different invocations of the associated actor. Each instance of a “vectorized actor” may be mapped to an individual thread or process, allowing the replicated instances to be executed in parallel.

An application developer may consider vectorization within and across actors while writing kernels for CUDA acceleration. In TDIF, the actor interface need not change as the vectorization degree changes, which makes it relatively easy for designers to start with the programming framework provided by CUDA and wrap the resulting vectorized kernel designs in individual modes of an actor for integration at the dataflow graph level.

In the GPU-targeted version of our Gaussian filtering application, a CUDA kernel is developed to accelerate the core Gaussian filtering computation (the `process` mode), and each thread is assigned to a single pixel, which leads to a set of parallel independent tasks. The threads are assembled into blocks to maximize data reuse. Each thread uses the same matrix for application to the local neighborhood, and there is significant overlap in the neighborhoods of the nearby pixels. To this end, the threads are grouped by tiles in the image. Once the kernel is launched, threads in a block cooperate to load the matrix, the tile to be processed, and a surrounding neighborhood of points. The image load itself is vectorized to ensure efficient bursting from memory. Because CUDA recognizes the contiguous accesses across threads, the subsequent image processing operations induce vectorized accesses to global memory.

We use the same canonical scheduler in the GPU implementation that we used in the simulation phase. The performance of our Gaussian filtering application with and without GPU acceleration is compared to demonstrate the ability of our design flow to support cross-platform actor implementation exploration in a manner that is systematically coupled with the simulation-level application model. We use the same experimental setup — in terms of input and output images and overall dataflow graph structure — as used in the simulation. To accelerate the Gaussian Filtering actor, we applied an NVIDIA GTX 285 running CUDA 3.1 and compared the associated implementation to the simulation system. The measurement results are reported in Table 2.

Table 2. Execution time of the Gaussian filter actor (GF) and the Gaussian filtering application (App) with and without GPU acceleration.

Filter size	5X5	11X11	21X21	25X25	37X37
GF. CPU (ms)	50	280	1080	1540	3310
GF. GPU (ms)	4.228	4.874	10.257	12.759	21.72
GF. **Speedup**	**11.83**	**57.45**	**105.29**	**120.70**	**152.39**
App. CPU (ms)	70	295	1100	1550	3340
App. GPU (ms)	70	80	140	115	130
App. **Speedup**	**1**	**3.69**	**7.86**	**13.48**	**25.69**

As shown in Table 2, our design flow provides a flexible and efficient transition from the simulation system to a GPU-accelerated implementation that has superior performance compared to the corresponding simulation design for these experiments. The actor-level speedup realized by this acceleration process is in the range of 10*X* to 100*X*. However, the application-level speedup levels, while still significant (up to 25*X* speedup), are consistently less than the corresponding actor-level speedup levels. This is due to factors such as context switch overhead and communication cost for memory movement, which are associated with overall schedule coordination in the application implementations.

4.4.2. Integral Histogram Computation

The integral histogram (IH) first maps pixels into a set of non-overlapping ranges (“bins”), and then performs a 2-D scan. Two scan orders, cross-weave and wavefront, are explored in [Bellens 2011]. The cross-weave scan processes the image in the first dimension (horizontal scan) followed by a scan in the second dimension (vertical scan). Instead of applying two passes, the wavefront scan propagates an anti-diagonal wavefront calculation as it operates through a single scan.

In our experiments, we incorporate use of a tiled image processing approach, where the image is separated into blocks (tiles) of neighboring pixels. Tiled approaches can be useful for GPU implementation to enhance parallel execution across multiple threads [NVIDIA 2012]. In particular, we explore in this case study a tiled integral histogram (TIH) approach for efficient mapping into GPU implementations.

The overall input image size for IH computation is denoted as (Iw × Ih) pixels, and the number of histogram bins is denoted as Nb. In TIH computation, an image is tiled as an (Nw × Nh) rectangular arrangement of tiles, where each tile has a (Tw × Th) rectangular arrangement of pixels. Here, Tw = Iw /Nw, and Th = Ih/Nh. For each (Tw × Th) tile, the IH is calculated independently.

After computation of all (Nw × Nh) tile-level IHs, the results can be processed to derive the image-level IH result.

For GPU-based implementation of IH computation, we design three types of two-dimensional signal processing actors. These actors are parameterized so that they can be statically or dynamically configured for the desired type of IH computation. Each of the three actors employed in our IH case study has a single input port and a single output port. These actors are described as follows.

First, the Bin-Check actor determines bin membership for pixels. The actor executes pixel checks of an image column for all bins with CONS = (1, Ih) and PROD = (1, Ih × Nb). Here, and in the remainder of this section, we denote the two-dimensional (MDSDF) production and consumption rates of a given actor port as PROD and CONS, respectively.

Second, the Intra-Tile-IH actor computes the IH, where the size of the input tile is specified by the actor parameters Tw (width) and Th (height), and the scan order is specified by the scan order parameter of the actor. The supported settings for the scan order parameter are as follows.

- *CWS*: Compute the IH using a cross-weave scan with tiling. The actor ports satisfy $CONS = PROD = (T_w, T_h)$.
- *WFS*: Compute the IH using a wavefront scan with tiling. The ports again satisfy $CONS = PROD = (T_w, T_h)$
- *NT*: Compute the IH using a cross-weave scan without tiling — that is, calculate the IH for the input image directly with $CONS = PROD = (T_w, T_h)$.

The Inter-Tile-IH actor performs accumulation among tiles with a parameter, called the accumulation order parameter, to support different scan orders for performing the accumulation. In particular, horizontal, vertical, and wavefront scans are used for accumulation order settings that are denoted HS, VS, and WFS, respectively. The actor ports of this actor (regardless of the accumulation order setting) satisfy CONS = PROD = (Iw, Ih). In addition, the accumulation order parameter can be set to the value IDLE to bypass any accumulation. While in the IDLE configuration, the actor performs no computation, and simply passes its input to its output (through a simple pointer transfer to avoid memory transfer overhead).

Given the actors developed, one can implement the IH application with the MDSDF graph shown in Figure 3. The desired scan orders and tiling settings can be achieved by setting the actor parameter values appropriately. In the experiments, we show performance comparisons among three specific application modes, which are defined by the groups of parameter settings shown in Table 3. Here, SOP stands for "scan order parameter."

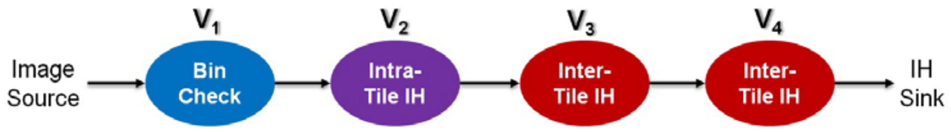

Figure 3. Multidimensional dataflow graph for image histogram computation.

We customize the implementations for the different application modes by examining their MDSDF application graph representations separately, and deriving separate DHs to guide the application mapping processing.

In our experiments, an NVIDIA GTX260 GPU and an Intel Xeon 3 GHz CPU are used. We compare the three different application modes in Table 3. Table 4 depicts the grid and block sizes for GPU kernels. Performance is compared for four image sizes ($I_w \times I_h$): 32x32, 64x64, 256x256, and 512x512. Based on the number of GPU threads employed for each kernel, we choose a tile size of (32×16) in the APP-CWS mode for all image sizes. For the APP-WFS mode, tile sizes of (4×4), (8×8), (16×8), and (32×16) are chosen for successively larger image sizes. We evaluate the frame processing time, including the time required for memory transfer from the host to the device (GPU) and the processing time on the device. We do not include the time for memory transfer from the device back to the host because many applications that employ IH can be implemented on the GPU efficiently without need for data transfer back to the CPU.

Table 3. Application modes.

App mode	Method	V2 SOP	V3 SOP	V4 SOP
APP-CWS	cross-weave TIH	CWS	HS	VS
APP-WFS	wavefront TIH	WFS	WFS	IDLE
APP-NT	no tiling	NT	IDLE	IDLE

Table 4. Grid sizes (upper) and block sizes (lower) derived from DH in our experiments.

mode	V2 kernel	V3 kernel	V4 kernel
APP-CWS	$(N_w, N_h N_b)$ $(T_w, 1)$	$(1, N_b)$ (T_w, T_h)	$(1, N_b)$ (T_w, T_h)
APP-WFS	$(1, N_b)$ (N_w, N_h)	$(1, N_b)$ (T_w, T_h)	N/A
APP-NT	$(1, N_b)$ $(I_w, 1)$	N/A	N/A

Figure 4 shows the frame rates (i.e., $1/\tau$, where τ represents the average time in seconds required to process a single frame) for various bin sizes ranging from 16 to 1024. From the experimental results, we see that the GPU implementation of the IH consistently outperforms the CPU implementation, and that the speedup gains are approximately 35X for image sizes 32x32 and 64x64; 67X for image size 256x256; and 75X for image size 512x512.

From the results, we observe that the best application mode for IH calculation depends on the image size, and thus parameterized MDSDF application modeling in conjunction with our multidimensional DH mapping approach is useful design methods to map IH computations systematically onto the targeted GPU platform. Such a systematic mapping approach leads to designs that can be mapped and adapted more efficiently, and that are more portable, and easier to maintain and extend.

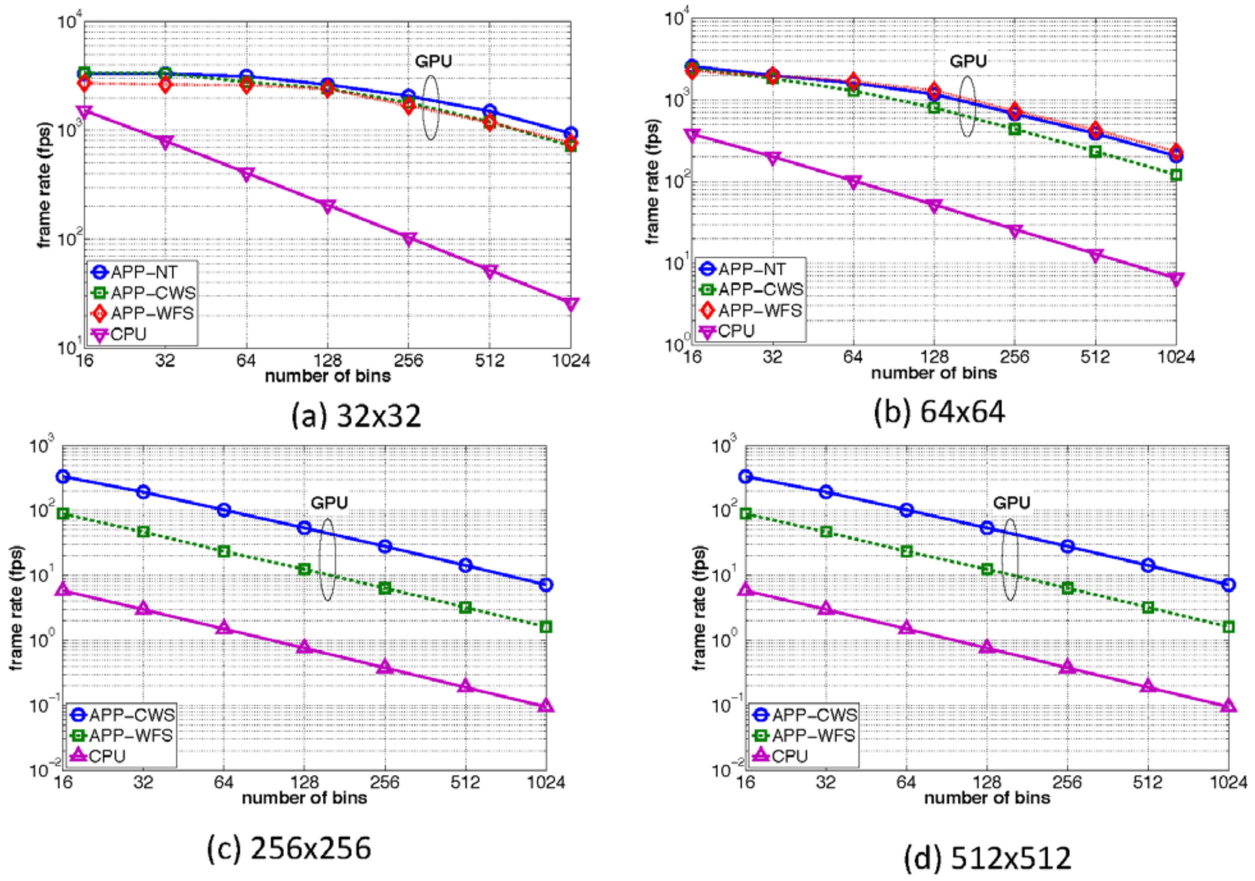

Figure 4. Performance comparisons for different image sizes.

5. CONCLUSIONS

In this project, we have developed new models and methods, rooted in high-level dataflow-based models of computation, for optimized design and implementation of multicore signal processing systems. We have developed new methods for hardware description of multicore systems, including the *CFDF-LWDF-TDIF* (*CLT*) *design flow*, and *the DSG-based design and implementation of embedded image processing systems* (*DEIPS*) *methodology*. These approaches emphasize, respectively, (1) integrated dataflow-based modeling, simulation and platform-specific optimization processes, and (2) systematic experimentation with and optimization of high level, dataflow graph schedules on multicore systems. Building on these methods, as well as on modeling techniques for multidimensional dataflow graphs, we have developed techniques for efficiently mapping code blocks for signal, image and video processing applications into optimized parallel implementations. We have demonstrated these new methods through extensive experimentation with key algorithms and application subsystems in the domains of video surveillance and pattern recognition. Useful directions for further investigation that have been motivated by this research include development of dynamic strategies for applying the models and methods underlying the CLT design flow and DEIPS methodology to adapt schedules and implementation configurations at run-time based on changes in data characteristics and application requirements.

6. REFERENCES

[Bellens 2011] P. Bellens, K. Palaniappan, R. M. Badia, G. Seetharaman, and J. Labarta. Parallel implementation of the integral histogram. In *Proceedings of the International Conference on Advanced Concepts for Intelligent Vision Systems*, 2011.

[Bhattacharyya 2013] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, editors. *Handbook of Signal Processing Systems*. Springer, second edition, 2013. ISBN: 978-1-4614-6858-5 (Print); 978-1-4614-6859-2 (Online).

[Bhattacharyya 1997] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. *Journal of Design Automation for Embedded Systems*, 2(1):33–60, January 1997.

[Bhattacharyya 1996] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Software Synthesis from Dataflow Graphs. Kluwer Academic Publishers, 1996.

[Dennis 1975] J. B. Dennis. First version of a data flow procedure language. Technical report, Laboratory for Computer Science, Massachusetts Institute of Technology, May 1975.

[DSPCAD 2013] Maryland DSPCAD Research Group website.
http://www.ece.umd.edu/DSPCAD/home/dspcad.htm. 2013.

[Haubelt 2007] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubühr, A. Deyhle, A. Hadert, and J. Teich. A SystemC-based design methodology for digital signal processing systems. *EURASIP Journal on Embedded Systems*, 2007:Article ID 47580, 22 pages, 2007.

[Hsu 2005] 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*, pages 37-49, Dallas, Texas, September 2005.

[Kahn 1974] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress*, 1974.

[Karp 1966] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queuing. *SIAM Journal of Applied Math*, 14(6), November 1966.

[Kienhuis 2001] B. Kienhuis and E. F. Deprettere. Modeling stream-based applications using the SBF model of computation. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 385-394, September 2001.

[Ko 2008] M. Ko, C. Shen, and S. S. Bhattacharyya. Memory-constrained block processing for DSP software optimization. *Journal of Signal Processing Systems*, 50(2):163–177, February 2008.

[Ko 2007] M. Ko, C. Zissulescu, S. Puthenpurayil, S. S. Bhattacharyya, B. Kienhuis, and E. Deprettere. Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software implementation. *IEEE Transactions on Signal Processing*, 55(6):3126–3138, June 2007.

[Kwon 2004] S. Kwon, H. Jung, and S. Ha. H.264 decoder algorithm specification and simulation in simulink and PeaCE. In *Proceedings of the International SoC Design Conference*, pages 9-12, October 2004.

[Lalgudi 2000] K. N. Lalgudi, M. C. Papaefthymiou, and M. Potkonjak. Optimizing computations for effective block-processing. *ACM Transactions on Design Automation of Electronic Systems*, 5(3):604–630, July 2000.

[Lee 1995] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, pages 773-799, May 1995.

[Lee 1989] E. A. Lee and S. Ha. Scheduling strategies for multiprocessor real time DSP. In *Proceedings of the Global Telecommunications Conference*, volume 2, pages 1279-1283, 1989.

[Lee 1987] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235-1245, September 1987.

[Murthy 2002] P. K. Murthy and E. A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50(8):2064–2079, August 2002.

[NVIDIA 2012] *NVIDIA CUDA C Programming Guide*, April 2012. Version 4.2.

[NVIDIA 2007] NVIDIA CUDA Compute Unified Device Architecture: Programming Guide, Version 1.0, June 2007.

[Owens 2008] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[Plishker 2008a] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya. Functional DIF for rapid prototyping. In *Proceedings of the International Symposium on Rapid System Prototyping*, pages 17–23, Monterey, California, June 2008.

[Plishker 2008b] W. Plishker, N. Sane, M. Kiemb, and S. S. Bhattacharyya. Heterogeneous design in functional DIF. In *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 157–166, Samos, Greece, July 2008.

[Plishker 2009] W. Plishker, N. Sane, and S. S. Bhattacharyya. A generalized scheduling approach for dynamic dataflow applications. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 111–116, Nice, France, April 2009.

[Ritz 1993] S. Ritz, M. Pankert, and H. Meyr. Optimum vectorization of scalable synchronous dataflow graphs. In *Proceedings of the International Conference on Application Specific Array Processors*, October 1993.

[Robbins 2002] C. B. Robbins. Autocoding Toolset software tools for automatic generation of parallel application software. Technical report, Management, Communications & Control, Inc., 2002.

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

[Shen 2012] C. Shen, S. Wu, N. Sane, H. Wu, W. Plishker, and S. S. Bhattacharyya. Design and synthesis for multimedia systems using the targeted dataflow interchange format. *IEEE Transactions on Multimedia*, 14(3):630–640, June 2012.

[Shen 2010] C. Shen, W. Plishker, H. Wu, and S. S. Bhattacharyya. A lightweight dataflow approach for design and implementation of SDR systems. In *Proceedings of the Wireless*

Innovation Conference and Product Exposition, pages 640–645, Washington DC, USA, November 2010.

[Stevens 1997] R. S. Stevens. The processing graph method tool (PGMT). In *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors*, pages 263-271, July 1997.

[TI 2012] Texas Instruments, Inc. TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor Data Manual, February 2012.

[Wu 2013] H.-H. Wu. *Modeling and Mapping of Optimized Schedules for Embedded Signal Processing Systems*. PhD thesis, Department of Electrical and Computer Engineering, University of Maryland, College Park, 2013.

[Wang 2012] L. Wang, C. Shen, G. Seetharaman, K. Palaniappan, and S. S. Bhattacharyya. Multidimensional dataflow graph modeling and mapping for efficient GPU implementation. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 300–305, Québec City, Canada, October 2012.

[Wu 2011] H. Wu, C. Shen, N. Sane, W. Plishker, and S. S. Bhattacharyya. A model-based schedule representation for heterogeneous mapping of dataflow graphs. In *Proceedings of the International Heterogeneity in Computing Workshop*, pages 66-77, Anchorage, Alaska, May 2011.

[Wu 2010] H. Wu, H. Kee, N. Sane, W. Plishker, and S. S. Bhattacharyya. Rapid prototyping for digital signal processing systems using parameterized synchronous dataflow graphs. In *Proceedings of the International Symposium on Rapid System Prototyping*, pages 1-7, Fairfax, Virginia, June 2010. DOI:10.1109/RSP_2010.10.

[Zhou 2013] Z. Zhou, C. Shen, W. Plishker, and S. S. Bhattacharyya. Dataflow-based, cross-platform design flow for DSP applications. In A. Sangiovanni-Vincentelli, H. Zeng, M. Di Natale, and P. Marwedel, editors, *Embedded Systems Development: From Functional Models to Implementations*, pages 41–65. Springer, 2013.

[Zivojnovic 1994] V. Zivojnovic, S. Ritz, and H. Meyr. Retiming of DSP programs for optimum vectorization. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 492–496, April 1994.

A. APPENDIX A — Publications and Presentations

The following is a list of **publications** that were produced as outcomes of this project.

[2014-1] H.-H. Wu, C.-C. Shen, H. Kee, N. Sane, W. Plishker, and S. S. Bhattacharyya. Mapping parameterized dataflow graphs onto FPGA platforms. In R. Chellappa and S. Theodoridis, editors, *Academic Press Library in Signal Processing*, volume 4, pages 643-673. Academic press, Elsevier Ltd., 2014.

[2013-1] L. Wang, C.-C. Shen, S. Wu, and S. S. Bhattacharyya. Parameterized scheduling of topological patterns in signal processing dataflow graphs. *Journal of Signal Processing Systems*, 71(3):275-286, June 2013. DOI:10.1007/s11265-012-0719-x.

[2013-2] H.-H. Wu. *Modeling and Mapping of Optimized Schedules for Embedded Signal Processing Systems*. PhD thesis, Department of Electrical and Computer Engineering, University of Maryland, College Park, 2013.

[2013-3] Z. Zhou. *Multi-Scale Scheduling Techniques for Signal Processing Systems*. PhD thesis, Department of Electrical and Computer Engineering, University of Maryland, College Park, 2013.

[2013-4] Z. Zhou, C. Shen, W. Plishker, and S. S. Bhattacharyya. Dataflow-based, cross-platform design flow for DSP applications. In A. Sangiovanni-Vincentelli, H. Zeng, M. Di Natale, and P. Marwedel, editors, *Embedded Systems Development: From Functional Models to Implementations*, pages 41-65. Springer, 2013.

[2012-1] L. Wang, C. Shen, G. Seetharaman, K. Palaniappan, and S. S. Bhattacharyya. Multidimensional dataflow graph modeling and mapping for efficient GPU implementation. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 300-305, Québec City, Canada, October 2012.

[2011-1] S. S. Bhattacharyya, W. Plishker, N. Sane, C. Shen, and H. Wu. Modeling and optimization of dynamic signal processing in resource-aware sensor networks. In *Proceedings of the Workshop on Resources Aware Sensor and Surveillance Networks in conjunction with IEEE International Conference on Advanced Video and Signal-Based Surveillance*, pages 449-454, Klagenfurt, Austria, August 2011.

[2011-2] H. Wu, C. Shen, N. Sane, W. Plishker, and S. S. Bhattacharyya. A model-based schedule representation for heterogeneous mapping of dataflow graphs. In *Proceedings of the International Heterogeneity in Computing Workshop*, pages 66-77, Anchorage, Alaska, May 2011.

The following is a list of **presentations** that were delivered in connection with this project.

1. L. Wang, Multidimensional dataflow graph modeling and mapping for efficient GPU implementation. *IEEE Workshop on Signal Processing Systems*, Québec City, Canada, October 2012.

2. S. S. Bhattacharyya, Modeling and optimization of dynamic signal processing in resource-aware sensor networks. *Workshop on Resources Aware Sensor and Surveillance Networks in conjunction with IEEE International Conference on Advanced Video and Signal-Based Surveillance*, Klagenfurt, Austria, August 2011.
3. H. Wu, A model-based schedule representation for heterogeneous mapping of dataflow graphs. *International Heterogeneity in Computing Workshop*, Anchorage, Alaska, May 2011.

B. APPENDIX B — Abstracts

This section provides abstracts of conference, journal, and book chapter publications that were produced as outcomes of this project.

[2014-1] H.-H. Wu, C.-C. Shen, H. Kee, N. Sane, W. Plishker, and S. S. Bhattacharyya. Mapping parameterized dataflow graphs onto FPGA platforms. In R. Chellappa and S. Theodoridis, editors, *Academic Press Library in Signal Processing*, volume 4, pages 643-673. Academic press, Elsevier Ltd., 2014

As the speed and logic capacity of field programmable gate arrays (FPGAs) have been improving steadily, FPGAs have become increasingly attractive for a wide variety of signal processing systems. FPGAs are increasingly employed in the form of platform FPGAs, which are integrated circuits that combine significant amounts of configurable logic fabric along with additional subsystems, such as application-specific accelerators, processor cores, memory blocks, and input/output interfaces, to facilitate FPGA-based, system-on-chip design. FPGA fabric is also integrated into application specific integrated circuits (ASICs) to allow implementations that provide a mix of programmable and custom hardware.

Through support for dynamic reconfiguration, modern FPGAs allow customization of hardware structures both statically and at run-time, thus allowing streamlining of processing configurations in response to application requirements or data characteristics that are not known at design time. In addition to allowing for dynamic changes in system functionality, dynamic reconfiguration, when carried out effectively, can enhance performance, resource utilization, and energy efficiency.

However, in addition to such potential for improved operation, incorporating dynamic reconfiguration into the digital system design space also brings increased design complexity. Model-based design methodologies have been evolving steadily over the years to help address issues of design complexity in embedded systems. In model-based design, applications are represented and analyzed in terms of formal models of computation, which promote analysis of functionality as well as hardware and software structure at a high level of abstraction. In the domain of signal processing, model-based techniques based on dataflow models of computation are particularly popular, and are employed in a growing variety of design tools.

While dataflow techniques allow for high level reasoning about and manipulation of application dynamics, there are important challenges in mapping dataflow models into FPGA platforms in ways that systematically and effectively exploit the dynamic reconfiguration capabilities of the platforms. This paper provides a review of state-of-the-art model-based design techniques and FPGA implementation techniques for signal processing systems, and explores the challenges involved in effectively mapping high level application models into efficient implementations on dynamically reconfigurable FPGA platforms.

The exploration presented in this paper on mapping models into implementations builds on our earlier work in this area, which was presented in preliminary form in [Wu 2010]. The reconfiguration-aware mapping techniques presented in this paper go beyond the developments of [Wu 2010] in a number of ways. Specifically, this extended paper enhances the hardware architecture mapping methodology of [Wu 2010] and provides two alternative perspectives on scheduling. These two perspectives affect important trade-offs between performance and modularity. An important new aspect integrated into one of these scheduling perspectives involves integration of the recently-developed dataflow schedule graph model into processes for FPGA mapping of dynamically reconfigurable signal processing systems.

> **[2013-1]** L. Wang, C.-C. Shen, S. Wu, and S. S. Bhattacharyya. Parameterized scheduling of topological patterns in signal processing dataflow graphs. *Journal of Signal Processing Systems*, 71(3):275-286, June 2013. DOI:10.1007/s11265-012-0719-x.

In recent work, a graphical modeling construct called "topological patterns" has been shown to enable concise representation and direct analysis of repetitive dataflow graph sub-structures in the context of design methods and tools for digital signal processing systems. In this paper, we present a formal design method for specifying topological patterns and deriving parameterized schedules from such patterns based on a novel schedule model called the scalable schedule tree. The approach represents an important class of parameterized schedule structures in a form that is intuitive for representation and efficient for code generation. Through application case studies involving image processing and wireless communications, we demonstrate our methods for topological pattern representation, scalable schedule tree derivation, and associated dataflow graph code generation.

> **[2013-4]** Z. Zhou, C. Shen, W. Plishker, and S. S. Bhattacharyya. Dataflow-based, cross-platform design flow for DSP applications. In A. Sangiovanni-Vincentelli, H. Zeng, M. Di Natale, and P. Marwedel, editors, *Embedded Systems Development: From Functional Models to Implementations*, pages 41-65. Springer, 2013.

Dataflow methods have been widely explored over the years in the digital signal processing (DSP) domain to model, design, analyze, implement, and optimize DSP applications, such as applications in the areas of audio and video data stream processing, digital communications, and image processing. DSP-oriented dataflow methods provide formal techniques that facilitate software design, simulation, analysis, verification, instrumentation and optimization for exploring effective implementations on diverse target platforms. As the landscape of embedded platforms becomes increasingly diverse, a wide variety of different kinds of devices, including graphics processing units (GPUs), multicore programmable digital signal processors (PDSPs), and field programmable gate arrays (FPGAs), must be considered to thoroughly address the design space for a given application. In this chapter, we discuss design methodologies, based on the core functional dataflow (CFDF) model of computation, that help engineers to efficiently explore such diverse design spaces. In particular, we discuss a CFDF-based design flow and associated design methodology for efficient simulation and implementation of DSP applications. The design flow supports system formulation, simulation, validation, cross-platform software implementation, instrumentation, and system integration capabilities to derive optimized signal processing implementations on a variety of platforms. We provide a comprehensive specification of the design flow using the lightweight dataflow (LWDF) and targeted dataflow interchange format (TDIF) tools, and demonstrate it with case studies on CPU/GPU and multicore PDSP designs that are geared towards fast simulation, quick transition from simulation to the implementation, high performance implementation, and power-efficient acceleration, respectively.

[2012-1] L. Wang, C. Shen, G. Seetharaman, K. Palaniappan, and S. S. Bhattacharyya. Multidimensional dataflow graph modeling and mapping for efficient GPU implementation. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 300-305, Québec City, Canada, October 2012.

Multidimensional synchronous dataflow (MDSDF) provides an effective model of computation for a variety of multidimensional DSP systems that have static dataflow structures. In this paper, we develop new methods for optimized implementation of MDSDF graphs on embedded platforms that employ multiple levels of parallelism to enhance performance at different levels of granularity. Our approach allows designers to systematically represent and transform multi-level parallelism specifications from a common, MDSDF-based application level model. We demonstrate our methods with a case study of image histogram implementation on a graphics processing unit (GPU). Experimental results from this study show that our approach can be used to derive fast GPU implementations, and enhance trade-off analysis during design space exploration.

[2011-1] S. S. Bhattacharyya, W. Plishker, N. Sane, C. Shen, and H. Wu. Modeling and optimization of dynamic signal processing in resource-aware sensor networks. In *Proceedings of the Workshop on Resources Aware Sensor and Surveillance Networks in conjunction with IEEE International Conference on Advanced Video and Signal-Based Surveillance*, pages 449-454, Klagenfurt, Austria, August 2011.

Sensor node processing in resource-aware sensor networks is often critically dependent on dynamic signal processing functionality — i.e., signal processing functionality in which computational structure must be dynamically assessed and adapted based on time-varying environmental conditions, operating constraints or application requirements. In dynamic signal processing systems, it is important to provide flexibility for run-time adaptation of application behavior and execution characteristics, but in the domain of resource-aware sensor networks, such flexibility cannot come with significant costs in terms of power consumption overhead or reduced predictability. In this paper, we review a variety of complementary models of computation that are being developed as part of the dataflow interchange format (DIF) project to facilitate efficient and reliable implementation of dynamic signal processing systems. We demonstrate these methods in the context of resource-aware sensor networks.

[2011-2] H. Wu, C. Shen, N. Sane, W. Plishker, and S. S. Bhattacharyya. A model-based schedule representation for heterogeneous mapping of dataflow graphs. In *Proceedings of the International Heterogeneity in Computing Workshop*, pages 66-77, Anchorage, Alaska, May 2011.

Dataflow-based application specifications are widely used in model-based design methodologies for signal processing systems. In this paper, we develop a new model called the dataflow schedule graph (DSG) for representing a broad class of dataflow graph schedules. The DSG provides a graphical representation of schedules based on dataflow semantics. In conventional approaches, applications are represented using dataflow graphs, whereas schedules for the graphs are represented using specialized notations, such as various kinds of sequences or looping constructs. In contrast, the DSG approach employs dataflow graphs for representing both application models and schedules that are derived from them.

Our DSG approach provides a precise, formal framework for unambiguously representing, analyzing, manipulating, and interchanging schedules. We develop detailed formulations of the DSG representation, and present examples and experimental results that demonstrate the utility of DSGs in the context of heterogeneous signal processing system design.

7. LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

API	Application Programming Interface
APGAN	Acyclic pairwise grouping of adjacent nodes
BMP	Bitmap Image File
CDSG	Concurrent DSG
CFDF	Core Functional Dataflow
CLT	CFDF-LWDF-TDIF
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DDR	Double Data Rate
DEIPS	DSG-based design and implementation of embedded image processing systems
DH	Design Hierarchy
DIF	Dataflow Interchange Format
DSG	Dataflow Schedule Graph
DSP	Digital Signal Processing
FCV	Filter Coefficient Vector
FIFO	First-In-First-Out
FPGA	Field-Programmable Gate Array
GF	Gaussian Filter
GHz	Gigahertz
GPU	Graphics Processing Unit
GST	Generalized Schedule Tree
HSDF	Homogeneous Synchronous Dataflow

HS	Horizontal Scan
IH	Integral Histogram
LWDF	Lightweight Dataflow
MDSDF	Multidimensional Synchronous Dataflow
MPSoC	Multiprocessor System-on-Chip
PDSP	Programmable Digital Signal Processor
PGM	Processing Graph Method
SDF	Synchronous Dataflow
SDM	Signal-processing-oriented Dataflow Model (of computation)
SDSG	Sequential Dataflow Schedule Graph
SRAM	Static Random Access Memory
STC	Store Coefficients
TDIF	Targeted DIF
TDL	The DIF Language
TDP	The DIF Package
TI	Texas Instruments
TIH	Tiled Integral Histogram
VS	Vertical Scan
WFS	Wavefont Scan