BLOCK PARALLEL PROGRAMMING FOR REAL-TIME APPLICATIONS ON MULTI-CORE PROCESSORS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

David Black-Schaffer
April 2008

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(William J. Dally) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Mark Horowitz)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Anwar Ghuloum
(Intel Inc.))

Approved for the University Committee on Graduate Studies.

Abstract

This thesis presents a streaming block-parallel programming language for describing applications with hard real-time constraints and several transformations for parallelizing and mapping such applications to many-core architectures. The language parameterizes the data movement within the application in such a manner as to enable simple application analysis and rapid software development. Key enhancements for both programmer productivity and application analyzability include the use of native two-dimensional data streams to simplify image processing algorithms, flexible control distribution for intuitive and analyzable synchronous and asynchronous control, explicit data dependency edges for specifying limited parallelism, and multiple computation methods per kernel. The benefits of this application description are shown through the intuitive analyses and manipulations required to parallelize, buffer, and map such applications to a many-core architecture while insuring that they meet their real-time computation requirements.

To evaluate this programming approach, a program analysis and manipulation framework and a cycle-accurate simulation environment were constructed. These were used to automatically analyze, parallelize, buffer, and execute a variety of test programs to ensure that they met their specified real-time constraints. The results suggest that this block-parallel programming approach is both easier to use and analyze than previous streaming programming models.

Acknowledgements

First and foremost I must thank my wife Annica. Without her support, encouragement, and belief in my ability to complete this work, it would never have happened.

Secondly, I am grateful to my advisor Bill. Not only has he supported and guided me through this processes, but he is one of those rare people with enough experience and knowledge to almost always be right, but enough self-confidence and grace to admit it when he is not. Interacting with him has always been academically challenging, but his push to keep the big picture in mind and hone in on the essence of the problem has been invaluable.

Thirdly, my committee members Mark and Anwar have provided me with different points of view which have helped me to focus on the value of my work and its presentation.

I must also thank my family, in particular my parents, for their unending support and encouragement during my long tenure at Stanford. Even their gentle insinuations that graduating would be appreciated have been helpful.

And of course I thank my friends and colleagues here at Stanford. In the CVA group I am grateful to James Balfour, Mattan Erez, Paul Hartke, Abhishek Das, JungHo Ahn, Tim Knight, Vishal Parikh, JongSoo Park, James Chen, and Curt Harting, and in the SOE Charlie Orgish and Keith Gaul. My special thanks go to my friends Kristof Richmond, Greg and Cami Kuhnen, Oren and Dorit Kerem, Tibor Fabian and Rika Yonemura, and Sarah Seestone and Darsi Rueda. And, of course, thanks to the members of the *Seleccion Natural* soccer team for all of the fun (and injuries) I've had over the past several years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The goal of block-parallel programming is to provide a flexible structure for writing readily analyzable data-parallel applications that can be mapped to future processors with hundreds or thousands of computation cores. In addition to the traditional problem of identifying parallel computation within an application, to fully utilize these architectures the harder problem of mapping and scheduling data movement across a distributed non-coherent memories needs to be addressed. The block-parallel programming approach presented here strives to provide a programming approach that exposes data movement and parallelism such that the compiler tool chain can easily manipulate the application and intelligently map it to the hardware.

Traditional imperative programming languages have proven very hard to analyze for automatic parallelization and data movement. The most successful automatically parallelizing compilers are only able to readily handle affine inner-loops with minimal control flow dependencies in (reasonably) well-typed languages. Attempts to analyze typical code in languages that allow arbitrary pointer access constructs, such as C, have proven enormously less successful.

However, despite many decades of research, even the most capable program analysis and manipulation frameworks run into the fundamental issue that even when they can analyze a program, it is not at all clear what manipulations should be applied for optimal performance. The complexity of the code, as evidenced by the number of possible non-commuting loop transformations, leads to an extremely large search space.

This search is made even more difficult by the discrete, and often non-monotonic, performance results from varying the transformations, their parameters, and their orderings. The result is that most optimizing compilers are extremely conservative, optimizing only the innermost loop or two of purely affine loop nests or applying crude blocking to the outermost loops. For architectures with sufficiently low memory latencies or applications with sufficiently high arithmetic intensity this simplistic approach is acceptable. Unfortunately this approach has not been motivated by either of those artifacts per se, but rather by the difficulty of full-program analysis for complex architectures and applications. As architectures continue to become more distributed and less coherent due to scaling difficulties (e.g., they look less and less like a traditional von Neumann architectures), successful parallelization approaches will need to be based on more structured inputs to the compiler.

This difficulty of analyzing traditional imperative programming languages motivates the need to provide programming approaches that are easier to analyze. Stream-based programming in general, and the block-parallel programming presented here, in particular, attempt to address this issue. These approaches take advantage of the way programmers tend to naturally think about decomposing algorithms, by structuring applications as graphs of computation kernels and data streams. This provides the compiler with a high-level view of the data movement and parallelism present in the application without the complex analysis required for imperative languages. However, for this approach to be successful, it must present a simple enough interface to the programmer to make it productive. Previous stream-based programming systems have presented programmers with very awkward methods for dealing with control between kernels and two-dimensional data streams.

The block-parallel application model presented here strives to allow programmers to intuitively specify applications as a graph of parameterized kernels connected by two-dimensional data and control streams. In this approach, the kernels and streams are parameterized to enable analysis by the compilation system for automatic parallelization, data movement, and placement on an array of processors.

1.0.1 The 2D Streaming Model

Stream programming systems, such as Brook [6], StreamIt [43], StreamC/KernelC [35], and a variety of Synchronous Data Flow (SDF) [32] approaches, treat their inputs as infinite or quasi-inifinite streams of data. The computation kernels that act on these streams specify the number of stream elements they need to execute, and the number of iterations of execution is implicitly defined by the length of the stream divided by the number of unique elements required per execution. The computation kernels are connected together and to inputs and outputs by channels which act as FIFOs for the stream data. This programming model maps well to application domains where the input is generated and processed continuously, and where reuse of the data is largely local in both time and space. Applications such as media and signal processing are prime examples of this domain. Thies [43] describes such applications as having six salient characteristics: 1) large streams of data, 2) independent stream computation kernels, 3) a stable computation pattern, 4) only occasional modification of stream structure, 5) only occasional out-of-stream communication, and 6) high performance expectations.

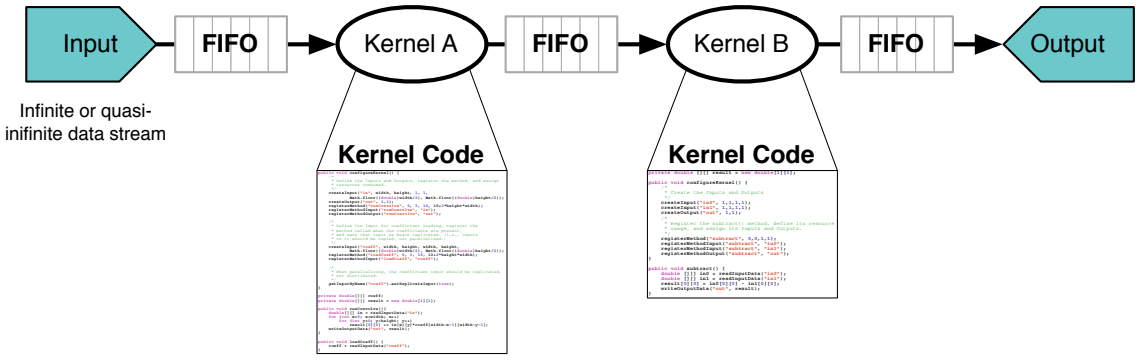


Figure 1.1: Basic stream program example
The kernel code for each kernel is logically executed on the incoming data whenever enough data arrives. The kernel then produces output data which is sent downstream via the FIFOs between kernels.

The execution flow of a stream program is defined by connecting multiple kernels together in a directed graph that represents the desired flow of data through the

computation kernels. To apply kernel "A" to an input stream, the stream would be connected to the input on kernel "A". To then apply kernel "B" to the result, the output of kernel "A" would then be connected to the input of kernel "B". Such a toy application is illustrated in Figure 1.1. This programming style closely matches signal and image processing problems where algorithms tend to be designed by applying reusable computation kernels to data streams.

Most streaming systems define their data streams as one-dimensional series of data, although there have been multiple proposals for multi-dimensional streaming [32, 26, 3] and blocking [14] languages. Despite this one-dimensional limitation, they can handle higher dimensional data by appropriately (and usually manually) indexing into the single-dimensional streams. This enables image and video processing within the same framework without explicit two-dimensional data support. However, this approach complicates the program analysis by hiding the underlying program structure, and requires that the programmer keep track of the particular data layout throughout the application. The approach taken in this work is to provide a native two-dimensional data stream to simplify writing and analyzing programs that manipulate two-dimensional data. (See Figure 1.2.) Like other streaming systems, the two-dimensional stream consists conceptually of an infinite number of elements, but unlike one-dimensional approaches, each element in the stream is a two-dimensional *data frame*. While the number of frames is assumed infinite (such as might be found in a video stream), each frame itself is of a fixed, and statically determined, size. The data within each input frame is further assumed to stream into the application, in a left-to-right, top-to-bottom manner. One-dimensional streaming applications can be readily written by simply defining the streams to be height one.

To take advantage of the two-dimensional input streams, the computation kernels that operate on them are also defined with two-dimensional inputs and outputs. This parameterization greatly simplifies the programming system's job of analyzing how the data is used and reused for applications that map nicely to one- or two-dimensional inputs. For one-dimensional streaming systems, this two-dimensional information must be inferred from the use of the streams within the kernels, and can only be utilized if the inferences can be proven to hold throughout the application. The

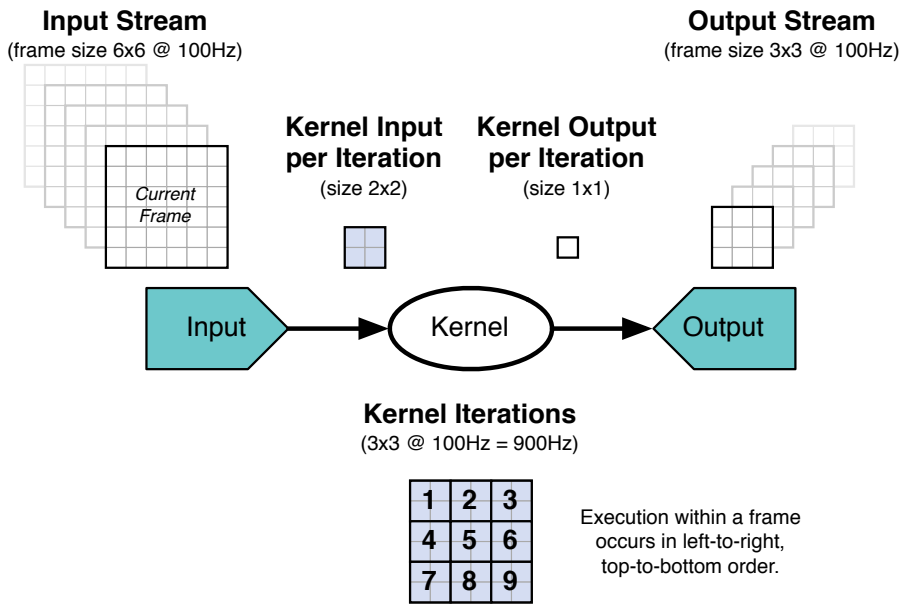


Figure 1.2: 2D streaming application model example
The input stream is depicted on the left. The stream consists of frames of size 6×6 which are processed through the kernel shown in the middle. The kernel's input is 2×2, which tiles the input frame size 3×3 times, producing an output of size 3×3, which defines the output stream.

manipulations enabled by explicit two-dimensional parameterization include reuse calculation, buffer sizing and partitioning, and application consistency analysis. With this enhanced knowledge of the data usage patterns of the application, it is readily possible to parallelize and map the application to an array of processors to meet a specified data rate without heroic compiler analysis.

1.0.2 Block Programming Example

To motivate this programming approach, consider applying a median filter (Figure 1.3) to an input image. The goal is to provide an automatic system for parallelizing this filter to meet the real-time constraint imposed by the input image's size and rate.

A median filter processes the input image in overlapping windows, sorting the pixel values in each window, and outputting the median value for the window at each pixel

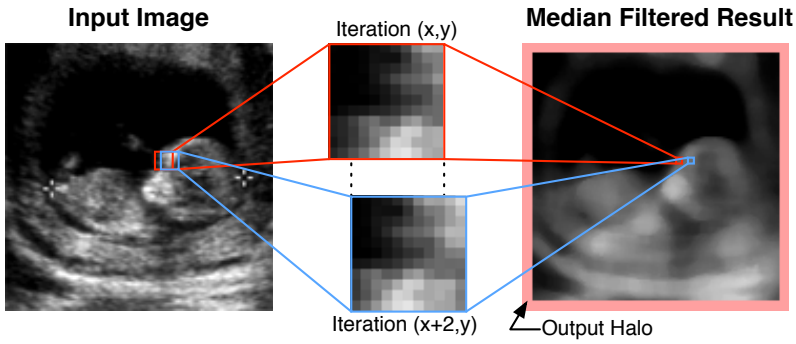


Figure 1.3: Example median filter

The application of a median filter to an input image is shown. The input windows and output pixels for two iterations of the filter are shown in red and blue. The input windows overlap as can be seen from the alignment of the two filter input windows shown in the middle. The output image is surrounded by a halo of invalid data that must be either removed or generated by zero-padding the original input image.

location. The straight-line imperative code for the median filter, as shown in Figure 1.4, is quite straightforward. It consists of four sets of nested loops: two outer loops for walking over the image to choose the window, and two inner loops for walking over the window to calculate the median value. For a compiler to automatically analyze and parallelize this code to meet a given data rate, it needs to be able to analyze this loop nest. However, the code within the inner two loops has data-dependent behavior. That is, the sorting of the values to calculate the median can not be statically predicted, and therefore can not be statically analyzed. This limits the compiler analysis to examining the outer two loops.

The two outer loops in this code are very straightforward, and can be readily understood and parallelized with a basic affine analysis[1]. Unfortunately, examining just the outer two loops is insufficient in this case. The median filter processes data in windows that overlap between iterations, with the size of these windows and the overlap being defined by the data access patterns in the inner loops. This means that any parallelization needs to calculate this overlap by examining the range of

[1]Note that while this analysis is "understood" in the academic sense, no common compiler is actually advanced enough to implement it.

```
image[width][height];
output[width-halo][height-halo];

for (y=0; y<height-halo; y++)                          Outer Loops
    for (x=0; x<width-halo; x++) {

        sortedList.clear();                            Inner Loops
        for (xx=0; xx<halo; xx++) {
            for (yy=0; yy<halo; yy++) {
                sortedList.insert(image[x+xx][y+yy]);
            }
        }
        output[x][y] = sortedList.getMedian();

    }
}
```

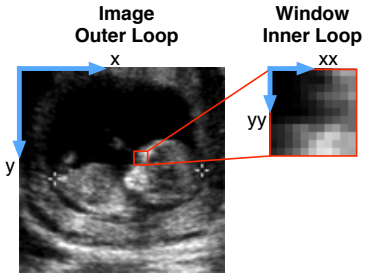


Figure 1.4: Imperative (standard) median filter implementation

The imperative code for a median filter consists of two *outer loops* for walking over the input image and two *inner loops* for walking over the window at each location. The pixels in the window to be processed are sorted and the result is the median value of that sorted list.

data accessed in the inner two loops for each execution of the outer two loops, and duplicate this data accordingly.

This analysis leaves the compiler with the choice of parallelizing along either of the two outer loops, with the inner loop code being executed for each iteration. The choice of which to use is driven by an analysis of the data reuse enabled by each approach when loading the input image from off-chip memory. (See Figure 1.5.) If the parallelization splits along the outermost loop, the result will be poor data locality as each processor will be accessing data for a separate portion of the input image simultaneously. Without appropriate banking of the memory system and careful data layout this will result in poor performance. If the parallelization splits along the innermost loop, each processor will be reloading the same data its neighbor requested recently, which will significantly increase the total traffic in the absence of a higher-level cache. These problems are a result of the interplay of the parallelization and data access patterns of the underlying code, something which is poorly exposed through the code analysis, and which is even harder to reason about even if it is exposed. Unfortunately this is likely to be the performance dominating feature of the analysis, particularly for many-core architectures where on-chip data movement must be carefully coordinated due to a paucity of off-chip bandwidth.

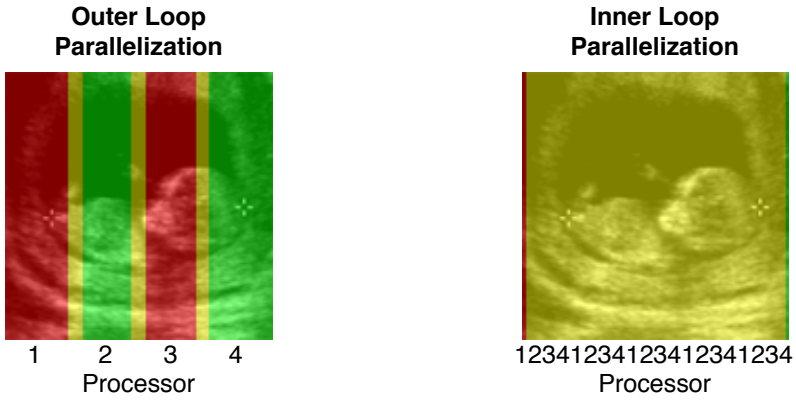


Figure 1.5: Data distribution for inner- and outer-loop parallelization

The portion of the image filtered by each processor is shown alternatingly in red and green, with shared data is shown in yellow. Outer-loop parallelization divides up the image into vertical slices which are accessed at the same time by each processor, causing a stream of non-sequential external memory requests. Inner-loop parallelization causes each processor to process sequential blocks, which results in each processor generating memory accesses for the same data that the previous processor requested on the previous iteration.

To determine the degree of parallelization required, the compiler needs to know two things: the size and rate of the input image, and the amount of time each iteration of the filter takes. The processing time per iteration depends on how the compiler chooses to parallelize the code, but, if we assume for simplicity that the compiler treats the inner two loops as the unit of computation, the processing time can be reasonably approximated by a static analysis or with some help from the programmer. The input rate is specified no where in the code, and must be provided externally or through a custom pragma to the compiler.

The block-parallel programming approach presented here greatly simplifies this analysis by parameterizing the application in two ways. The first is that the computation is expressed as a graph of computation kernels with parameterized inputs and outputs. The second is to define an order of iterating over input data. Taken together, these eliminate the inner loop analysis (via kernel parameterization) and outer loop analysis (via an explicit data iteration order) required to manipulate the

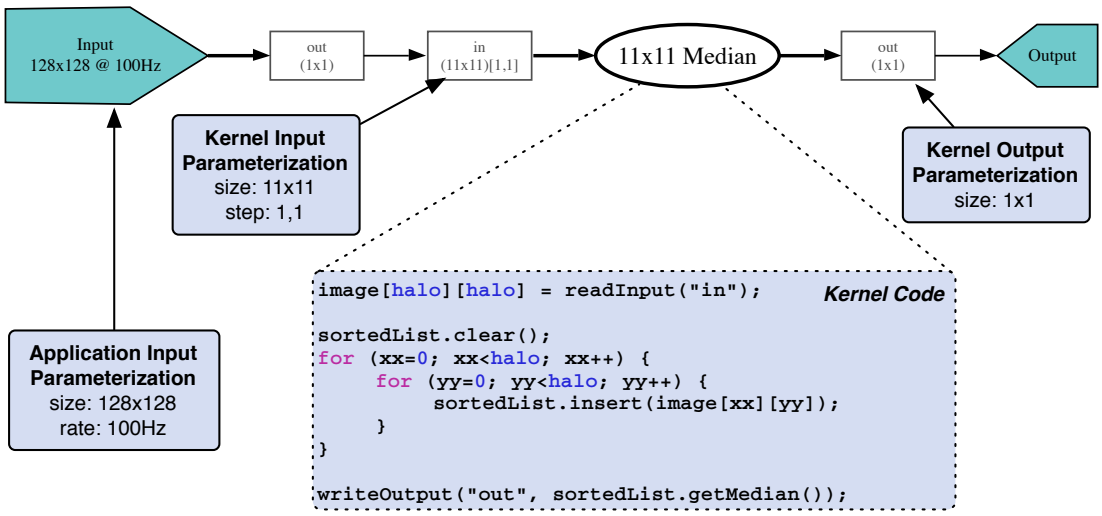


Figure 1.6: Block-parallel median filter implementation

The block-parallel median filter implementation defines a median filter kernel which contains the code to execute on each iteration. The kernel also defines the input window size it requires and the output data size. The program image input size and rate are specified by the input "Input" on the left.

median filter presented above. The tradeoff is that the programmer must now provide the data derived by the above analysis, but as the programmer has intimate knowledge of both the domain and the filter being implemented, this is a relatively low-cost requirement.

Indeed, the information the programmer must provide actually simplifies the application development and encourages code reuse. For the median filter, shown in Figure 1.6, the programmer needs to specify that the input window size is (width $\times$ height) and that this window steps through the input in (1,1) steps for each horizontal and vertical iteration, respectively. Taken together these define how much data the median filter needs for each iteration, and how much reuse and overlap is present in the computation. The output is similarly simply defined as being size ($1\times$ 1). The kernel must finally specify the amount of computation required per iteration, which is just as easily calculated for the kernel implementation as for the straight-code imperative implementation. When the full block-parallel application is written, the input

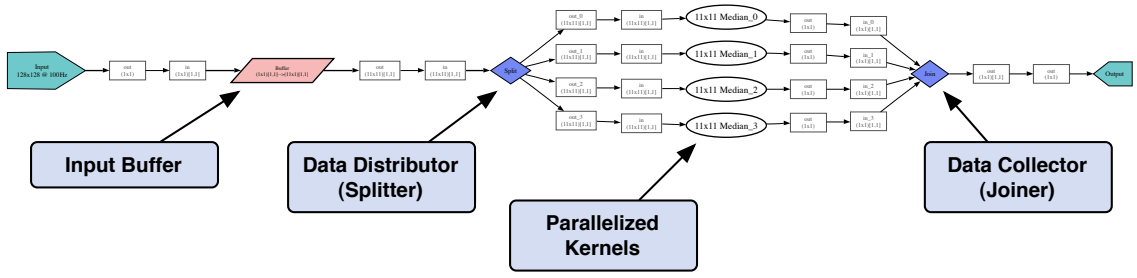


Figure 1.7: Automatically parallelized block-parallel median filter implementation

The block-parallel median filter implementation enables the automatic buffer insertion and parallelization shown here. The median kernel has been replicated four times and appropriate data distribution and collection (Split/Join) kernels and an input buffer have been inserted.

size and rate are explicitly defined by the parameterization of the input connected to the kernel, and therefore no special pragmas are required to pass on this information.

With this information, the compiler can avoid all of the analysis described above. The addition of a fixed input data ordering (here left-to-right, scanline order), simplifies the compiler's decision as to how to parallelize the processing, as well. (See Figure 1.7.) By specifying the input data ordering, the compiler can determine the parallelization so as to maximize reuse within the input data, thereby minimizing access to off-chip memory. The result is that the parallelization analysis need merely determine how many kernels are required ($\frac{inputRate \times inputSize}{timePerKernelIteration}$) and then send the data in a round-robin fashion to that many separate instances of the kernel on different processors. The compiler can insert a buffer to store the incoming data in the known input order to ensure that the processors obtain maximum reuse and bandwidth to the off-chip memory.

The downside of this approach is that the compiler is left with less flexibility in determining the parallelization, and the programmer is forced to make the programs conform to the kernel/stream programming model. However, with the language features discussed in this thesis the difficulty of writing traditionally non-data-parallel applications in a streaming language can be significantly reduced, and the losses from a more constrained parallelization program are readily compensated for by the ability

to robustly and efficiently parallelize a general, and useful, class of programs.

1.1 Contributions

This thesis describes a block-parallel streaming programming approach and the associated analyses and application transformations required to automatically parallelize such applications to meet hard real-time constraints. The specific contributions of this work are as follows:

- A practical two-dimensional "block-parallel" streaming language whose programs are described as an application graph with data streams connecting computation kernels. Each kernel supports multiple inputs and outputs and multiple computation methods.
- A compiler system that can automatically parallelize the block-parallel application to meet real-time constraints. The constraints are defined by the size and rate of the application's inputs, which are propagated through the application graph by a dataflow analysis. This analysis enables the compiler to determine the required computation rates, and hence degree of parallelization, at each point in the application. The regular streaming structure of the program then enables automatic parallelization to meet these constraints.
- A cycle-accurate functional simulator for evaluating the performance of the parallelized applications that uses the same program code as the compiler, but avoids the need to implement a hardware simulator.
- Analyses to determine the need to inset or zero-pad data to fix inconsistencies in the intuitive description of two-dimensional streaming applications, thereby improving the reusability of code.
- A method for describing the distribution of control tokens which is both analyzable and flexible. By including the control tokens in the application analysis, the overhead of non-trivial control handling code can be accurately incorporated in the performance estimation.

- The use of multiple methods per kernel which can be independently triggered by incoming data or control to simplify application structure and data sharing. Combined with the flexibility of the control tokens, this allows powerful multiplexing of control and data over a single logical input to a kernel.
- The use of data dependency edges in the application graph to intuitively and flexibly limit the degree of parallelism allowed for certain computations.
- Automatic sizing and insertion of two-dimensional circular buffers between kernels.

1.2 Thesis Overview

This thesis describes a block-parallel programing and compilation system and provides examples of the transformations and analyses needed to map applications to many-core architectures. The thesis begins with an overview of relevant background material in Chapter 2. This includes an introduction to Synchronous Data Flow (SDF) application descriptions and current streaming hardware architectures. SDF is important as it has formed much of the basis for stream-kernel programming, and is the only approach that has produced a decent mathematical formulation for analyzing the application structure.

Next an overview of related work is presented in Chapter 3, with a focus on a variety of relevant streaming languages, including StreamIt (Section 3.1), StreamC/ KernelC (Section 3.2), Brook (Section 3.3), Sequoia (Section 3.4), and multi-dimensional SDF (Section 3.5). Particular detail is paid to the StreamIt language as it is one of the most developed implementations.

With this background, the programming model for the block-parallel programming system is presented in Chapter 4. The *Application Graph* used to describe the overall application and data flow is introduced in Section 4.1. The details of the *Data Model*, including data streams, data transport, and control are described in Section 4.2. This is followed by a description of the *Computation Model* in Section 4.3, and several

examples in Section 4.4. The chapter concludes with a discussion comparing the model presented here with the other languages discussed in Chapter 3.

After having established the basic application description framework in Chapter 4, Chapter 5 describes the analysis required to determine the computation rates of the various kernels in the application. The salient values, *frame size*, *frame rate*, and *iteration size*, are introduced and defined in Section 5.1, and the data flow analysis required to calculate them is described in Section 5.2. The chapter concludes with a brief discussion of feedback (Section 5.2.1) and an example detailing the application of the described data flow analysis (Section 5.3). The example reveals two types of *application inconsistencies* in the sample application which are to be addressed in Chapter 6.

Chapter 6 starts where Chapter 5 ends by addressing the aforementioned application inconsistencies. Their causes are explained, and the need to insert buffers and insets to correct them is discussed. Section 6.1 describes the two-dimensional circular buffers required for buffering a two-dimensional streaming application, and derives the sizing needed for minimal and double-buffering, with particular care paid to buffering between input frames. The implementation of the buffers as *BufferKernels* is presented in Section 6.1.2. Insets, the extended data flow analysis required to calculate them, and their implementation as *InsetKernels* are presented in Section 6.2. Two detailed examples are also provided to demonstrate the analysis. An alternative to inset insertion, zero padding, is discussed in in Section 6.2.2. The chapter concludes with a description of the automated insertion of buffers and insets to create complete and consistent applications.

Chapter 7 takes the complete and consistent applications from the automatic analysis in Chatper 6 and adds automatic parallelization to meet the required data rates of the application's inputs. To implement the parallelization, Section 7.1 introduces *Split* and *Join* kernels to allow flexible data distribution to parallelized kernels. The remainder of the chapter discusses the issues involved with parallelizing purely data-parallel kernels (Section 7.2), kernels with limited parallelism (Section 7.3) and finally BufferKernels (Section 7.4), whose parallelism is defined by the order in which data

must be written into them. Examples of parallelized applications showing the automatic analysis and adjustment for both application data input rates and processor capabilities are shown in Section 7.5, and the chapter concludes with discussions of data reuse from parallelized buffer kernels (Section 7.6.1), the inefficiencies of Split/Join kernels (Section 7.6.2), and a comparison to related work.

The automatic parallelization of Chapter 7 implies a naïve 1:1 mapping of kernels to processors, which results in a low overall utilization. Chapter 8 addresses this by first analyzing the utilization of the 1:1 mapping (Section 8.1) and then proposing a simple greedy merging algorithm for time-multiplexing the kernels to achieve better utilization (Section 8.2). The greedy merge algorithm is implemented and the final full analysis, buffering, parallelization, and time-multiplexing is evaluated across a range of programs (Section 8.3).

The thesis concludes in Chapter 9 by discussing the overall merits of the proposed block-parallel programming system and the analyses discussed herein. Three appendices briefly touch upon using simulated annealing for kernel placement on an array of processors (Appendix A), the simulator implementation developed for this analysis (Appendix B), and future directions of interest for this system (Appendix C).

Chapter 2

Background

This background section provides background information on two topics related to programmable embedded systems: a formal framework for describing and analyzing kernel-based applications known as Synchronous Data Flow, and an overview of current streaming processor hardware implementations.

2.1 Synchronous Data Flow

The concept of data flow programming has been elegantly codified into the Synchronous Data Flow (SDF) application descriptions [33]. SDF originated from the desire to map signal processing algorithms to DSPs efficiently, and as such focuses on the static scheduling of computation kernels (actors) operating on streams of data (tokens). The SDF description builds an application from a graph of actors with statically determined token consumption and production rates. The actors are connected by a directed graph edges, or FIFOs, which can optionally specify delays. By insisting on static data rates, the application can be analyzed to determine the optimal schedule for firing (executing) the actors to minimize buffering and scheduling overhead, while avoiding deadlock. The application definition is quite general, allowing multiple inputs and outputs to actors and feedback loops within the application.

SDF applications can be analyzed by putting the static production and consumption rates of each actor on each edge into a *topology matrix*. The topology matrix can

then be manipulated to determine a *static* Periodic Admissible Serial Schedule (PASS) for the application. Such a schedule defines a repeating firing pattern for all the actors in the graph that guarantees that the size of the buffers between each actor does not increase between iterations. The PASS thereby defines the maximum buffer sizes required along each edge within the graph. The derivation and proof of such a schedule reveal several nice properties of this description of the application including proof of the existence of such a schedule and the requirements for a consistent application description. The initial work on SDF also introduced a reasonable methodology for generating parallel schedules and acknowledged the possibility of trading off schedule length for buffer size. The limitations of the SDF model are in its static nature. By itself it can not handle state changes or data dependencies in the structure of the application.

To enable applications with state dependencies, SDF has been extended to encompass a Cyclo-static Dataflow (CSDF) approach [13]. This approach allows each actor in the graph to have a cyclical sequence of statically defined production and consumption rates. The application can then be analyzed much like a regular SDF application by expanding the topology matrix to contain vectors representing the possible firing states for each actor. The resulting static schedules are necessarily much longer than the regular SDF ones as they must encompass the correct number of internal cycles for each of the actors.

CSDF applications can also be translated into SDF form [39]. Such a translation has the advantage of allowing the use of methods for scheduling and analyzing the simpler SDF graphs, but has the potential cost of exponential schedule growth. If the CSDF model is explicit as to which actors have internal state, the transformation to SDF can expose additional parallelism that would not have been present in a purely CSDF representation. The overall increase in flexibility provided by allowing each actor to have an internal cyclical consumption/production pattern makes SDF significantly more general, but greatly (indeed, sometimes exponentially) increases the complexity of scheduling and analysis.

Subsequent work on generating schedules for SDF applications focused on minimizing the memory resident sizes of the resulting applications, as reviewed in [5].

Across a range of applications targeted at DSPs, the size of the code for the compiled SDF applications was found to dominate over the required buffering between the actors due to the need to inline the actor code to avoid the historic overhead of function calls. This led to several approaches to generate minimum sized schedules by focusing on obtaining Single Appearance Schedules (SASs) wherein each actor's code need only be inlined once. SAS schedules exist in general only when each strongly connected component of a SDF graph itself has a valid SAS schedule. This leads to algorithms which partition graphs into strongly connected (tightly interdependent) components, for which an SAS schedule can not necessarily be found, and "subindependent" components for which one can. To obtain decent schedules for the tightly interdependent components, the SDF graph must be ordered in such a way that each actor can be fired the correct number of times in a row without stalling, if possible. Two heuristics for finding such an ordering are Acyclical Pairwise Grouping of Adjacent Nodes (APGAN) and Recursive Partitioning by Minimum Cuts (RPMC). These two approaches complement each other in that they work well on different types of graphs, and generally construct SAS schedules for acyclic SDF graphs that minimize the buffer sizes required, and have been extended to cyclic graphs.

While SAS schedules produce the smallest inlined code for SDF applications, the justification such code-size reduction efforts has diminished over time. The drive for SAS schedules was based on the need to inline all actor code to avoid the overhead of function calls and dynamic dispatch. As caches have become larger and ubiquitous on even the smallest DSPs, and as many-core processors lead to communications being the dominant performance concern, dynamic scheduling and dispatch of actors is rapidly becoming a small, or even negligible, overhead [48]. As much of the work in SDF in general has focused on obtaining optimal *static* schedules for DSPs, it is not clear how valuable this model will be moving forward.

2.2 Streaming architectures

Streaming architectures have arisen in response to the growing disconnect between the amount of computational resources that can fit onto a chip and the amount of off-chip

bandwidth available to feed them. At the same time, a push towards multi-core architectures has been seen due to the increasing power cost and decreasing performance benefits of trying to extract dynamic parallelism from a single instruction stream. Combined, these two trends lead to a range of architectures that feature multiple software-controlled processing units and explicit software-controlled data movement. The most relevant academic projects in this regard are MIT's Raw [45] and Stanford's Imagine [40] architectures. In industry, the IBM-Sony-Toshiba Cell processor [21], Stream Processor's Storm-1 [28], a broad range of graphics processors, and a healthy collection of special-purpose embedded architectures are making commercial stream processing a reality.

2.2.1 MIT's Raw

The Raw microprocessor architecture [45, 42] was designed to implement the minimum set of features in hardware to expose the hardware resources to the compiler. The driving motivation was that as devices become smaller wire delays become more expensive and the opportunities for parallelism and specialization increase. This led to a tiled architecture that exposed the logic (functional units in each tile), wires (on-chip networks), and I/O pins (via the chip edge network ports) at the ISA level to allow the compilation system to make best use of the resources. By exposing this level of architectural detail to the compiler, the Raw project aimed to enable a broad range of applications to execute efficiently and scale well.

The Raw implementation consists of an array of identical tiles, each containing a MIPS-like 8-stage, in-order, single-issue pipelined processor and a network interface. The network interface supports four networks: two static and two dynamic. To encourage the use of multiple tiles to exploit ILP, the networks are mapped directly into the register files in the processors. This enables a static route to communicate register values between functional units in adjacent processors in only 3 cycles. The dynamic networks are divided into a privileged network for cache, DMA, and I/O access, and a general purpose network. The processors contain simple floating point

units, 32kB of data cache, and 32kB of software-managed instruction memory,[1] while the network interfaces contains 64kB of instruction memory, thereby enabling great flexibility in the scheduling of the static networks.[2] To enable software control of the I/O resources, the on-chip networks are simply routed to the I/O pins, and can thereby be accessed with regular network commands.

2.2.2 Stanford's Imagine

The Imagine streaming architecture [40] was designed to execute media processing kernels with high arithmetic intensity efficiently [27]. The focus was on providing a good hardware match for both the instruction- and data-level parallelism present in

[1]It appears that not including an instruction cache was a significant issue for development as almost all reported results are run on a simulator with an instruction cache.

[2]Unfortunately, this flexibility in the network router resulted in them being roughly as large as the processor itself.

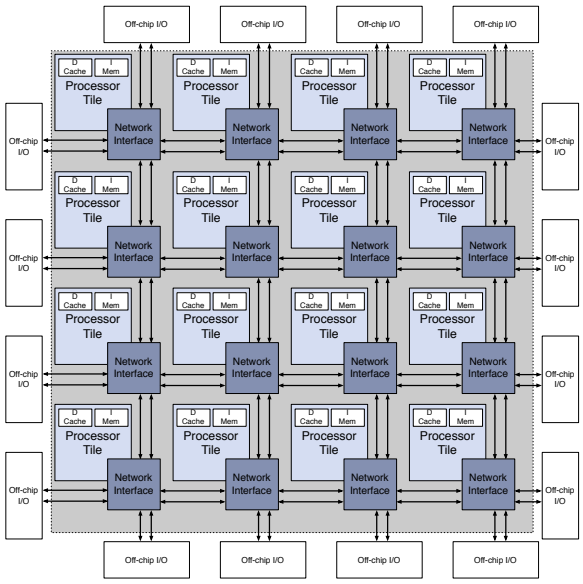


Figure 2.1: Raw architecture
The Raw architecture consists of an array of 16 tiles, each with a RISC processor and network interface. The processors have data caches and software-controlled instruction/scratchpad memories. The network interfaces terminate in I/Os at the edge of the chip.

media applications, while improving memory system efficiency by restricting access semantics to regularly indexed streams. For applications that map to this architecture, it was able to achieve very high utilization and efficiency.

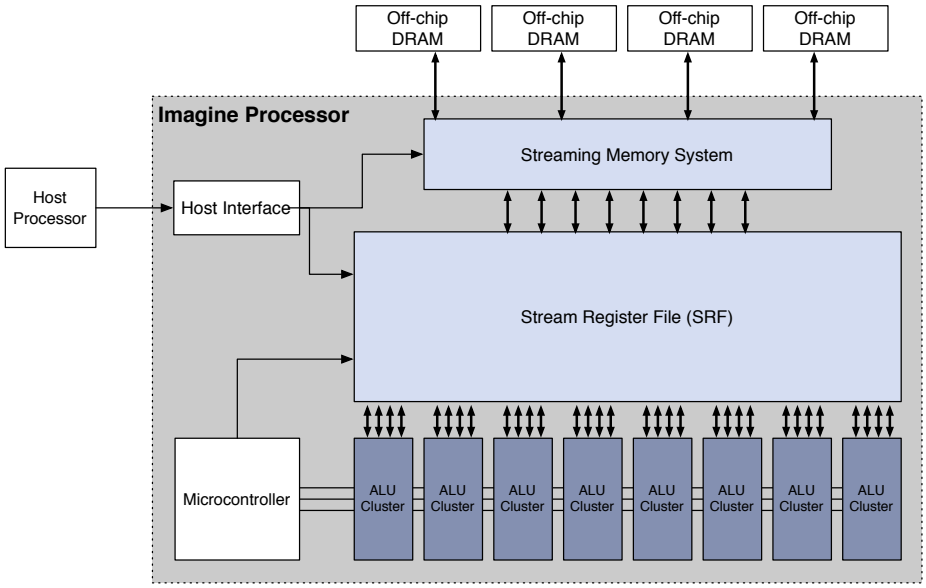


Figure 2.2: Imagine architecture
The streaming register file (SRF) provides a data staging area and bandwidth multiplication between the off-chip memory and the VLIW processing lanes. Figure after from [40], Figure 3.

Imagine executes SIMD-style across 8 VLIW computation "lanes". Each VLIW lane is designed to take advantage of the ILP present in the kernel. Data-level parallelism is exploited by having 8 lanes operate together in a SIMD-manner. The architecture functions as a streaming co-procesor and relies on a host CPU to load stream memory instructions and kernels to a small hardware scoreboard. Imagine then issues the memory instructions and kernels when their scoreboard dependencies have been satisfied, executing up to two stream memory instructions and one kernel instruction at a time.

Streams are loaded from off-chip memory into an on-chip streaming register file (SRF) which provides high-bandwidth regular data access to the computation units. The computation kernels are executed by an on-board micro-controller that issues

instructions across the 8 VLIW lanes in a SIMD fashion with each lane receiving the n-th data element from the data stream(s) and writing out the n-th result. The SRF is the only global memory structure accessible by the computation kernels, and it can only be accessed in either strided or indexed fashion. This access-pattern restriction enables the memory sub-system to efficiently use its bandwidth by pre-fetching data for the processing lanes. In addition to accessing the SRF, each lane can read and write scalar control variables to the micro-controller. Inter-lane communications is provided by an exchange instruction which allows the lanes to rotate data in an arbitrary, but static fashion. As Imagine only executes one kernel at a time, the only task-level parallelism that can be utilized is that of pre-loading streams to the SRF.

2.2.3 Others Streaming Architectures

Mainstream and Server Processors

As the performance return for more aggressive single-core processor design has diminished at the cost of greatly increased power consumption, all major processor designers have shifted to placing multiple processor cores on a die to achieve higher performance and better market penetration. This trend is rapidly moving commodity computing to the point where CMP platforms dominate the desktop, starting at first with dual-core, and moving rapidly towards quad-core, and beyond.

In addition to mainstream architectures, server manufacturers are also moving towards many-cored processors. Sun's Niagara [31] is an 8-core hardware multi-threaded UltraSparc processor optimized for transaction processing. It competes with Azlu's Vega 2 [12], a 48-core RISC-style processor designed for running virtual machines and transactional code. Even non-procssor companies such as Cisco have moved in this direction. Their CRS-1 [47] contains 192 Tensilica Xtensa cores and is designed for network packet processing.

While none of these processors are designed for streaming per se, [19] has shown that they can benefit from the data pre-fetching enabled by streaming applications, and that it would only take a small amount of hardware to fully enable streaming

processing within their architectures [18]. As the number of cores increase, data-parallel applications will require extensive analysis to ensure that the memory system can keep the cores busy, thereby making stream-enabling hardware enhancements all the more appealing.

Commercial Streaming Architectures

Commercial streaming architectures first appeared with the Sony-IBM-Toshiba Cell processor [21]. This processor provided 8 streaming processors with SIMD execution and software-managed shared instruction and data memories, along with a single dual-threaded RISC control core. The individual stream processors, referred to as Synergistic Processing Units for obscure reasons, communicate over a ring network in a non-coherent fashion. ClearSpeed's CSX600 [20] is a 96 processor floating point accelerator with 1TB/s internal network bandwidth. The picoArray PC102 from PicoChip [11] contains a heterogeneous array of 322 processors. The PC102 includes 3-way VLIW Harvard architecture units, control processors, and memory tiles, all connected by a statically programmed network of 32-bit busses. The Tilera Tile64 [46] is a commercial realization of the MIT Raw [45] architecture. Its initial implementation contains 64 VLIW cores with 5-on chip networks supporting user and system static and dynamic access. The Tile64 provides coherent shared-memory and keeps Raw's register-mapped network support, but adds hardware support for network endpoint buffers and more industry-oriented I/O. Similarly, the Storm-1 from Streaming Processors Incorporated [28], is a commercial fixed-point implementation of Stanford's Imagine [40] streaming processor. The Storm-1 doubles Imagine's 8 VLIW processing lanes to 16, and adds extensive support for full system integration and industry-standard I/O.

Intel has also disclosed two prototype streaming processor designs. The first, the Teraflop Research Chip [22], is an 80-core floating point processor with a simple on-chip mesh network and no memory coherence. In addition to exploring the construction of on-chip networks, the chip serves to test techniques for 3D integration by stacking SRAM dies on top of the processor die. More interestingly, Intel's

Larabee project [7] presents a 16-core x86 architecture designed to be performance-competetive with the special-purpose hardware of contemporary graphics processors. The initial design calls for multiple hardware threads per core, high-performance short-vector units, a large coherent, but distributed, cache, and specialized hardware texture fetches.

Graphics Processors

Graphics Processing Units (GPUs), such as NVIDIA's G80 and ATI/AMD's R600 architectures, are effectively stream processors. Historically, GPUs have been special-purpose processors with fixed-function hardware to implement the graphics processing pipeline required to render real-time 3D images. However, as the rendering requirements have evolved (e.g., with the addition of programmable shaders and geometry manipulation), the hardware architectures have evolved to be more and more generic. The G80 and R600 generations have both moved to a generic set of multi-threaded processing units on which the stages of the graphics pipeline are dynamically scheduled and executed. For example, the G80 contains 16 processors with 8 cores each, and each core being time multiplexed across several threads [38]. The cores support 16KB of local shared memory and the system has direct access to high-speed graphics memory, and much slower access to the system's main memory. The R600 contains 4 SIMD arrays of 16 stream processing units each. The processors are 5-way scalar VLIW processors with support for 32-bit floating point multiplication [9].

While the individual processing cores, instruction dispatch, and data memories are still optimized for graphics rendering, these architectures have been successfully used for streaming computations. Indeed, by exposing producer-consumer locality and data parallelism, streaming models map very nicely to the GPU processing model [6]. The manufacturers have recognized this and begun to supply generic programming environments for accessing these resources for non-graphics tasks [34].

Chapter 3

Related Work

The concept of computing on streams of data, or stream computing, has been around for quite a while. Stephens [41] traces it back to P J Landin and his work in the 1960s on ALGO 60, and follows it as an active research and development topic up through the late 1990s. More recent developments are sumarized in [16, 15]. This section focuses on several particularly relevant contemporary streaming languages for media processing (StreamIt, StreamC/KernelC, Brook, and Sequoia) and the generalization of Synchronous Data Flow to multiple dimensions. Particular emphasis is placed on the StreamIt language and its implementation for the Raw hardware back-end as it closely resembles the target hardware described in this thesis. For an in-depth review of previous work in streaming programming languages, please refer to the two aforementioned references.

3.1 StreamIt

One of the most well-explored example of a streaming programming language is StreamIt. The authors summarize the distinctive features of the language as follows:

> "StreamIt differs from other stream languages in the single-input, single-output hierarchical structure that it imposes on streams. This structure aims to help the programmer by defining clean, composable modules that admit a linear textual representation."[43, 23]

StreamIt has been targeted to many-core architectures [16], clusters [44], and portable streaming frameworks [48]. In addition, multiple generations of the compiler have explored a broad range of optimizations across many large benchmarks [23, 16, 15].

StreamIt represents applications as a hierarchical series of *filters* (e.g., computation kernels). The hierarchy comes from encapsulating multiple related filters in Pipeline, SplitJoin, and FeedbackLoop connecting filters. (See Figure 3.1 reproduced from [16], Figure 3.) Pipelines define a series of filters where the first sub-filter's output feeds the input of the next filter, and so on. SplitJoins are parameterized round-robin filters that duplicate or distribute the input data amongst multiple filters and collect the result. FeedbackLoops contain explicit "body" and "loop" filters and the description of what data is fed-back. The authors claim strongly [43, 16] that by enforcing this specific notion of hierarchy, instead of allowing the "flat and arbitrary network of filters" used by other streaming languages, applications written in StreamIt are both easier to write and easier to compile.

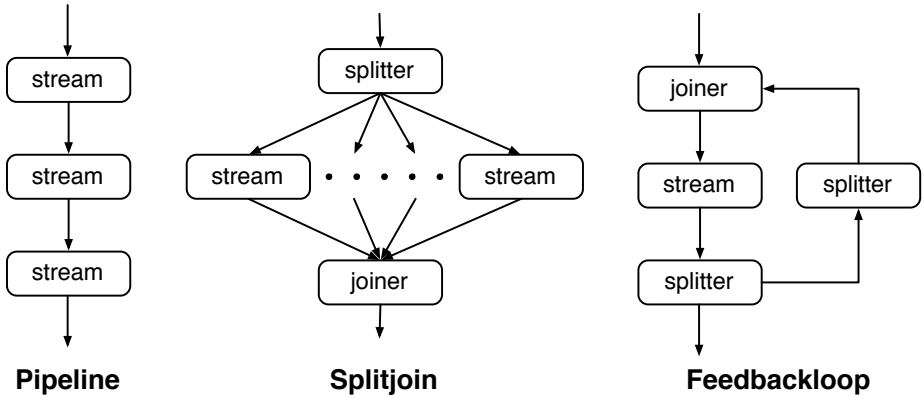


Figure 3.1: StreamIt supported hierarchical structures
Figure reproduced from [16], Figure 3. The "streams" can be other hierarchical structures or filters.

Filters are defined in terms of the number of input values they use and the number of output values they generate per iteration. As the StreamIt inputs and outputs are one-dimensional streams, and each filter has only one input and output, data access is defined in terms of the number of input values the filter *pops*, *peeks*, and *pushes* on

each iteration. Data that is popped is consumed on the current iteration, and pushed data is generated to be sent downstream to the next filter via the output. The amount of peeking required specifies how much data a filter reuses between iterations, as this data will still be available on the next iteration. StreamIt can handle two-dimensional data, but to do so the filters all need to know how to index into the one-dimensional stream to treat it as a two-dimensional array. This complexity limits the utility of `peek` for expressing reuse to the compiler in image processing applications. This limitation is incurred because the reuse must be specified in the one-dimensional mapping of the two-dimensional image, which may not accurately reflect the actual two-dimensonal reuse patterns.

Each filter in a StreamIt program contains three main methods: `init()`, `prework()`, and `work()` [43, 16] (see Figure 3.2). The `init()` method is called when the application is built and configures the filter. For example, the Pipeline connecting filter's `init()` function instantiates the filters to be included in the pipeline and connects them up as needed. The main execution of the filter occurs in `work()`, which is executed whenever the kernel is scheduled to be run. The `prework()` method is called before the first execution of `work()` to allow filters to generate sufficient outputs to fill the initial peek range required for downstream filters. Although undiscussed, it appears that the programmer must manually determine how many outputs to generate in the `prework()` method to satisfy downstream filters.

StreamIt provides a *portal*-based method for communicating control information out-of-band from the data streams. By separating this from the data-stream, the authors claim the application code is simplified as it keeps control and processing separate. The initial portal implementation allowed the programmer to manually define the latency range for the delivery of messages and send them via simple function calls. Later work on "teleport" messaging [44] enhanced this approach by applying analysis from the Cyclo-static Synchronous Data Flow (CSDF) domain [13] to improve the latency guarantees and reduce the overhead. Through these portals, StreamIt applications can accomplish one-to-many communications and provide multiple control outputs from a single filter. Except for enforcing the required delivery latency, the authors argue that these communications can be ignored in terms of their impact on

```
float->float filter FIRFilter (float sampleRate, int N) {
    float[N] weights;

    init {
        weights = calcImpulseResponse(sampleRate, N);
    }

    prework push N-1 pop 0 peek N {
        for (int i=1; i<N; i++) {
            push(doFIR(i));
        }
    }

    work push 1 pop 1 peek N {
        push(doFIR(N));
        pop();
    }

    float doFIR(int k) {
        float val = 0;
        for (int i=0; i<k; i++) {
            val += weights[i] * peek(k-i-1);
        }
        return val;
    }
}
```

Figure 3.2: StreamIt FIR filter example
Figure reproduced from [16], Figure 1. The `init()`, `prework()`, and `work()` functions are shown along with their definitions in terms of how many items the filter peeks, pops, and pushes.

the data processing schedule, although the language enforces no requirement that the message handlers execute quickly nor occur at a negligible rate.

In addition to fixed push, pop, and peek sizes for the filters, StreamIt applications have static rates that are known at compilation, with each filter's computation time being estimated by a static analysis of its code. This allows the compiler to treat the application as a quasi-Synchronous Data Flow problem [23], thereby allowing the use of a wide range of SDF scheduling techniques [5].[1] The initial motivation for generating static schedules for StreamIt applications appears to come from the need to statically schedule one of the two communications networks on the targeted Raw hardware [45, 16]. Historically, SDF applications have been statically scheduled to eliminate the overhead of dynamically determining the appropriate kernel to execute

[1] Gordon [16] notes that the irregular round-robin nodes in StreamIt require that SDF-type analysis be applied in the context of the more general, but much more complex, CSDF framework.

on a single DSP or processor, but this has become much less of an issue with many-core processors [48]. As later compilers appear to have moved to entirely off-chip buffering for Raw [15][2], thereby reducing the dependency on the on-chip network, it is not clear why so much emphasis is put on obtaining a static schedule.

The StreamIt compiler [16, 15] implements a variety of application transformations to try and achieve maximum performance across different benchmarks. The compiler tries explicitly to achieve the right balance of task-, data-, and pipeline-parallelism to maximize the utilization of the 16 processors in the Raw architecture while minimizing the overhead introduced due to synchronization and communications. The transformations are necessary, because although the StreamIt representation itself contains much parallelism, it may not map well or be well balanced. For example, the task parallelism exhibited by filters on different branches in the application requires significant synchronization if mapped directly to individual processors, and is unlikely to provide sufficient parallelism for the target architecture. If the data parallelism present in stateless filters is exploited for filters with low computation-to-communications ratios, the result will increase communications overhead and buffering. Similarly, pipeline parallelism from chains of producers and consumers may be plentiful, but is likely to exhibit poor load-balancing.

To alleviate these issues, the StreamIt compiler [15] utilizes two broad transformations. The first is to increase the granularity of the parallelism of data-parallel filters by fusing them with their producers and consumers. This increases the computation-to-communication ration for the fused filter. However, to avoid introducing "internal state" in the form of inter-kernel buffering, data-parallel filters are only fused when their peek ranges are small enough. This limitation is due to the buffers being integrated within the kernel implementation, and therefore hiding data that other instances might need. Once fused, these data-parallel kernel chains can then be split to generate the required parallelism to fill the available processors, while maintaining the increased computation-to-communication ratio. To avoid eliminating task-level parallelism from the application, the original task-level parallelism is maintained in

[2]This does not appear to have been motivated by resource constraints as the buffers were noted to be small enough to fit on chip, but rather due to the complexity of implementing them for the Raw architecture.

this process and the data-parallel filters are split only sufficiently to fill the remaining processors. For good results, a the compiler attempts to balance the level of data-parallelism and task-level parallelism in the application to ensure that each final filter (whether originally a task-parallel portion of the application or as a result of a fused and then parallelized data-parallel chain) represents approximately $\frac{1}{n}$ of the total work, for n processors.

The second transformation is applying a coarse-grained software pipelining to the application. Pipeline parallelism is needed to allow filters with data dependencies to execute in parallel by having them operate on different iterations together. StreamIt eschews simply mapping filters to processors and executing them dynamically as data is available because of the potential overhead of the dynamic dispatch, and the inability to use the Raw architecture's static communications network for the required dynamic data transfers. Instead the StreamIt compiler builds a static coarse-grained software pipelined schedule whereby the steady-state kernel executions access input and output buffers, followed by a separate communications stage. This approach allows the steady-state kernel executions to be scheduled completely independently, which potentially enables better load balancing. Unfortunately, static scheduling was not compared to a dynamic dispatch, so the true overhead is unknown.

3.2 StreamC and KernelC

The streaming language developed for the Imagine streaming media processor [40] divided applications into two levels of hierarchy. Computation kernels were described in KernelC and the application connectivity and data movement in StreamC [35]. This explicit separation of languages allowed the compiler efforts to concentrate more-or-less independently on efficiently compiling each language.[3] KernelC was targeted to Imagine's SIMD streaming architecture, which requiring efficient exploitation of ILP through deep unrolling and sophisticated resource allocation, but only in the context of a single kernel [36]. Compilation of StreamC focused on efficient scheduling of data

[3]Unfortunately, accurate data scheduling is closely intertwined with the performance optimizations applied to the kernels, making this separation only true to a first approximation.

movement through Imagine's large single streaming register file (SRF), to hide latency while the computation kernels executed [8]. The separation of these two tasks into separate languages handled by separate compilers recognizes the hierarchical nature of the problem, and is similar to the high-level partitioning and placement done for StreamIt, while the low-level kernels are compiled by a reasonably standard MIPS backend [15].

The StreamC/KernelC programming system dealt with data as either scalar configuration values or streams of records [35]. At its most basic, a record is simply the size of the data a given kernel reads from an input or writes to an output. However, by allowing new streams to be derived from existing streams, the definition of a record enabled flexible data access within the streams, while still providing the compiler with a high-level knowledge of how the streams were manipulated. For example, by defining a derived stream to have a stride equal to the width of an image, a two-dimensional image encoded in a one-dimensional stream could be more readily accessed by a kernel as it would read in a line at a time. KernelC additionally allowed the definition of kernels with multiple inputs and outputs, but each execution of the kernel could only access one iteration of the input at a time.

To obtain high throughput on the Imagine architecture, the essential optimization was to hide the latency of loading streams (or portions of streams) into the streaming register file by scheduling execution to overlap the kernel computation with the loading of the next set of inputs [8]. This problem boils down to an NP-hard dynamic storage allocation problem for the time-space usage of the SRF. The problem is particularly challenging because the SRF allocation is directly coupled to the program execution. This implies that limits on the capacity of the SRF feedback into the details of the kernel scheduling, which may result in further changes in the original SRF usage. The stream scheduler attempts to minimize execution time by improving concurrency between stream operations (loads and stores) without introducing unnecessary spills to the off-chip memory. In order to implement this scheduling, however, the runtimes and sizes of the streams and kernels must be statically known. This is determined by a static analysis of stream sizes and a combination of static and profiled kernel runtime analysis, made possible by the limited control structures

allowed in the KernelC kernels.

Unfortunately the size of the SRF is often small compared to the declared size of the streams. This requires that the streams be double-buffered, which limits the effective bandwidth to that of the off-chip interface. To overcome this limitation, stream programs can be strip-mined, whereby the larger streams are broken into strips and all execution for one portion of the input strip is executed at once. This can reduce the working set size sufficiently to allow the inputs and intermediate results to fit into the SRF, thereby boosting the effective bandwidth to that of the on-chip SRF. In the StreamC stream compiler this is implemented by having the programmer manually place a looping construct around each large stream usage, but the size of the strip is determined by the compiler.

3.3 Brook

The Brook language [6] has enabled stream programming of graphics hardware. Its managed to largely abstract the details of the rather primitive level of programmability provided by the fragment shader engines at that time. The language extends C to provide data-parallel constructs and the concept of streams, kernels, and parallel reduction operators. Applications in Brook are written as a collection of kernels with input and output streams. The streams may have multiple dimensions and may consist of user data types. Kernels are allowed to access streams in strided patterns or indirect patterns, but all other inputs are read-only constants. Brook does not allow kernels to have input and output sizes that do not match. When this is encountered the input stream is either replicated or decimated to ensure the sizes match.

The Brook compiler and runtime system handle transferring streams as textures to the GPU and executing kernels on those textures. Brook includes back-ends and runtimes for targeting both GPUs and CPUs. Unlike the stream scheduling for StreamC/KernelC, however, Brook does not attempt to modify stream loads and kernel execution order to improve performance. With Brook as an intermediary, the GPU can be viewed as a streaming processor, thereby making it possible to implement similar transformations.

Brook's attempt to make a generally programmable interface to GPUs revealed several areas in which the hardware could be extended to ease programmability. As discussed in Section 2.2.3, GPUs have increased in their generality, and the vendors are beginning to provide their own languages and compilation systems for accessing their processing power. For example, NVIDIA's Complete Unified Driver Architecture (CUDA) [34] language provides programmers with direct access to each of the thousands of threads running on the GPU, but does not provide any high-level program transformations for staging and managing data and computation.

3.4 Sequoia

Although not technically a streaming language due to the explicit nature of its memory address calculations, the Sequoia programming language [30, 14] has many of the same goals. Sequoia achieves high performance on a variety of parallel processing platforms (clusters, SMPs, CMPs, Cell) by analyzing application data movement and proactively moving data to ensure that dynamic data access does not slow down computation. This approach is essential for architectures with software-managed memories, and can produce significant speedups on cache-based systems by providing more intelligent prefetching than hardware alone.

To enable this level of compiler analysis, Sequoia requires that programmers explicitly divide programs into data movement and computation steps. Data distribution is described with parameterized n-dimensional bulk operations which can partition and copy data in a variety of compiler-analyzeable manners. The computation is described as acting on some parameterized subset of the data. The program is mapped to the target hardware by building a tree of nodes where the leaf nodes execute the actual computation on the processors, and the remaining nodes handle distributing the data down to the leafs, with each level in the tree corresponding to a level of memory in the architecture. The user must parameterize how much data is kept at each node, but the analysis and code generation to implement it are automatic.

Sequoia's biggest limitation is its fundamental assumption that memories exist in vertical hierarchies. The only way data can be shared or moved between processors

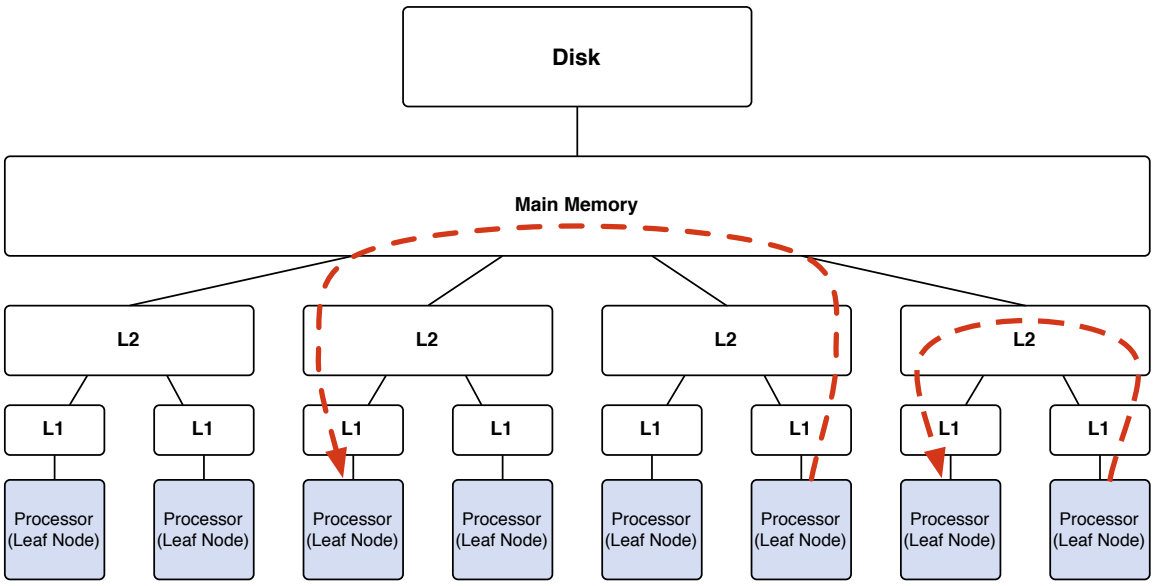


Figure 3.3: Sequoia hierarchical memory model

The hierarchical memory model supported by Sequoia is shown for an 8-way SMP machine with two levels of cache. Processors run the actual computation tasks, while data movement tasks execute logically at the different levels of the memory hierarchy, although they may be implemented as threads running on the processors depending on the capabilities of the architecture. Data sharing between processors is indicated by the red dashed lines, showing how the memory hierarchy must be used for communication, regardless of the presence of any direct inter-processor links.

in Sequoia is via a memory node which is a parent to both of the child processors, as illustrated in Figure 3.3. This maps well to traditional systems where a large disk backs a smaller memory, which in turn is backing for more and more smaller and smaller levels of cache, but it breaks down when direct processor-to-processor transfers become important. The addition of "virtual levels" in the Sequoia memory hierarchy helps to alleviate this problem, but does not enable general analysis of direct processor-to-processor communication. For many-core architectures, this approach ignores the high bandwidth and efficiency that can be achieved from direct processor-to-processor communications.

3.5 Multi-dimensional Synchronous Data Flow

It has long been recognized [32] that extending the SDF framework (See Section 2.1) from one-dimensional inputs to a more general multi-dimensional processing would be of great value to signal processing, and image processing in particular. The "Generalized Multidimensional Synchronous Data Flow" GMDSDF of [37] was an extremely complex attempt to attain this generality. The focus was on extending SDF to n-dimensional inputs and outputs and allowing non-rectangular sampling grids. In particular, GMDSDF provided transforms to warp, resample, and adjust the dimensionality of the inputs and outputs. This generality was motivated by the desire to represent 3D matrix multiplication and video format conversion using language constructs to implicitly express the data access patterns. To attain this flexibility, however, the approach relied on defining a "fundamental parallelepiped" which defined the iteration direction and size with which the kernel uses data from its input. The awkwardness of this generality was not lost on the authors who commented that, "it is not clear at this point whether these principles will be easy to use in a programming environment."[37]. Indeed, mapping this onto the SDF framework resulted in Cyclostatic Syncrhonous Data Flow (CSDF), which is significantly harder to work with. A subsequent attempt to implement the GMDSDF ideas in the Ptolemy development environment appears to have been fraught with the difficulties of actually implementing the complex transformations required to support the generality.

Windowed Synchronous Data Flow

Significantly more success was had adapting SDF to fixed rectangular sampling grids. The "windowed" SDF (WSDF) of [26, 24] allows n-dimensional data use and reuse to be represented in a SDF framework. The representation allows the specification of border extensions or contractions to match data sizes, but does not do so automatically. To achieve this, the authors define "virtual tokens" which can be built up of output tokens from a source actor to produce the required input size for a sink actor. This allows the actor to determine when it can fire at a finer granularity than SDF, which has the potential to result in significantly smaller buffer requirements.

Using WSDF it is possible to calculate the minimum required buffer sizes for applications [25]. As SDF applications do not specify the order of the firing of the individual actors, and the sizes of the buffers between actors for WSDF applications depends heavily on the firing order, the approach taken was to simulate the application for various firing orders and count the maximum required buffer sizes for each application repetition. By using a firing order that only fires a source actor when its sink is stalled waiting for input, they claim they can find the minimum required buffer sizes. However, the execution time for this simulation-based approach is significant even for simple kernel pairs, and is likely to scale very poorly for larger applications where choices are interdependent. The difficulty of analyzing data movement between kernels in WSDF is due largely to the generality of its data access descriptions and the lack of a specified data access ordering. Combined these present a very large design space for possible implementations for even the smallest applications.

Array Oriented Language

A non-SDF based approach to describing similar multi-dimensional signal processing applications was taken with the Array Oriented Language (Array-OL) originally developed for multidimensional sonar processing[3, 4]. Array-OL describes the data dependencies of the application but does not specify the execution order of the application. In this sense it is not explicitly a *dataflow* description, and the implementation must choose an appropriate (and hopefully optimal) ordering for execution. Indeed this approach taken to an extreme wherein time is an explicit infinite-length dimension in the input matrices to the application. While this complicates analysis, as this single infinite-length dimension must be handled, it results in an elegant data model where time can be treated as just another dimension rather than a special concept.

Applications in Array-OL are described hierarchically. Global levels consist of kernels connected via data streams, where the streams consist of toroidal arrays of data. (e.g., the arrays are assumed to wrap around at the borders.) The data is processed at the local level by the kernels in "patterns", which are defined in by a series of matrices which specify their origin, shape, paving, and fitting (see Figure 3.4). Patterns are then "paved" to fill an output array. This pattern definition allows

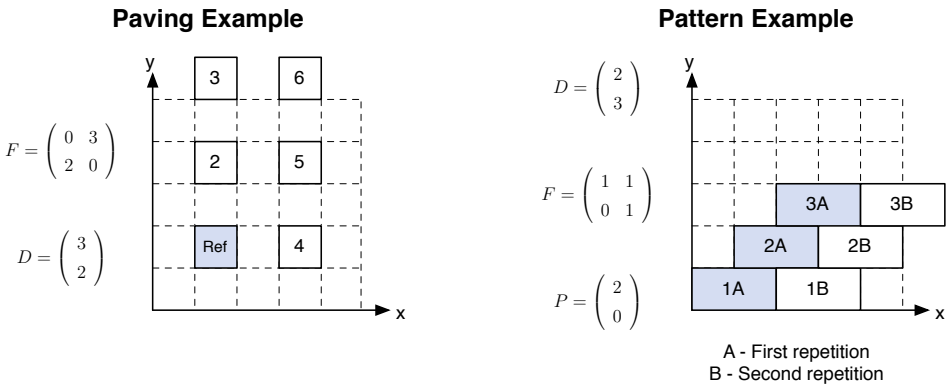


Figure 3.4: Pavings in Array-OL
Examples of complex paving and patterns for mapping data in Array-OL to arrays. The D and F matrices define the shape and fitting, respectively, of the data access. The P matrix defines the paving. Combined, these matrices define the order in which data is read in by a kernel from a two-dimensional input array or written out to a two-dimensional array. For a detailed description of how these matrices are used, please see [4]. Figure after Figures 1 and 3 in [4].

for very complex data access patterns within the kernels, including decimation in arbitrary orders, reuse, windowing, and dimensional transformations. The flexibility is significantly greater than that provided by WSDF, but much less complex than that of GMDSDF. However, the flexibility of this definition makes certain operations on Array-OL applications, such as merging kernels very complex in the general case.

Array-OL application descriptions can be mapped to distributed process networks for execution with only a few transformations. These include dividing up the incoming data's infinite dimension to fit in a finite memory, and inserting kernels to manage transforming the array sizes between kernels. While these portions of the application transformation are relatively straightforward, the application description provides no hints as to how to partition and distribute the kernels amongst multiple processors. As the processing order is also not defined, the space of possible implementations is enormous, and highly architecture dependent.

3.6 Summary

The related work presented here covers a broad range of stream programming systems. The most well-understood, from a theoretical point of view, are the SDF systems. However, SDF suffers from the historic drive to obtain an optimal static schedule and a lack of practical implementations for handling multi-dimensional data. The work on StreamIt has provided a SDF-like system with significant advances in compilation for multi-core architectures. Unfortunately, StreamIt's choice to limit filters to a single input and output and one-dimensional data make implementing and analyzing many applications, in particular image processing, difficult. StreamC/KernelC and Brook broke away from this purely SDF background, but did so in different directions. StreamC/KernelC applied a SIMD model to the computation while Brook exposed data-parallelism through explicitly multi-dimensional data. Sequoia stands out from this group of applications as it is not technically a streaming language, but instead manages a more traditional memory hierarchy to try and obtain similar efficiency. While Sequoia handles multi-dimensional data well, its reliance on a vertical memory hierarchy for communications and sharing limits its applicability to streaming processors and mutli-core architectures.

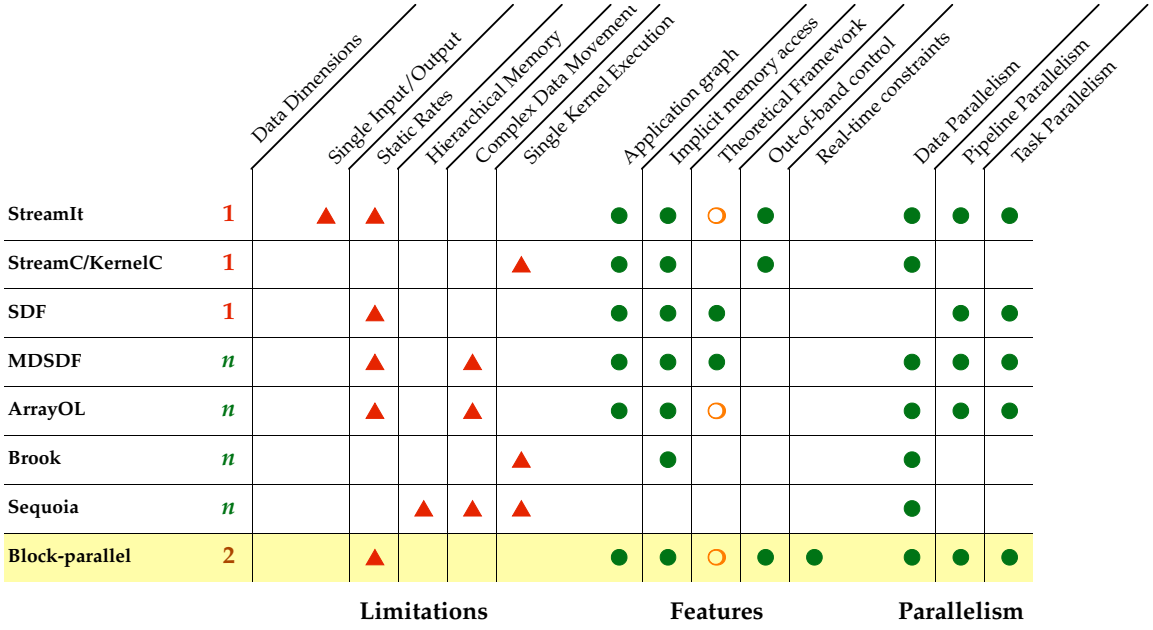


Figure 3.5: Comparison of Related Work

Red triangles indicate limitations in the language; solid green circles are features. The orange circles indicate partial features. In the case of StreamIt, the language conforms to a CSDF framework, but is not analyzed as such. Array-OL is also close to CSDF but is analyzed differently. The block-parallel framework presented here could conform to CSDF, but the regular data movement enables much simpler analysis.

Chapter 4

Application Model

The block-parallel programming model provides an easy-to-use kernel/stream based application description that is readily amenable to compiler analysis. The description incorporates sufficient information in the application description to allow automatic compiler analysis and manipulation at a high-level. However, it does so without impinging the low-level flexibility of the programmer, nor overly-burdening him with the need for detailed knowledge of either the hardware or the data and computation rates.

Applications are built from a collection of computations kernels connected together by data streams to form an application graph, as shown in Figure 1.2. The description statically specifies the data sizes, rates, and resource requirements for each of the computation kernels. The data streams consist of natively two-dimensional data,

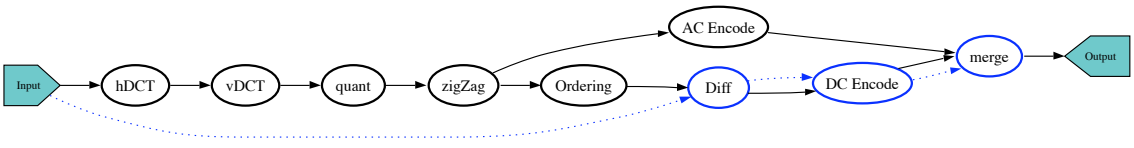


Figure 4.1: Simplified application graph for JPEG compression
The JPEG application shown here includes explicit data dependency edges (blue dotted lines) that express the limited parallelism of encoding the DC components.

as two-dimensions significantly simplifies the writing and analysis of image processing applications, without incurring the full complexity of general n-dimensional data support. An example application graph for a JPEG compression program is shown in Figure 4.1. The block-parallel approach presented here provides a number of enhancements over previous streaming languages to improve the application description in both expressibility (usability for the programmer) and analyzability (usability for the compiler writer). The overall result is to provide a kernel-based programming system that facilitates high-level automated analysis while improving programmer productivity.

Enhancements to the basic stream/kernel model include:

- A data movement model that natively incorporates both one- and two-dimensional data, thereby significantly increases analyzability for image-processing application and simultaneously easing programmability.
- Flexible ControlTokens which can be sent over both data streams and separate control streams, thereby enabling simple and concise inter-kernel synchronization. (Section 4.2.2)
- Data dependency edges that allow programmers to specify the level of parallelism permitted for kernels relative to the rest of the application, thereby permitting data-parallel and serial operations to be efficiently described and handled together. (Section 7.3)
- A computation model that allows kernels to define multiple execution methods per kernel, thereby simplifying initialization and enabling different computations to share data effortlessly, while still encapsulating computation and related data within a kernel structure. (Section 4.3)

4.1 The Application Graph

As in StreamC/KernelC [35], applications are divided into two portions: the *application graph*, which defines the connectivity between kernels, and the *computation*

kernels, which define the methods executed on each iteration of the input data. The application graph describes the data flow for the application by connecting computation kernels to data sources and sinks via *data streams*. This approach fits naturally to data-parallel applications such as image- and signal-processing where multiple filters (or kernels) are applied to an input stream of images or signals. In addition, by adding *data dependency edges* between kernels, to explicitly define the allowable parallelism, and allowing *control tokens* to be sent over data streams, to communicate state, the kernel-stream model is extended to naturally encompass a broader range of applications.

4.1.1 Simplified Application Graph

In a simplified form, the application graph is a directed acyclical graph (DAG) of *Application Elements*.[1] These consist of *DataInputs*, *Kernels*, and *DataOutputs*. The hierarchy of application elements is shown in Figure 4.4, and an example of a simplified application graph is given in Figure 4.1. The DataInputs and DataOutputs are the inputs and outputs from the application as a whole, while the Kernels contain the actual computation methods executed by the application.

DataInputs statically specify the size of each frame of input data and the rate at which those frames are generated. They are arbitrarily, but uniformly, assumed to generate data in a left-to-right, top-to-bottom order. (E.g., one row at a time.) One-dimensional applications, such as traditional signal processing, would simply have DataInputs with a height of one. DataOutputs act as sinks for the data they receive, and therefore are not parameterized by size or rate. The Kernels themselves specify the amount of memory they consume and the number of operations they require per execution. This simplified view of the application graph is useful for visualizing the application and for certain graph traversal algorithms. However, it contains insufficient information to fully define the application as will be seen in Section 4.1.2.

[1] The astute reader will note that the use of a DAG prohibits the inclusion of loops, or feedback, in the application description. This has been done to simplify application analysis. Feedback support is discussed in Section 5.2.1.

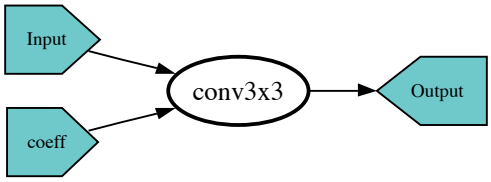


Figure 4.2: Simplified application graph for a convolution program

The simplest program possible within this framework is one which applies a single kernel to a single input generating a single output. An example is applying a convolution kernel to an input image stream. Such a program has two DataInputs (one for the convolution coefficients and one for the input data), one Kernel (the convolution), and one DataOutput. The simplified application graph for such a program is shown in figure 4.2.

An example of a more complex program is shown in Figure 4.3. This program processes an input through two differently-sized convolution kernels, subtracts the result of the first from the second, and then downsamples the result of the subtraction. This "differencing" program contains four kernels along with the DataInputs and a DataOutput. An implementation of the JPEG compression algorithm is shown in Figure 4.1. It contains multiple processing kernels and data dependency edges (dashed blue lines) to limit the parallelism of the serial components of the algorithm. For example, in each image processed by JPEG, the DC components of each 8×8 block are serially subtracted from one another in order. This subtraction step can not be parallelized, as each block depends on the previous one. For the example in Figure 4.1, this is expressed as a data dependency edge from the Input to the DC differencing kernel ("Diff"), indicating that the degree of parallelism allowed for the kernel must be no greater than that of the Input. For JPEG, the DC difference operation is serial across each input frame, so this correctly expresses the serial nature of the computation done by the "Diff" kernel. For this example, the variable length output of the Huffman encoders is accommodated by assuming the worst case.

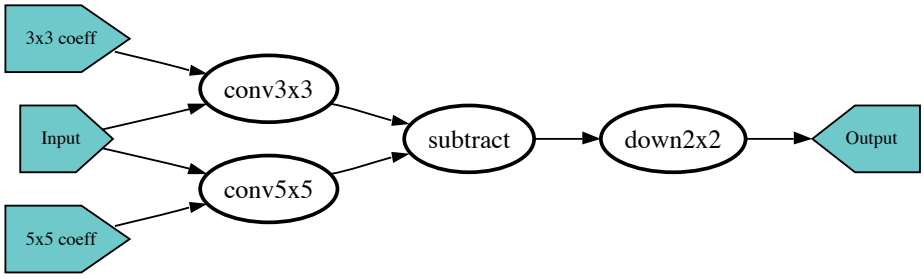


Figure 4.3: Simplified application graph for a differencing program

4.1.2 Full Application Graph

The simplified application graphs discussed in Section 4.1.1 provide an overview of the application, but they do not specify the program completely. The full application graph must additionally contain explicit *Inputs* and *Outputs* between each element. The Inputs and Outputs specify the block size in which the data is consumed or produced, and the step size with which the kernel moves through the input data frame. For example, a 3×3 convolution kernel requires an *input size* of 3×3 for each execution, but between executions it moves over only one column in the x-direction and one column in the y-direction for each new row. The *step size* for the convolution input is then (1,1). From the input size and step size the reuse and output halo of the kernel can be calculated. Calculations of reuse and halos are discussed in Section 4.2.1 and Chapter 5.

In addition to defining data sizes, Inputs and Outputs are added explicitly to the application graph to provide a level of indirection to support distinguishing multiple independent Inputs to a kernel and multiple Inputs driven by a single Output. Allowing multiple Inputs to a kernel is essential for such basic operations as defining filter coefficients (the separate coefficient and data inputs to the convolution kernel in Figure 4.5(a)) and operating on multiple data streams (the subtraction kernel in Figure 4.5(b)). Driving multiple Inputs from a single Output is also a common operation. Both of these topologies are cleanly addressed by explicitly adding the Input and Outputs as nodes to the application graph.

The full application graphs for the sample programs from Section 4.1.1 are shown

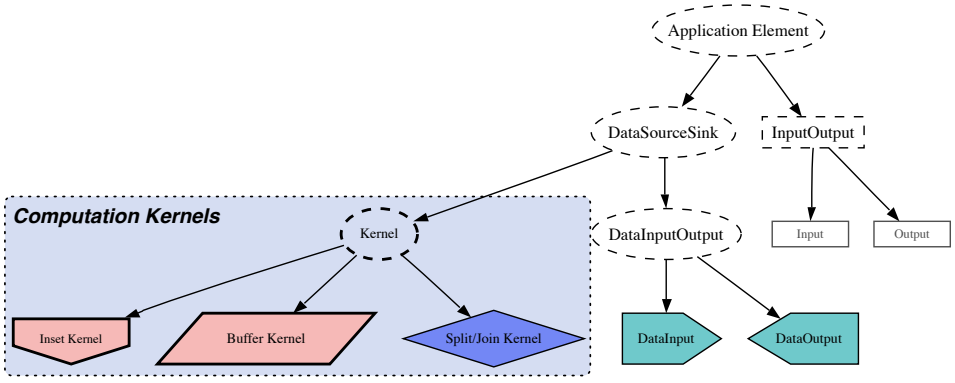
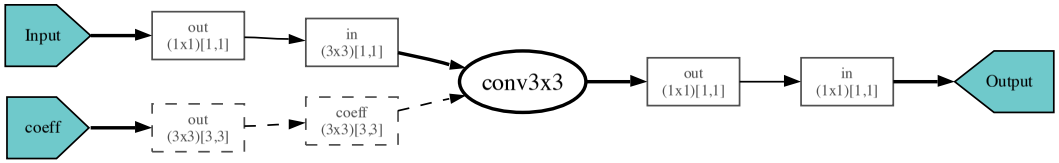


Figure 4.4: Application element hierarchy
The full class hierarchy is shown with the abstract class types indicated with dashed lines. Computation Kernels are constructed by sub-classing from the "Kernel" class. Inputs and Outputs are discussed in Section 4.2.1. Buffer and Inset kernels are discussed in Chapter 6 and Split/Join kernels are discussed in Chapter 7. Pink color indicates kernels that have been automatically inserted into an application.

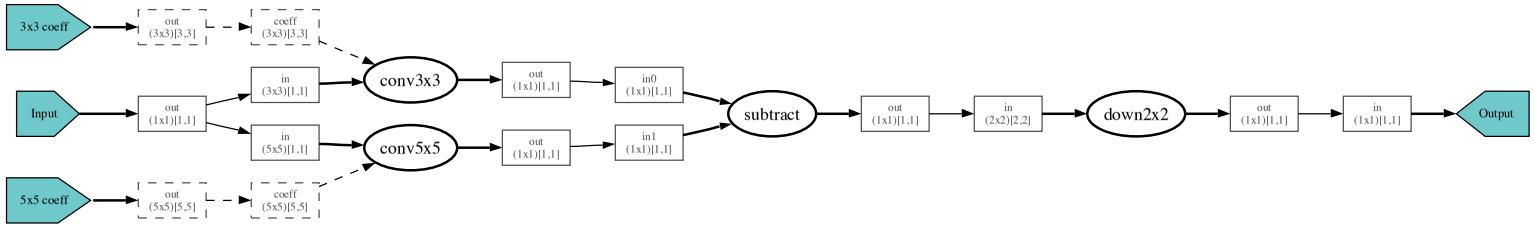
in Figures 4.5(a), 4.5(b), and 4.5(c). While these graphs are significantly more complex than the simplified ones, the addition of the explicit Inputs and Outputs provide a complete description of the application.

4.1.3 Building Applications

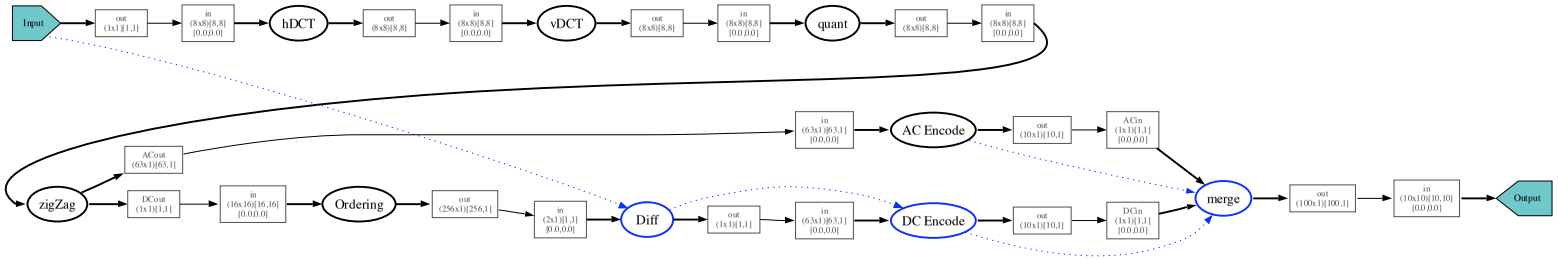
Applications are built by instantiating kernels, adding them to the application graph, and then connecting their Inputs and Outputs to form the desired topology. The code for building the convolution program is shown in Figure 4.6. The code builds the application graph by first instantiating the DataInputs for the input data ("input") and the convolution coefficients ("coeffLoader"), then instantiating the convolution kernel ("conv"), and finally the DataOutput ("out"). If the kernels are parameterized, as is the size of the convolution kernel in this example, their parameters are set when they are instantiated. Once the application elements have been instantiated, they are added to the graph. When a kernel is added to the application graph, its Inputs and Outputs are automatically built, added to the graph, and connected to the kernel. The



(a) Full application graph for a convolution program



(b) Full application graph for a differencing program



(c) Full application graph for JPEG compression

Figure 4.5: Full Application Graph Examples
Dashed edges indicate Inputs that are not data parallel. (See Section 4.2.1.) The parameterization for the Inputs and Outputs in Figure 4.5(c) include the Input and Output offsets, which are discussed in Section 6.2.

```
Application a = new Application("3x3 Convolution");

/*
 * Define the Inputs.
 * For the simulation it will read input from the file "inA.txt"
 * and coefficients form "kern33.txt".
 */
DataInput input =
    new FileDataInput("Input", "inA.txt", 2, 2, 1);
FileDataInput coeffLoader =
    new FileDataInputInitializer("coeff load", "kern33.txt");

ConvolutionKernel conv =
    new ConvolutionKernel("conv3x3", 3,3);
DataOutput out =
    new DataOutputVerifyier("Output");

/*
 * Add the DataInputs and Kernels to the Application Graph.
 */
a.add(input);
a.add(coeffLoader);
a.add(conv);
a.add(out);

/*
 * Connect up the Kernels and DataInputs.
 */
a.connect(coeffLoader, "out", conv, "coeff");
a.connect(input, "out", conv, "in");
a.connect(conv, "out", out, "in");
```

Figure 4.6: Application graph code for a convolution program

The `3,3` in the convolution kernel instantiation code builds a convolution kernel for a (3×3) Input. The `2, 2, 1` in the DataInput instantiation code defines the Input size and rate for the input.

application's connectivity is then defined by connecting Outputs from one element to Inputs on another.

4.2 Data Model

The data movement through the application is determined by the data stream connections (edges) between kernels in the application graph. The data moves over these edges as *Tokens*, which can contain either data or control information. The size of the data consumed and produced by a kernel, and the reuse, if any, of the consumed data, is defined by the kernel's Inputs and Outputs. This data model allows for intuitive and flexible data- and control-driven execution of kernels, within the confines

of a static rate system. This flexibility allows applications to be written in a more straightforward manner than previous streaming systems.

4.2.1 Inputs and Outputs

As discussed in Section 4.1.2, data moves through the application following the edges in the application graph, from an Output to an Input. Inputs and Outputs define the *input size* ($in_X \times in_Y$) of the data consumed or produced on each execution of a kernel. To express data reuse between iterations, Inputs further specify the *step size* (S_X, S_Y) of the input. If the step size is less than the input size, then some of the data is reused between iterations. Conversely, if the step size is greater than the input size, some input is skipped on each iteration. This description of data reuse is essential to allow filters that operate on sliding windows to be implemented without internal state, which is important for automatic parallelization [15]. The input data reuse in the x- and y-direction can be calculated as:

$$reuse_X = (in_X - S_X) \times in_Y \tag{4.1}$$

$$reuse_Y = (in_Y - S_Y) \times in_X \tag{4.2}$$

The output halo (H_X, H_Y), or amount of input data that does not have a corresponding output (See Figure 5.2), is similarly:

$$(H_X, H_Y) = (in_X - S_X, in_Y - S_Y) \tag{4.3}$$

And the total new data required for each iteration of the kernel in the steady state is then ($S_X \times S_Y$).

These properties are illustrated in Figure 4.7 for a 3×3 convolution kernel. The ease with which the reuse can be calculated and analyzed indicates that this parameterization is a good match for one- and two-dimensional sliding window kernels, where maximizing reuse is critical for achieving good efficiency.

By default, Inputs are assumed to have the same degree of parallelism as the kernel to which they belong. This is in general the case for data-parallel kernels:

if a kernel is replicated n times, the Input should be split up so that each kernel's Input receives $\frac{1}{n}$ of the data. However, for some Inputs, such as the coefficient Input to the convolution kernel in Figure 4.5(a), each of the n parallelized kernels should receive replicated versions of the single original Input to ensure they all have the same coefficients. This can be specified by the programmer in the kernel by setting the Input to be a *Replicated Input*, as is shown in the code for the convolution kernel in Figure 4.10. Replicated Inputs are indicated with dashed lines in application graphs.

In addition to defining the size and reuse of data, Inputs and Outputs also serve to represent single buffers in the application implementation. The data movement model assumes that when output data is generated by a kernel it is written into the appropriate Output and held there until the receiving Input(s) are all ready. At that point the data is transfered from the original Output to the receiving Input(s). Similarly, when a kernel reads an Input, the data is received from a buffer within the Input itself. This model provides double-buffering between producers and consumers, which serves to smooth transient bursts of data in the steady-state behavior of the application. In addition, this buffering models the effect of a hardware DMA engine which transfers data into a waiting Input buffer without the kernel's interaction and then allows the kernel to read the data when it is present. This type of DMA engine is present on the Tilera Tile64 processor [46] to make this type of data movement efficient.

4.2.2 Tokens

Kernels communicate within the application by sending *Tokens* between Outputs and Inputs. The majority of the communication consists of *DataTokens* which are two-dimensional arrays of data values. Unlike one-dimensional streaming languages, the native use of two-dimensional streams enables writing applications with two-dimensional inputs without having to contort two-dimensional inputs into one-dimensional streams, as demonstrated in Figure 4.8. This eliminates the need for the

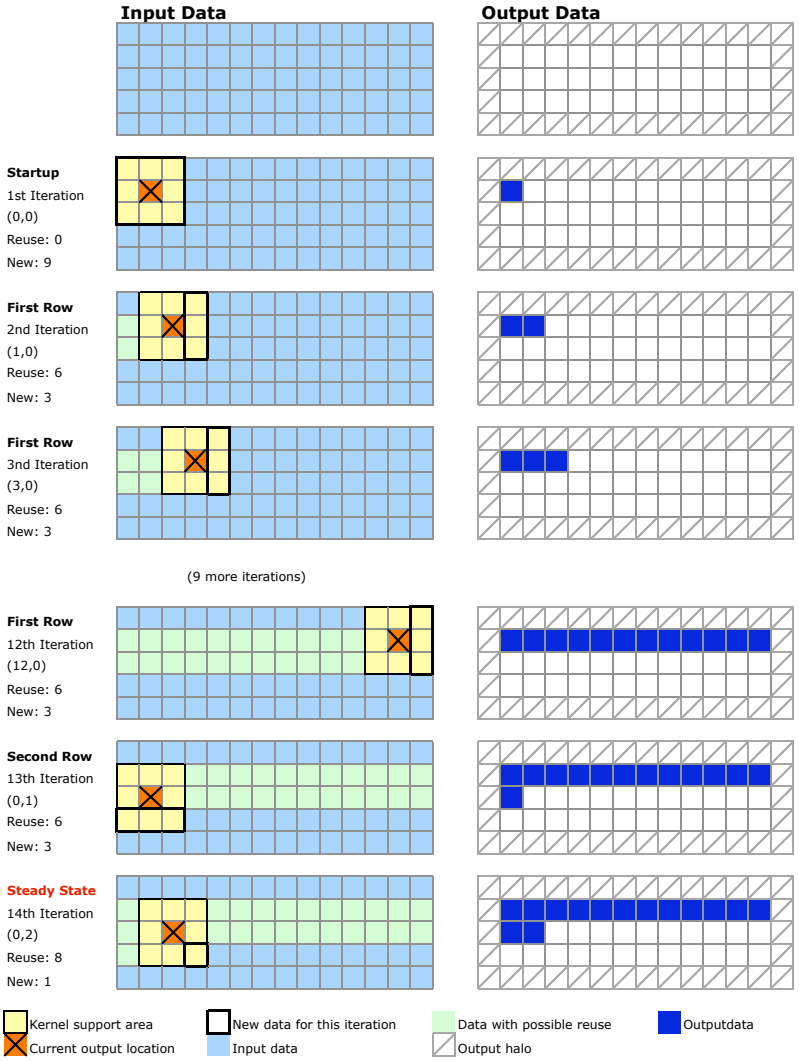


Figure 4.7: Input data usage and reuse for a 3×3 convolution kernel

The kernel input size is (3×3) with a step size of $(1, 1)$. This results in an output halo of (2×2), which can be seen in the right-hand column of the figure. (The 14×5 input results in a 12×3 output, as is discussed further in Section 5.1.) The kernel begins at the upper-left of the image and processes the data left-to-right, top-to-bottom. Data that can be reused is marked behind the kernel in light green and the new data required for each iteration is indicated with a dark border. By the second entry of the second line of processing, the kernel has reached its steady-state reuse pattern of needing only one new input per iteration, as the previous two lines of data can be fully reused as well as the previous two entries on the last line.

```
/*
 * 2D Streaming code for reading a vertical input column.
 */
public void runVDCT() {
    double[][] column = readInput("in");
    ...
}


/*
 * 1D StreamIt code for re-ordering input to form a vertical
 * input column from a size*size block that has been flattened
 * into a 1D stream.
 */
work pop size*size push size*size {
    for (int c = 0; c < size; c++) {
        int x0 = peek(c + size * 0);
        int x1 = peek(c + size * 4) << 8;
        int x2 = peek(c + size * 6);
        int x3 = peek(c + size * 2);
        int x4 = peek(c + size * 1);
        int x5 = peek(c + size * 7);
        int x6 = peek(c + size * 5);
        int x7 = peek(c + size * 3);
        ...
    }
}
```

Figure 4.8: Comparison of native 2D stream access with 1D
The data access from the vertical DCT kernel in the StreamIt MPEG-2 implementation [10] (bottom) is compared with a vertical DCT kernel implemented for the block-parallel programming system presented here (top). Extracting 2D data from the 1D streams in StreamIt requires intimate programmer knowledge of the data layout and explicit calculation of the location of the desired data within the stream. This significantly complicates the application and complicates compiler analysis.

programmer to manually calculate how the two-dimensional inputs map to the one-dimensional stream.[2] Furthermore, native two-dimensional streams allow for more intuitive analysis of the program as the two-dimensional nature of a particular stream does not have to be inferred for optimal analysis as it would in a one-dimensional streaming system.

[2]While two-dimensional streams handle the majority of the cases required for image processing, there are instances where higher dimensional streams would be desired. In these cases the programmer will have to manually calculate indices into the two dimensional stream or add streams for extra dimensions. E.g., RGB data might be stored as either three two-dimensional streams or one stream with a width three times as wide where the RGB values must be manually extracted. Providing complete flexibility in the dimensions of the streams has to date significantly increased the complexity of buffer management [3, 25].

In addition to sending and receiving data, kernels can generate and consume *ControlTokens*. These tokens allow kernels to generate and receive state information. As with DataTokens, ControlTokens have statically determined rates. This property allows the compilation system to account for the communications and computation incurred by ControlTokens in the same manner as it does for DataTokens. (See Chapter 5 for a discussion of propagating data rates.) ControlTokens can be sent along regular data streams (synchronously), in which case they are kept in order with the data. Alternatively, separate Outputs, Inputs and edges can be added to the application graph to enable out-of-band (asynchronous) communications. Indeed, ControlTokens largely provide a convenience to the programmer by allowing kernels to execute different code at different rates, depending on the type of received Token, and exposing this to the compilation system to enable accurate analysis.

There are two intrinsic types of ControlTokens generated in all applications: *End-of-line Tokens* and *End-of-frame Tokens*. These tokens are intrinsic in the sense that they are automatically generated by DataInputs when the input generates the last value for a given line and the last value for a given frame, respectively. Kernels are free to generate their own tokens as well, and may act upon or ignore any incoming tokens. By default, any received tokens that are not handled by a kernel are copied to the output in order with the received data so that kernels downstream can receive them.

4.2.3 Implementation

Kernels define their Inputs and Outputs in the `configureKernel()` method which is called when a kernel is added to an application. (See Figure 4.9.) The definition specifies the width and height ($in_X \times in_Y$) and the step in X and Y (S_X, S_Y) of each Input, and the width and height (out_X, out_Y) of each Output. The kernel also registers the methods that should be called when the inputs are ready, and specifies the Outputs generated by each method. When a kernel is executed, it accesses its inputs by calling `readInputData()` specifying the Input to read. This returns the data stored in the Input buffer and clears the buffer. Once the kernel has calculated its

output data, it writes it to the specified Output by calling `writeOutputData()`. If the Output is full (e.g., the previous output has not been read by its consuming Input(s)), the kernel's execution blocks on this call until the buffer empties. Kernels similarly read and write DataTokens with the `readInputToken()` and `writeOutputToken()` methods. These methods provide runtime checks that the kernel is indeed reading the correct Token and simplify type-checking for the implementation. Both the data and control variants of the read and write methods access the same buffers for the kernel. For simulation, the Input and Output access functions are implemented as function calls with runtime lookups of the specific buffers. For performance, a compiled version would want to inline these calls and replace the runtime lookups with static buffer placements.

4.2.4 Potential Optimizations

Unfortunately, the functional model presented here of moving data between kernels via Output/Input buffers can be inefficient. For kernels placed on separate processors, the cost of this additional buffering and data movement is small compared to actually moving the data between the processors. For kernels that may be time-multiplexed on the same processor, however, copying the data from an Output buffer to an Input buffer in the same memory is inefficient. For such kernels the buffer copy should be replaced with a pointer exchange which allows the Output and Input to share the same buffers efficiently, without changing the semantics of the underlying operation.

Sending ControlTokens throughout the application can also be inefficient as it requires that the Output/Input buffers hold the token instead of double-buffering data. As it is very likely that the control token are much smaller than the data, this would waste storage. This problem, too, is amenable to optimization enabled by the application definition: portions of the application that ignore certain tokens can be set to not propagate them at all, thereby avoiding wasting their local buffer space. Additionally, if a ControlToken's schedule can be statically determined, as would be the case for end-of-frame and end-of-line Tokens, it would possible to replace the Token's movement with local state machines that generate them directly. However,

while ControlTokens have a defined rate, which allows the compiler to schedule appropriate resources to handle them, the specific timing of their generation may not be perfectly regular, and therefore could not be predicted statically.

4.3 Computation Model

Traditional stream programming assumes one block of executable code per kernel. This single *method* is executed each time sufficient new data arrives. Typically such languages must also provide some method to initialize the kernel before it begins execution, as the model does not inherently support a separate initialization method. The kernels defined here are similar in that they execute when their Inputs are ready, however, unlike most other stream programming languages, each kernel may have multiple methods triggered by disjoint sets of Inputs.[3] Each method can in turn generate zero or more Outputs upon execution, although this number must be statically defined. Within a kernel, all methods share the same memory space. This makes initialization very easy as all it requires is that a special `init()` method be executed when the application starts.[4] As multiple methods can be readily defined, the `init()` method fits in cleanly with the overall design, and is not a special case for the language. The flexibility provided by allowing multiple methods enables writing sophisticated kernels very easily, as is demonstrated in Section 4.4.

Within the confines of accessing only memory local to the kernel and the current set of Input and Output buffers, each method can execute arbitrary computation. This allows for full flexibility in implementing control and looping structures within a given method. Unfortunately this flexibility incurs the cost of enabling programmers to write methods that take variable amounts of time to complete, do not terminate, or potentially use unlimited resources. This poses a problem for the compilation system

[3]For this purpose, two instances of the same Input are considered disjoint if the method is registered to trigger on different types of Tokens. E.g., method A may trigger on DataTokens and method B may trigger on ControlTokens from the same Input.

[4]The `init()` method described here is roughly equivalent to the `prework()` method added in StreamIt 1.1[16]. The StreamIt `init()` method is used to build the stream graph and is not executed as part of the application except when the filter is re-initialized. In this model, the kernel's constructor and `configureKernel()` methods do the work of the the StreamIt `init()` method.

which needs to know how long each method takes to execute and how many resources it consumes. Rather than imposing structural limits on the kernel methods to eliminate this problem, the programmer must instead specify the resources consumed by the method when it is registered. A more sophisticated system could attempt infer these values from a simulation or analysis of the program [16].

While methods are logically executed when all of their Inputs are ready, the computation model does not require that scheduling be dynamic. Dynamic execution by constantly checking for ready inputs is straightforward, but potentially inefficient. For methods whose Inputs are derived from static-sized inputs generated at a fixed rate, the result of this dynamic execution is a static schedule. This static schedule can be determined ahead of time and encoded in a simple finite state machine that executes on the processor to which the kernel has been mapped. Such an approach could eliminate the dynamic scheduling burden for applicable kernels in an application.

In addition to executing methods when their Input data is ready, methods can be defined to execute when they receive ControlTokens. This is done by adding an additional parameter to the `registerMethodInput()` or `registerMethodOutput()` call in the `configureKernel()` method, as seen in the `registerMethodInput()` call for the "finishCount" method in Figure 4.12. If an Input does not have a method defined for handling ControlTokens, any ControlTokens received are passed on to the Outputs for the Input's method, once the same token has been received on all other Inputs for that method. All Inputs to a given method must receive the same ControlToken before it can be processed to ensure that the tokens are consumed in order. For example, the subtraction kernel from the differencing program in Figure 4.9 will not pass on any end-of-line tokens until it has received an end-of-line token on both of its Inputs. This is the desired behavior, as it ensures that only one end-of-line token is generated by the kernel for each line it finishes processing. By automatically handling uncaught ControlTokens in this manner, kernels may safely ignore any ControlTokens they wish as they will make their way downstream through the application automatically. Methods may, however, register to receive DataTokens of any type as well. Indeed, the computation model allows multiple methods to register as being triggered by the same Input as long as they are triggered on different

types of Tokens. This enables a kernel to take different action when it receives, for example, an end-of-line token than when it receives data.

4.4 Kernel Examples

The following three kernels demonstrate the use of multiple inputs, multiple methods, and ControlTokens in the definition of kernels. These features combine to make it easier to describe common data and control flow patterns in a stream programming model.

4.4.1 Multiple Inputs

The simplest example of writing a kernel is the subtraction kernel seen in the middle of the differencing program (Figure 4.5(b)). This kernel simply takes in two values and outputs the result of differencing the inputs. The code is shown in Figure 4.9. The kernel code consists of two methods: `configureKernel()` and `subtract()`. The first method is called when the kernel is added to the application. It defines the required Inputs ("in0" and "in1", both with size (1×1) and step size $(1, 1)$) and the single Output ("out", of size (1×1)). When the kernel is added to the application graph, the Inputs and Outputs are created and automatically added to the application graph and connected to the kernel. The kernel then registers the method(s) within the kernel that should be executed, and specifies which Input(s) are needed before the method(s) can fire, and what Output(s) they generate. In this example, the `subtract()` method is registered to execute when the Inputs "in0" and "in1" are ready. The method itself simply reads the two inputs, places the difference of the two inputs into the result, and then writes the result to the Output "out".

4.4.2 Multiple Methods

A more complex example is the convolution kernel from Figures 4.5(a) and 4.5(b). The code, shown in Figure 4.10, contains three methods. The first, `configureKernel()`, configures the kernel by defining the Inputs and Outputs, and registering the kernel's

```
private double [][] result = new double[1][1];

public void configureKernel() {
    /*
     * Create the Inputs and Outputs
     */
    createInput("in0", 1,1,1,1);
    createInput("in1", 1,1,1,1);
    createOutput("out", 1,1);
    /*
     * Register the subtract() method, define its resource
     * usage, and assign its Inputs and Outputs.
     */
    registerMethod("subtract", 0,0,1,1);
    registerMethodInput("subtract", "in0");
    registerMethodInput("subtract", "in1");
    registerMethodOutput("subtract", "out");
}

public void subtract() {
    double [][] in0 = readInputData("in0");
    double [][] in1 = readInputData("in1");
    result[0][0] = in0[0][0] - in1[0][0];
    writeOutputData("out", result);
}
```

Figure 4.9: Code for a simple subtraction kernel

The `configureKernel()` method is called when the kernel is added to the application graph. The `subtract()` method is registered as executing when data is available on the Inputs "in0" and "in1". The `registerMethod()` call specifies the name of the method to register ("subtract") and the resources the method consumes (storage and operations per iteration). The `registerMethodInput()` and `registerMethodOutput()` calls associate methods and Inputs and Outputs. When no additional parameters are provided, these calls configure the method to be invoked when data is received.

methods.[5] The second method, `runConvolve()` is executed when the Input "in" is ready. This method executes the convolution, as can be seen from the double-nested `for()` loops, and then writes the result to the Output "out". The last method `loadCoeff()` is executed when the "coeff" Input receives data. This method takes the input data and assigns the `coeff[][]` array to point to that data. The array is then shared by the `runConvolve()` method when executing the convolution proper. In the `configureKernel()` method the "coeff" input is defined to be a replicated Input. This specifies that when parallelizing the application, the data to this input should be replicated and not split, since the same coefficients should be delivered to

[5]The `width` and `height` values are set by the kernel's constructor when it created.

all parallel instances of this kernel.

4.4.3 ControlTokens

An example of the use of ControlTokens is the counting kernel (Figure 4.12) from a histogram application (Figure 4.11). This kernel builds a histogram by keeping track of into which bin in the `counts[]` array each input falls. For each input, the `count()` method is executed, which finds the correct bin for the received value and increments its count. When the input reaches the end of the frame, the "in" Input receives an end-of-frame token (`EOFToken`) and the `finishCount()` method is executed. `finishCount()` outputs the final counts to the "out" Output and resets the `counts[]` to zero. This flow is illustrated in Figure 4.13 for a parallelized version of the program.

This kernel is an example where one Input triggers multiple methods depending on the type of data received. In the full histogram program (Figure 4.11), the "count" kernel feeds into a "merge" kernel. This allows the application to be parallelized by creating multiple "count" kernels which can count portions of the input in parallel. When the input frame is finished, all of the parallel "count" kernels will receive the end-of-frame token, which will cause them to generate their final count outputs, which are then combined by the "merge" kernel. In effect, the end-of-frame token triggers the reduction of the current counts to the single final count, without the need for a language-specific reduction operator as in Brook [6].

4.5 Discussion

The presented block-parallel programming model has many similarities to previous stream programming models. The following sections compare the Application Model (Section 4.5.1), Data Model, including control, (Section 4.5.2), and Computation Model (Section 4.5.3) to previous work. Differences in scheduling approaches (Section 4.5.4) are also discussed.

```
public void configureKernel() {
    /*
     * Define the Inputs and Outputs, register the method, and assign
     * resources consumed.
     */
    createInput("in", width, height, 1, 1,
            Math.floor((double)width/2), Math.floor((double)height/2));
    createOutput("out", 1,1);
    registerMethod("runConvolve", 0, 3, 10, 10+3*height*width);
    registerMethodInput("runConvolve", "in");
    registerMethodOutput("runConvolve", "out");

    /*
     * Define the Input for coefficient loading, register the
     * method called when the coefficients are present,
     * and mark that input as begin replicated. (I.e., inputs
     * to it should be copied, not parallelized.)
     */
    createInput("coeff", width, height, width, height,
            Math.floor((double)width/2), Math.floor((double)height/2));
    registerMethod("loadCoeff", 0, 3, 10, 10+2*height*width);
    registerMethodInput("loadCoeff", "coeff");

    /*
     * When parallelizing, the coefficient input should be replicated,
     * not distributed.
     */
    getInputByName("coeff").setReplicateInput(true);
}

private double[][] coeff;
private double[][] result = new double[1][1];

public void runConvolve(){
    double[][] in = readInputData("in");
    for (int x=0; x<width; x++)
        for (int y=0; y<height; y++)
            result[0][0] += in[x][y]*coeff[width-x-1][width-y-1];
    writeOutputData("out", result);
}

public void loadCoeff() {
    coeff = readInputData("coeff");
}
```

Figure 4.10: Code for a convolution kernel

The `configureKernel()` method registers two methods for this kernel. The first, `runConvolve()`, executes when input data is present on the "in" Input. The second, `loadCoeff()`, executes when input data is present on the "coeff" Input, and defines the shared `coeff[][]` array that the `runConvolve()` method uses for the actual convolution. The calculations shown for the "in" input above determine the correct offset for the output relative to the input. Offsets are discussed in detail in Section 6.2.

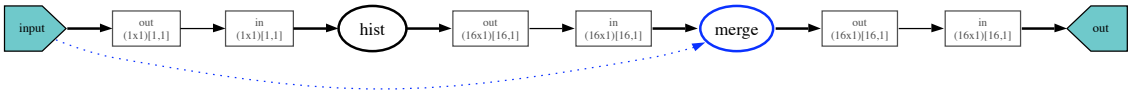


Figure 4.11: Histogram application graph
The blue dotted edge from the Input to the"merge" kernel is a data dependency edge which indicates that there can be no more than one "merge" kernel for each "in" Input when the application is parallelized.

4.5.1 Application Model

Hierarchy in applications

Applications written using the model presented here do not have an explicit hierarchical structure. This is similar to the model used for StreamC/KernelC and SDF-based approaches. StreamIt, however, requires that applications be built by explicitly using hierarchical blocks (pipelines, splitjoins, and feedbackloops, see Figure 3.1) to connect filters [43]. They claim that this enforced hierarchy makes it both easier to describe and analyze applications [16]. However, the majority of the work compiling the applications appears to come from breaking down the hierarchy and re-assumbling it to obtain the right granularity of parallelism for the application and the target architecture [15]. It is not clear that the presence of the hierarchy in the first place actually simplified this manipulation. Indeed, unless the lower levels of the hierarchy are completely independent of the rest of the application, the hierarchy must necessarily be flattened for whole-program analysis. While it is unclear if hierarchy in stream programs simplifies analysis and compilation, it does have a real potential to make code reuse easier by coarsening the level at which the programmers need to interact with the application structure. Such hierarchy, however, can be easily added to any of the traditionally "flat" streaming languages.

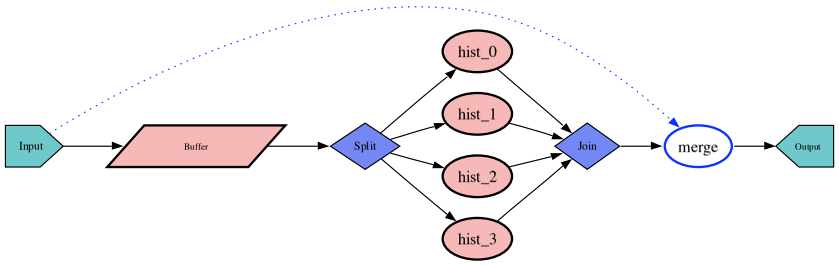
Static rates

The requirement that the data sizes, rates, and computation rates be statically known at compile time is quite similar to most other stream languages. StreamC and KernelC

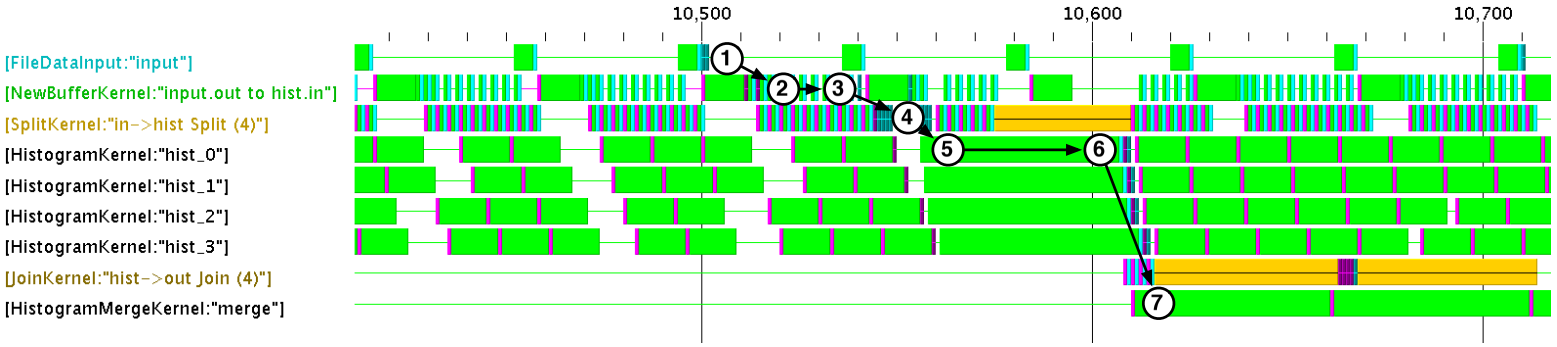
```
private int numberOfBins;
private int[]counts;
private double[][] finalCounts = new double[numberOfBins][1];

public void init() {
    super.init();
    Arrays.fill(counts, 0);
}

public void configureKernel() {
    createInput("in", 1,1,1,1);
    createOutput("out", numberOfBins, 1);

    // init() method clears the bins, so it takes some time and memory.
    registerMethod("init", numberOfBins, 0, 5, numberOfBins*2+3);

    // count() runs when we get new data
    // On average we search half way, so the run time is ~bins/2
    registerMethod("count", 0, 4, 15, numberOfBins/2+5);
    registerMethodInput("count", "in");

    // finishCount() runs when we get an End-of-frame Token.
    registerMethod("finishCount", 0, 4, 6, numberOfBins*3+3);
    registerMethodInput("finishCount", "in", EOFToken.class);
    registerMethodOutput("finishCount", "out");
}

/**
 * Does the counting.
 */
public void count() {
    double[][] input = readInputData("in");
    double value = input[0][0];
    counts[findBin(value)]++;
}

/**
 * Finishes the count by dumping the results and resetting the counts.
 */
public void finishCount() {
    for (int i=0; i<numberOfBins;i++) {
        finalCounts[i][0] = counts[i];
        counts[i] = 0;
    }
    writeOutputData("out", finalCounts);
}
```

Figure 4.12: Code for a histogram kernel

The histogram counts are stored in the `counts[]` array, which is initially cleared by the `init()` method. The `count()` method increments the count for the correct bin. (The `findBin()` method is omitted for clarity.) At the end of a frame of input, the `finishCount()` method is triggered by the End-of-frame ControlToken (`EOFToken`), which causes the kernel to send the final counts to the "out" Output and reset the counts to zero. This is configured by providing the additional parameter to the `registerMethodInput()` call for the "in" input to the "finishCount" method.



(a) Parallelized Histogram Program



(b) Parallelized Histogram End-of-Frame Trace

Figure 4.13: Parallelized Histogram Token Behavior

The histogram program (Figure 4.11) has been parallelized to use four histogram kernels (Figure 4.13(a)). The execution of this program can seen in the simulator output in Figure 4.13(b). The input generates an `EndOfFrame` token (1), which is sent through the buffer (2 and 3), and then through the data splitter (4) and on to the "hist" kernels (5). The histogram kernels then generate their count output (6), which is combined in the data joiner and passed on to the "merge" kernel (7). The detailed behavior of the Split and Join kernels is discussed in Section 7.1.

on Imagine, however, can handle variable length streams relatively easily. They do not suffer from the load balancing problems that most other languages have as their target architecture only executes a single kernel at a time. This guarantees perfect load balancing in the presence of variable stream lengths as long as the data can be prefetched sufficiently. For languages targeting multiple processors, the nature of the various kernels, be they data-, pipeline-, or task-parallel, plays a strong role in the overall load balancing, and hence efficiency, of the application. If these behaviors are not known statically, the compiler can not do a decent optimization.

Inputs and Outputs

StreamIt is the only language of the ones discussed here that limits each kernel (or filter) to a single input and output stream. This limitation makes analysis simpler, but leads to some very contorted code when multiple data streams must be synthesized from manually multiplexed single inputs. In particular, this limitation leads to a proliferation of splitjoin filters with odd patterns that are highly implementation-dependent. For example, the MPEG2 application in [10] has to manually split motion vectors and the macroblocks apart when processing the image. This requires global knowledge of their size and how they are encoded, which is not available to the compilation system for analysis. By allowing multiple stream inputs and outputs, the kernels written in most other languages can separate data in a logical fashion which not only makes the applications easier to write and maintain, but gives the compiler additional information to aid in program analysis.

4.5.2 Data Model

Stream Dimensions

The native two-dimensional stream type discussed here make writing and analyzing image-based applications much easier than the one-dimensional streams of KernelC, StreamIt, and most SDF variants. The more general n-dimensional streams of Array-OL, GMDSDF, and WSDF are potentially more elegant, but have significant implementation difficulties. Their problems stem largely from overly-flexible access methods

to the data and explicitly undefined execution order. The latter issue greatly complicates practical implementations because the search space for execution orderings grows tremendously with each dimension of the data [25]. Even for nominally simple two-dimensional data, both Array-OL and WSDF have shown this to be a difficult problem. The extremely flexible data access patterns (see Figure 3.4) are mathematically elegant, but lead to overly complicated inter-kernel analysis when different input and output patterns meet [4]. Unfortunately, these two issues are closely intertwined: to take advantage of the potential buffer savings and efficiency of being able to accurately describe complex access patterns to the compilation system, the system must be able to find the ordering that works best given the access patterns. Neither Array-OL nor WSDF have demonstrated the ability to do this on a significant scale.[6]

The choice of providing a two-dimensional primitive data type with only limited sliding window access patterns represents a reasonable tradeoff in terms of complexity and utility between one-dimensional streams and generalized n-dimensional stream. While n-dimensional streams are undoubtedly more powerful, it is not clear that the additional complexity incurred to describe, analyze, and manipulate them is justified.

Control

The integration of ControlTokens into the data streams provides a flexible communication method much like StreamIt's teleport messaging [44]. StreamIt, however, provides the ability for the programmer to specify the minimum and maximum latency for the message, although the determination of reasonable bounds is left up to the programmer. This is helpful in the StreamIt model because it is assumed that the application is compiled to a static schedule. A benefit of this level of static analysis is that it allows messages in StreamIt to be automatically synchronized with frame boundaries in different parts of the application. For example, a message to change filter coefficients sent upstream from a filter can be arranged to only arrive between an input window to the filter. The analysis required to implement this feature requires

[6]Generalized kernel merging for Array-OL has been implemented for the majority of cases, but it is a very complex transformation that does not fully address the issue of ordering.

only that the application have static data rates, and as such could be applied to the framework discussed here.

Overall, the StreamIt messaging architecture is both successful because of, and limited by, the fact that it is effectively bolted-on to the rest of the application. In StreamIt, the communication rates and processing times for handling messages are not taken into account when scheduling the application. This makes it easy to implement, and works well when the messages are infrequent and require little processing, but there is no way to enforce this in the language. By treating all Tokens, be they ControlTokens or DataTokens, similarly, the approach presented here allows the message rates and processing times to be explicitly accounted for in the application analysis. For example, this enables the end-of-frame and end-of-line tokens to trigger methods which take non-trivial amounts of time, as the analysis is the same as that of the DataTokens. The downside of this more general approach is that there is no easy way to specify that a message is generated at an unknown rate, as is effectively the case with all messages in StreamIt. This feature is useful, but completely unanalyzable.

4.5.3 Computation Model

The major difference in computation between the approach presented here and previous streaming models is the explicit inclusion of multiple execution methods within each kernel. Combined with multiple inputs and outputs, and the ability to have different methods execute when different token types are received, this provides a powerful approach to re-configuring and manipulating kernels. Brook kernels have the ability to receive scalar configuration variables on initial execution, but they are treated as constants [6]. Kernels written in KernelC have the ability to read and write microcontroller variables during their execution, but this is not analyzable by the compiler [35]. In StreamIt the event handlers for each message are similar to allowing multiple methods, but they are not analyzable by the compiler, and they break the hierarchical construction of the language (hence the “teleport” name).

The use of data dependency edges to define level of concurrency available in an

application provides a much finer level of control than is present in other languages. A data dependency edge allows a program to specify that the parallelism permitted for a given kernel is limited by another kernel or input. In StreamIt, program analysis determines crudely if a filter is parallel or not [15], with no way to specify that there is bounded parallelism. StreamC/KernelC and Brook do not suffer from this problem as they assume a computation target that only executes a single kernel at a time. Array-OL and WSDF assume complete data parallelism and do not address the issue.

4.5.4 Scheduling

The requirement that all parts of the application have statically-determined rates and sizes at compile time results in the applications described here falling into the broad category of Synchronous Data Flow (SDF)[33] applications. For all SDF applications, a static, repeated schedule can be determined by simulating the execution of the application with the known rates[33]. However, the cyclic nature of the two-dimensional inputs, in particular the potentially different behaviors and processing rates during steady-state processing and at the beginnings and ends of frames, complicates the schedule by introducing cycles within it. This category of applications, known as Cyclo-static Synchronous Data Flow (CSDF)[13], is characterized by a finite number of cycles within the overall schedule's cycle, and can be analyzed as such.

A significant amount of work has been done on scheduling CSDF[39] applications. The majority of this work should apply directly to the application model presented here. However, the benefits of static scheduling for a many-core architecture are unclear. Historically, static scheduling was beneficial as it avoided having the runtime system check each and every kernel in the application to see which ones could be run next on the single processor. Not only was this check slow, but it was unlikely to pick an optimal schedule. For a many-core processor, with most processors executing one or a only a few kernels, the scheduling overhead is virtually eliminated. Instead, inefficiencies can arise from having to manage the buffers for communicating data between processors. This overhead can be reduced by implementing the buffers with the help of simple hardware DMA engines, thereby making it possible to run dynamically

scheduled applications highly efficiently. For the particular case of the block-parallel applications presented here, the larger data transfer size compared to single inputs for most SDF applications, further amortizes the overhead of buffer management.

It should be noted that the StreamIt compilers have worked very hard to find optimal static schedules for their applications[23, 15]. This was driven by the desire to generate static schedules for the on-chip network of the Raw processor[45] for data communications, although this was apparently abandoned in later efforts [15]. Later work on StreamIt [48] demonstrated that static scheduling was not essential for good performance. For general SDF applications, there is also a tradeoff between schedules and required buffer sizes [5]. This arises from the build-up of data that can occur between kernels if they are executed in different patterns. However, as long as kernels run on their own processors and can block when their output buffers are full, the input buffers can be minimally sized for correctness. This was not possible with StreamIt on Raw due to the static nature of the communications network. Overall, the motivation to generate static schedules appears to be historical, having been driven first by the high overhead of function calls on early DSPs, and later by the need to optimally multiplex kernels on one or a few processors. Recent work [48] indicates that when distributed across multiple processors with reasonable load balance, the overhead of dynamic kernel invocation is acceptable.

4.6 Conclusions

The application model presented here is succinct enough to easily describe kernels and applications efficiently and yet flexible enough to support complex applications. Data-parallel media applications map easily to the model, while more complicated applications, such as the limited parallelism present in the JPEG and Histogram examples, fit cleanly as well. The kernel/stream model is similar to that of StreamIt, but provides much simpler two-dimensional access semantics without the debilitating complexity of a fully general multi-dimensonal SDF approach. The use of Control-Tokens to send non-data signals between kernels allows programmers to elegantly

encapsulate both in-band (i.e., in order with the data or synchronous) and out-of-band (i.e., asynchronous) control with ease. The default handling of ControlTokens allows kernels to handle only those they need explicitly address and safely ignore any others. While the logical scheduling model is a dynamic data-driven approach, kernels whose input schedule is statically known can be statically scheduled to reduce and distribute this burden, while allowing for dynamic operation where required.

However, the most valuable part of this approach is that the application description is parameterized. The application's data input sizes and rates are defined by the DataInput, and need not be referenced elsewhere in the application, except implicitly through the connections made in the application graph. This gives the programmers the flexibility to change the application processing requirements at any time by merely changing the size and rate of the DataInputs. This flexibility comes at the price of requiring a compilation system that can analyze the size and rate data correctly so as to parallelize the application to meet the requirements. Doing so requires first analyzing the application to determine the requirements for each kernel (Chapter 5), and then using those requirements to introduce sufficient parallelism to meet them without wasting resources (Chapter 7).

Chapter 5

Application Analysis

To determine the required processing and storage for each kernel in the application given the application's input size and rate, a data flow analysis is run that propagates the size and rate from the application's inputs through the application graph. At each kernel in the application graph, the sizes of the kernel's Inputs are used to determine the number of invocations required per input frame for each method. The size of the output from the kernel can then be calculated from the sizes of the Outputs for each method, and this can then be propagated on to the next kernel's Input. At the end of this analysis, the required invocation rate is known for each method in each kernel. The degree of required parallelism can then be readily inferred from the kernel's methods' invocation rates, their resource requirements, and the resources available on the target hardware's processor cores. For the program to be valid, this analysis must return consistent results for all points in the application graph where two or more different inputs come together. If the analysis is not valid, the application is inherently inconsistent and incorrect.

5.1 Frame Sizes, Frame Rates, and Iteration Sizes

To determine the invocation rates for each kernel, a data flow analysis is carried out that calculates the *frame size*, *frame rate*, and *iteration size* at each node in the application graph. Knowing these three values for all the Inputs to each method in

a kernel allows the data flow analysis to propagate them to the method's Outputs. Once the analysis is complete, these values give the invocation rates for each method in each kernel, and the size and rate of the data moving between kernels.

The *frame size* is simply the dimensions of the data at a given point in the application. For a 15×5 two-dimensional image, the frame size is 15×5. For a 128×1 vector the frame size is 128×1, and for a list of 64 motion vector descriptors (where each vector contains 4 words $[x_0, y_0, dx, dy]$), the frame size is 64×4.

The *iteration size* is the number of Input or Output chunks that fit in a given frame size. This value is calculated by determining how the Input's size and step size tile the frame size to which the Input is being applied. Given the frame size $(F_X \times F_Y)$ and the Input's size $(in_X \times in_Y)$ and step size (S_X, S_Y), the iteration size $(I_X \times I_Y)$ is calculated as:

$$I_X = \left\lfloor \frac{F_X - in_X}{S_X} \right\rfloor + 1 \tag{5.1}$$

$$I_Y = \left\lfloor \frac{F_Y - in_Y}{S_Y} \right\rfloor + 1 \tag{5.2}$$

This calculation takes the input frame width and subtracts the initial width required for the input's first iteration (hence the addition of 1 later). The remaining width is then divided by the step size to determine how many steps of the Input fit, and the $floor$ is taken to ensure that the last iteration does not step outside of the input frame. For example, the Input to the 3×3 convolution filter shown in Figure 4.7 is (3×3) with a step size of $(1, 1)$. The frame size for the Input is 14×5, resulting in an iteration size of (12×3):

$$I_X = \left\lfloor \frac{14 - 3}{1} \right\rfloor + 1 = 12 \tag{5.3}$$

$$I_Y = \left\lfloor \frac{5 - 3}{1} \right\rfloor + 1 = 3 \tag{5.4}$$

A 128 point FFT (Input size (128×1), step size $(128, 1)$) on a 128×1 input would have an iteration size of 1×1. For the 64 motion vector descriptors described above, the kernel processing them would take in inputs of size (1×4) and step size $(1, 4)$, resulting in an iteration size of 64×1.

The *frame rate* at each point is simply the number of frame size inputs per second. This value is propagated through the application untouched except when kernels are parallelized, at which point the frame rate for each kernel is scaled by $1/n$, where n is the degree of parallelization of the kernel.

5.2 Data Flow Analysis

The data flow analysis performed on the application graph calculates the frame size, iteration size, and frame rate at each point in the graph by propagating them across each element in the graph. The analysis operates on the graph in a topological order which results in producers (Outputs) being processed and updated before consumers (Inputs).[1] The analysis of an application starts at the DataInputs to the application and proceeds through their Outputs and on into the Inputs of any consumers. From there the analysis propagates through the methods of the consumer kernels via their Inputs and then on to subsequent consumers via the methods' Outputs. The data flow analysis transfer functions are given in Table 5.1 for general kernels.

Table 5.1: Default data analysis transfer functions

	Output → Input	Input → Method	Method → Output
Frame Size	-	-	Method Iteration Size × Output Size
Frame Rate	-	-	-
Iteration Size	Input chunks in Output Frame Size	-	-

A "-" indicates that the values does not change. Transfer functions may be different for other kernels, such as Buffer kernels (see Section 6.1.1) and Split and Join kernels.

The initial values for the frame size, iteration size, and frame rate are set by the DataInputs to the application. When the application is built and the DataInput

[1] This ordering imposes the requirement that there be no cycles in the application graph. To allow feedback, it would be possible to replace this single-pass topological ordering with an analysis that iterated to consistency or to break any cycles before analyzing the graph, as discussed in Section 5.2.1.

is instantiated, its size and rate are defined. By changing this instantiation, the analysis will automatically propagate the new size and rate information throughout the application, thereby making it trivial to adapt the same application to different input rates and sizes. In the simple convolution application shown in Figure 4.6, for example, the DataInput "input" is defined to have an output size of (2×2) at a rate of 1 frame per second (it reads its values and size from the file "inA.txt" for simulation). The DataInput's Output determines its iteration size by calculating how many Output chunks fit into the DataInput's frame size, and its rate is simply that of the DataInput itself. For example, a DataInput for a 30Hz 1920×480 RGB encoded input image that has a (3×1) output, would have an image size of 1920×480, an iteration size of 640×480, and a frame rate of 30Hz. A 4096×1 DataInput with a (128×1) Output would have an iteration size of 32×1.

The frame size, iteration size, and frame rate at the Inputs to a kernel are propagated through the kernel on a per-method basis. For each method in the kernel, the iteration size is calculated for each Input as in Section 5.1. All iteration sizes and rates must match for each Input to the method or the application is inconsistent.[2] Note that because the iteration sizes are related to the frame sizes via the Inputs' size and step size, the frame sizes of the Inputs do not have to match as long as the iteration sizes and rates do. The iteration size and frame rate for the method is then simply that of its Inputs. The method's Outputs' sizes and rates can then be calculated by multiplying the Outputs' sizes by the iteration size of the methods that generate them. The Outputs' rates are the same as their generating methods.

Methods that trigger on ControlTokens need to have frame size, iteration size, and frame rates calculated based on the token's rate. For end-of-frame and end-of-line tokens, these rates are inferred from the Input's iteration size and rate. Kernels that define their own ControlTokens need to determine how to appropriately propagate the analysis data for them. That is, given its input size and rate, the kernel needs to calculate the size and rate for any ControlTokens it may produce, to allow that information to be propagated to the rest of the application.

[2]The number of executions of a method is determined by the iteration size of its Inputs. If two Inputs to the same method have different iteration sizes, the iteration size of the method is undefined, and therefore the application is inconsistent.

Once the analysis data has been calculated for each method in a kernel, it can be used to calculate the degree of parallelism required for that kernel. To do so the total operations per second required are calculated by summing over all the methods the number of operations required per invocation for the method times the iteration size times the frame rate. This determines the total number of operations per second for the kernel.

$$kernel_{\frac{ops}{second}} = \sum_{methods} ops \times (I_X \times I_Y) \times rate \tag{5.5}$$

To determine the required degree of parallelism, the required kernel operations per second are divided by the target hardware's operations per second per processor. This calculation results in a crude estimate of the number of processors that must be devoted to a given kernel to meet the data rate requirement inferred from the initial DataInputs to the application.[3] This requires that the compilation system have an estimate of the number of operations the kernel requires on the target architecture. For this work, this number is specified when the kernel is created. In general, this value could be determined by profiling a compiled version of the kernel on the target hardware or by static analysis of the low-level compiled code. By repeating this process for all the kernels in an application, the total processing requirement for the application can be determined.

For example, a 3×3 convolution that takes 10 operations per invocation running on a 40×40 input image with a frame rate of 100kHz will have an iteration size of 38×38. To meet the 100kHz frame rate requirement, $38 \times 38 \times 10^5 = 1.4 \times 10^8$ invocations per second are required, which is 1.4×10^9 operations per second. If the processor cores are capable of 50MOPS, then 29 of them are required.

[3] A more accurate cycle calculation for the kernel must take into account the time required to transmit and receive the data by the Input and Output buffers.

5.2.1 Feedback

Section 4.1.1 defined the application graph as a directed *acyclic* graph (DAG), which implies that no cycles are allowed in the application definition. This constraint enables easy propagation of information from the application's DataInputs throughout the graph in one pass as discussed above, but prevents programmers from writing applications with feedback. As this is critical to many classes of algorithms, it is important to investigate what would be involved in fully supporting feedback.

The first issue with supporting feedback (or loops in the application graph) is enabling the data flow analysis to correctly propagate information from an Output to the feedback Input it feeds. As loops in the graph make topological traversal ill-defined, a multi-pass traversal is required. Such an algorithm can be implemented by either breaking the feedback loops in the graph for the first pass of analysis or by using a work-list to keep track of nodes that must be revisited due to feedback. In either case, an Input to a kernel that is generated by a feedback path must be ignored until the data analysis has determined the frame size, frame rate, and iteration size for the method that drives that Input. At that point, the calculated driving frame size, rate, and iteration size can be used to calculate the Input's frame size, rate, and iteration size. The analysis must then evaluate if the Input is consistent with what has already been calculated for the kernel, and if not report the inconsistency to the user.

The second issue with supporting feedback is defining the initial output value for the feedback loop for the time before data feeds back through it and it reaches a steady-state operation. The number of iterations of data required can be readily determined by the data analysis by looking at the frame rate and iteration sizes along the feedback path, but the values of the initialization data must be determined by the programmer. In many cases this will just be zero, but depending on the algorithm it might need to be different. As the feedback loop will effectively empty at the beginning of each frame, this initialization data will be generated at the start of each new frame. To generate such data, a feedback initialization kernel could be inserted that would output the initialization data at the start of each frame and then pass on its input thereafter.

5.3 Example

The data analysis for the first half of the differencing program shown in Figure 4.5(b) is shown in Figure 5.1. The DataInputs for loading the coefficients into the two convolution kernels have been removed for clarity. Each node has been annotated with the frame size (Image:), the iteration size (It:), and the rate (@x/s). Kernels have the iteration size calculated for each method as "methodName:It:". In addition, Inputs and Outputs are annotated with their inset, which is discussed in Chapter 6.

The data flow analysis starts at the DataInput "input" on the left side of the graph. This DataInput is of frame size 40×40 at a rate of 62.5Hz. The Output for this DataInput is of size (1×1), which means its frame size and image size are also 40×40. The frame rate stays the same. The Input to the "conv3x3" kernel takes its frame size and rate from its source Output. It has size (3×3) and step size $(1, 1)$, so the resulting iteration size is (38×38). This is simply the calculation from Section 5.1, which is:

$$I_X = I_Y = \left\lfloor \frac{40 - 3}{1} \right\rfloor + 1 = 38 \tag{5.6}$$

The "conv3x3" kernel has one method of interest (the `loadCoeff()` and `init()` methods are ignored for simplicity), which is `runConvolve()`. The `runConvolve()` method has one Input, so there is no question of consistency, and therefore that method's frame size, iteration size, and rate are the same as the Input's. The kernel's Output is size (1×1), and by multiplying that by the producing method's iteration size of (38×38), the Output's frame size is 38×38 and the rate is the same as the method. The same analysis follows for the "conv5x5" kernel, except its iteration size is (36×36) as the Input size of (5×5) results in a 4×4 output halo instead of the 2×2 output halo for the 3×3 convolution.

This application becomes interesting when the outputs from the two convolution kernels join up at the "subtract" kernel's single method. Processing the Output from the "conv3x3" kernel through Input "in0" on the "subtract" kernel results in an iteration size of (38×38), while processing the Output from the "conv5x5" kernel through Input "in1" results in an iteration size of (36×36). As the inputs to a given method must have the same iteration size for the computation to make sense, this

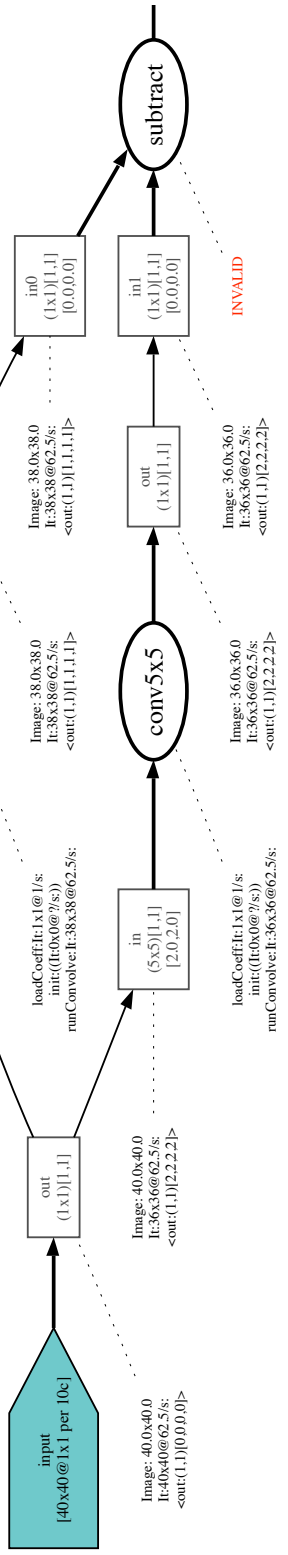


Figure 5.1: Data flow analysis for the first half of the difference program

Each node in this application graph has been annotated with the result of running the data flow analysis discussed here. In addition to the *frame size*, *iteration size*, and *frame rate*, this figure contains insets in the analysis annotations and offsets in the Input nodes, that will be discussed in Chapter 6.

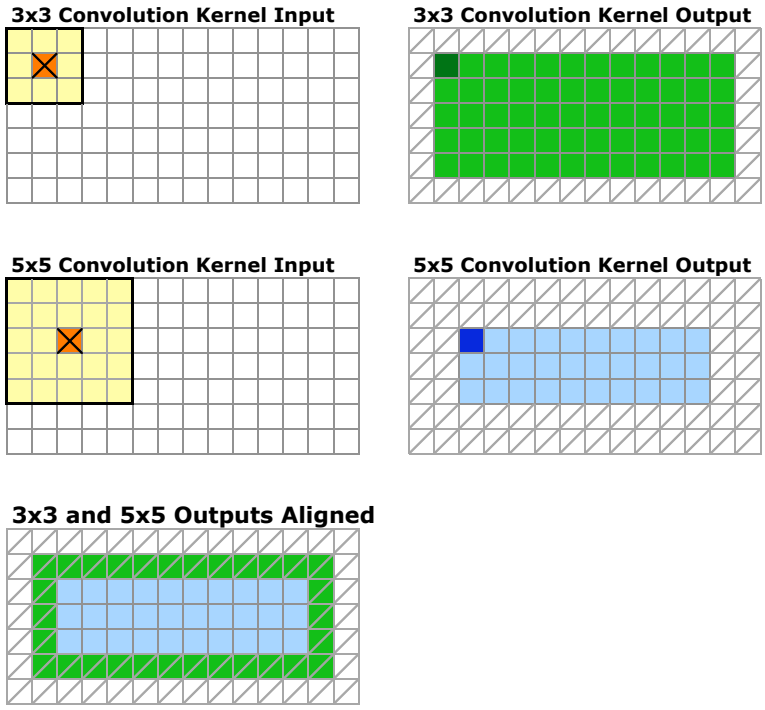


Figure 5.2: Halo differences between 5x5 and 3x3 kernels
The outputs from two kernels with (3×3) and (5×5) are shown aligned to their inputs. The bottom figure shows the two outputs superimposed. If both outputs are needed at any location (e.g., as in the "subtract" kernel in Figure 4.5(b)), the 1 pixel border of calculated outputs for the (3×3) kernel around those of the (5×5) outputs must be correctly handled.

indicates that the application is inconsistent, and hence invalid, as indicated on the graph.

The reason for this inconsistency is that the same application input has been processed through two kernels ("conv3x3" and "conv5x5") that have different output halos. The result is that while the upper-left-most output from the "conv3x3" kernel corresponds to the $(1, 1)$ location in the original input, the upper-left-most output from the "conv5x5" kernel corresponds to the $(2, 2)$ location in the same input. (See Figure 5.2.) For the "subtract" kernel to make sense, it needs to subtract values that are in the same position relative to the input from which they were generated. Equivalently, the "conv3x3" kernel produces an output that is 38×38 (its frame size)

while the "conv5x5" kernel produces an image that is $(2, 2)$ smaller, or only 36×36. If these two outputs are aligned relative to the original input from which they were calculated ("input" in the application), the output from the "conv5x5" kernel would be inset from the "conv3x3" kernel's output by 1 pixel on each side. Since there is no data from the "conv5x5" kernel for this 1 pixel inset, it does not make sense to "subtract" the different size outputs of the two kernels. This effect can be seen graphically in Figure 5.2.

The above example actually reveals two inconsistencies in the application. The first is that the shared input that is used to generate the output from the two convolve kernels has inconsistent sizes when the outputs are used together at the subtraction kernel. The solution to this is to throw out the extra outputs from the "conv3x3" kernel or to zero-pad the input to the 5×5 kernel. The second inconsistency is that the output from the application's DataInput is generated in (1×1) chunks, but it is consumed by the "conv3x3" and "conv5x5" kernels in (3×3) and (5×5) chunks, respectively. With the input arriving in left-to-right, top-to-bottom order, the first (3×3) chunk of data for the first iteration of the "conv3x3" kernel will not be ready until the third value on the third line. In order to use the previous values as well, they must be buffered. Chapter 6 discusses inserting InsetKernels and BufferKernels to correct both of these inconsistencies.

5.4 Discussion

The application analysis presented here is reminiscent of the analysis used in SDF applications to determine that the token production and consumption rates are matched in steady-state flow. These are determined by examining the topology matrix Γ (see Section 2.1) to derive a set of balance equations whose self-consistency indicates that production and consumption rates are valid. This analysis is quite simple for one-dimensional SDF applications, and can be generalized to WSDF and MDSDF multi-dimensional cases. The multi-dimensional analysis could be applied to the application description presented here, but as it does not assume an execution order, it would not be of assistance in determining how to solve the aforementioned inconsistencies.

The type of application analysis presented here is useful for fixing correctable errors in the application description (see Chapter 6) and for alerting the programmer to uncorrectable application inconsistencies. Languages that provide less description of the data movement to the compiler hamper such analysis and increase the complexity for the programmer. For example, the single input/output design of the StreamIt language, forces programmers to manually multiplex and demultiplex data in single streams, which is error-prone and difficult to read and maintain. This results in program structures where global constants are used throughout the program to manually multiplex multiple data streams, such as the Y, Cb, and Cr image data and their associated motion vectors in the MPEG-2 decoder example [10]. Similar difficulties arise with languages that only allow one-dimensional streams, such as StreamC/KernelC. These languages force users to manually map their two-dimensional data to one-dimensional streams which reduces the compiler's ability to analyze them.

The language presented here provides the programmer with the ability to more accurately describe how data is used to the compiler. The inclusion of multiple two-dimensional inputs and outputs per method and multiple methods per kernel, eliminates the need for most manual multiplexing and indexing of streams, while providing the compiler with a concise, parameterized description of how the program operates on its data. Not only does this simplify and increase the ability of the compiler to analyze the program, but it also makes it easier and cleaner to write the program. This results in a compilation system that is more helpful in detecting and correcting programmer errors, and more readily able to analyze and manipulate the flow of data for static programs.

Chapter 6

Buffers and Insets

As demonstrated at the end of Chapter 5, an application that appears well-defined can display inconsistencies when analyzed. The differencing application (Figure 4.5(b)) is a clear example of this problem. The application is specified correctly, but when analyzed two problems are revealed. The first is that the DataInput generates data in (1×1) chunks, a row at a time, while the convolution kernels consume the data in (3×3) or (5×5) chunks. This requires that a buffer be inserted between the DataInput and the convolution kernels to buffer a sufficient number of rows of input to generate the required data chunks. The second inconsistency arises from trying to subtract the output of one convolution kernel from the other. Because the convolutions are of different sizes (3×3 and 5×5), the outputs are of different sizes (38×38 and 36×36) despite having the same ancestor inputs. (See Figure 5.2.) This causes the two inputs to the subtract kernel to have different iteration sizes for the same method, which is inconsistent.

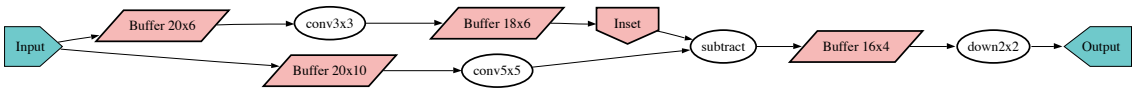


Figure 6.1: Automatically buffered and corrected differencing program
The differencing program from Figure 4.5(b) has been analyzed and appropriate InsetKernels (Section 6.2) and BufferKernels (Section 6.1) have been automatically inserted to correct the inconsistencies. This program now consistent.

These inconsistencies result in an invalid application description, that prevents analysis, manipulation, and execution. However, the original program definition (Figure 4.5(b)) is as expected from the programmer. To enable the programming system to accept program descriptions with these inconsistencies, it is necessary to automatically analyze and correct them, the results of which are shown in Figure 6.1.

6.1 Buffers

Inserting a buffer between two kernels is necessary whenever the Input size and step do not match that of the source Output. In these cases, a buffer needs to be inserted to store enough of the source data to allow the sink to read in the data in the size specified by its Input. Besides being necessary for correctness, buffers increase the overall application buffering beyond the single buffers in each Input and Output, and provide an opportunity to take advantage of the data reuse present in many kernels (see Section 4.2.1).

Buffers provide the abstraction of two-dimensional circular memory structures that write in new data as long as they can without overwriting old data that has not yet been read out. (See Figure 6.2.) This results in an access pattern whereby the buffers are written line-by-line, filling up horizontally to the end of a line, and then wrapping around at the bottom to overwrite the oldest data at the top. Buffers may be large enough to need to map to larger, higher-level memories in the target architecture, and may also require higher-level blocking to fit. These issues are not addressed here, but are discussed in Appendix C.

6.1.1 Buffer Sizing

The size of a buffer is determined by the image size of the source writing into the buffer and the Input size needed by sink kernel reading out of it. For the DataInput's row-by-row ordering, the buffer generally needs to be as wide as the source image.[1] At a minimum, the buffer needs to be tall enough to hold one row of the larger of

[1] Alternatively, if the application were defined such that the DataInputs generated data in a column-by-column order, the buffer would generally need to be as tall as the image.

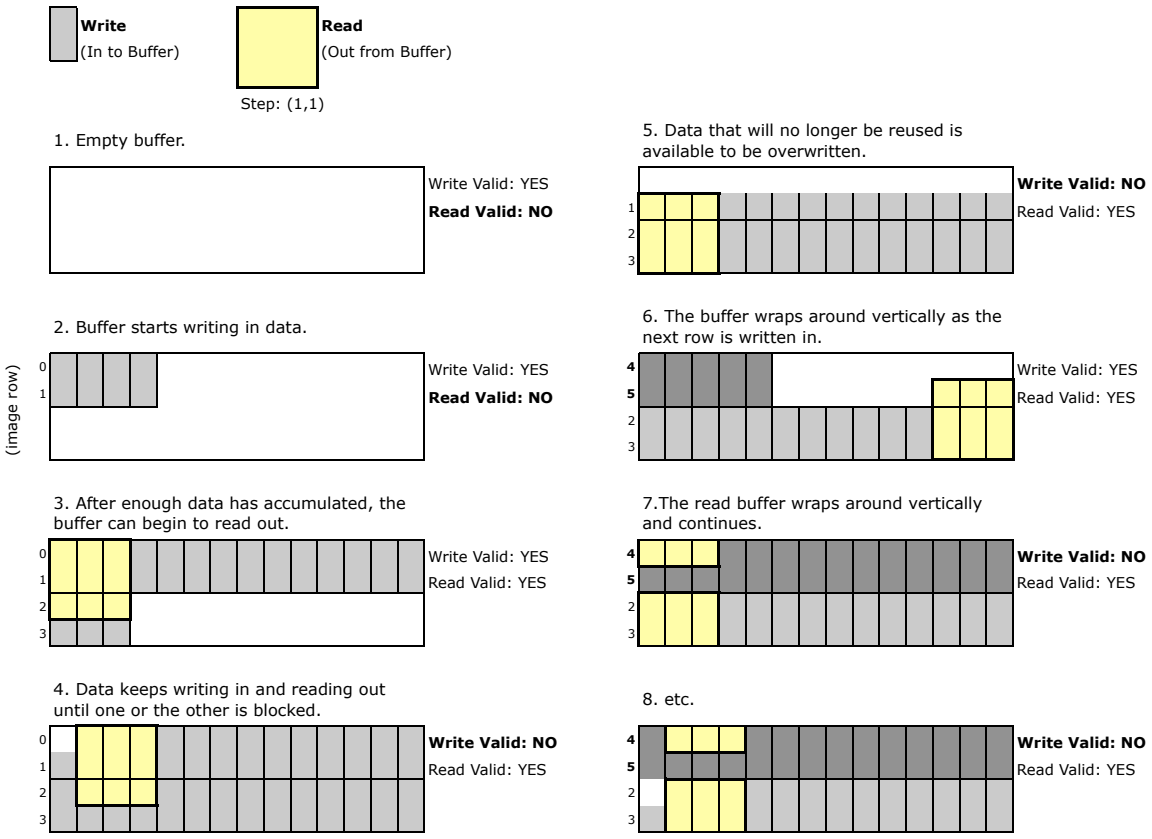


Figure 6.2: Two-dimensional circular buffer operation

its output or input. However, if the input size is not an even multiple of the output, more rows may be required to ensure that the buffer can store enough rows of the input to generate the outputs.[2] For example, for a source Output size of 3×3 feeding a sink Input size of 1×2, the buffer needs to be a minimum of 4 rows high to hold two rows of the input, which is the minimum amount of input to generate one row of output. If the Input was 1×1, the minimum height would be simply that of the

[2]If the Input and Output have the same height and step size, then the buffer can be simply that tall as there is no need to buffer more than one row. In this case a horizontal-only buffer may be used which is circular only in the horizontal direction and can further reduce buffer requirements.

Output, or 3. (See Figure 6.3.) Thus, the minimum buffer height is:

$$\max\left(out_Y, \left\lceil \frac{out_Y}{in_Y} \right\rceil \times in_Y\right) \tag{6.1}$$

While the above specifies the minimum size of the buffers, they should generally be large enough to double buffer the data, thereby allowing the next line of data to be written in while the current one is read out. The calculation to determine the height for double buffering is similar to the above, but the buffer should be tall enough to hold enough rows of input to cover two rows of output (taking into account the step size of the output) or two rows of input. The maximum of two rows of input or two rows of output is required to ensure that there is space to allow the next set of data to be written or enough to hold two sets of output data. If we define the minimum double buffered output height (one full row plus a the step size of the next row) to be:

$$\alpha = out_Y + S_Y \tag{6.2}$$

the minimum double buffered height is then:

$$\max\left(\alpha, \left\lceil \frac{\alpha}{in_Y} \right\rceil \times in_Y, 2 \times in_Y\right) \tag{6.3}$$

This ensures that the buffer is tall enough to hold the smallest of double-buffering the output, the input, or the number of input rows to make up the double-buffered output.

Unfortunately this double buffer sizing ignores the effects of buffering between input frames. (See Figure 6.4). When one input frame is ending, the buffer needs to be sufficiently large to hold enough input from the next frame to build up the first full row of output for the next frame. This is necessary to ensure that the buffer does not become a bottleneck by forcing any downstream kernels to wait at the end of one frame for the next frame to fill in. In addition, when a buffer is placed after a DataInput, it must ensure that it can continuously take in data at the DataInput's rate, or the application will drop data and producer incorrect results. Elsewhere in the application there is likely to be sufficient buffering in Inputs, Outputs, and other

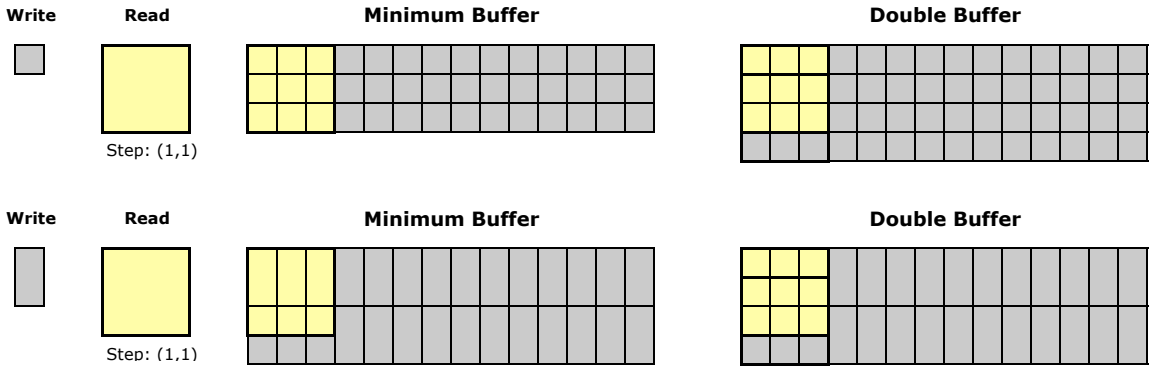


Figure 6.3: Buffer sizes for double-buffering

buffers to relax this requirement to only demand that the buffer accept inputs at the average rate of its source. The inflexible buffers that come after DataInputs therefore require sufficient space to ensure that they will always be able to accept data at their source DataInput's rate. The correct buffering for this case is enough to hold one row of output and enough rows of the input size to make up another full input row, or to double buffer the input. This gives the required height for buffers as:[3]

$$out_Y + \max\left(\left\lceil \frac{out_Y}{in_Y} \right\rceil \times in_X, in_X \times 2\right) \tag{6.4}$$

For the more flexible buffers elsewhere in the application, less buffering may be required.

Buffer Data Flow Analysis

The data flow analysis for a buffer is quite similar to that of a regular kernel. The primary difference is that the output size is not directly the method's iteration size times the output size due to reuse in the output. Instead, the buffer calculates the number of output iterations based on the original Output and Input between which the buffer has been placed. The frame rate remains the same.

[3]The buffering present in each Input and Output adds 4 more buffers (Output to buffer Input, buffer Input, buffer Output, and Input from buffer Output) to the total buffering as well. For a 1×1 Input to a 5×5 convolution kernel, this reduces the required buffering from the calculated 10 lines to 8, and from 11 lines to 10 for a 1×2 Input.

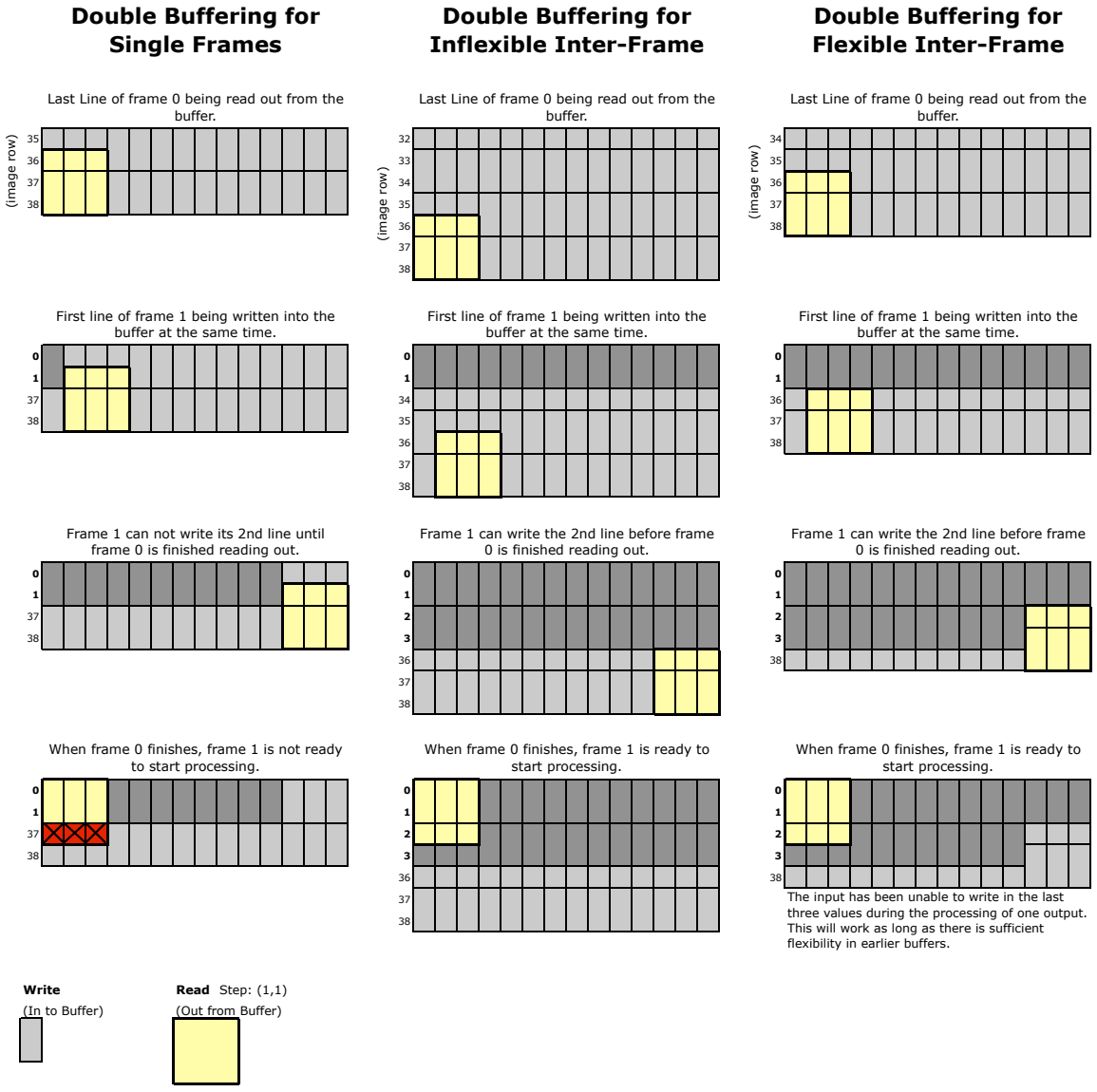


Figure 6.4: Buffer sizes for double-buffering between frames

To ensure that computation can continue uninterrupted between frames, the buffer needs to be large enough to hold the last row of output from the first frame and enough of the next frame to capture its first row of output. This ensures that the buffer will not cause a bottleneck that prevents the first output from the next frame from processing immediately after the last output from the first frame is processed. This constraint can be relaxed for more flexible buffers that are not fed directly from the application's DataInputs.

6.1.2 Implementation

Buffers are implemented as *BufferKernels*. The BufferKernel runs continuously (as opposed to being triggered when the input is ready) so they can simulate constantly reading and writing. The run loop tries to 1) write in new data if new data is available and space is available in the buffer, 2) read out data if there is data available in the buffer and the output is ready, 3) consume any tokens if they are available on the input, and 4) generate any required tokens on the output. The run loop is shown in Figure 6.5. For the simulator, the buffers are implemented in a brute-force manner whereby the current frame for each element in the buffer is stored in a separate array, thereby making it easy to check if an appropriate chunk of data or free space is available for reading or writing. This makes it straight-forward to read out the end of one frame while storing in the beginning of the next frame.

The difficulty of implementing the buffers comes from the different sizes and step sizes of the input and output, and the desire to have a double-buffered input that uses the available storage efficiently between lines and frames. By relaxing any of these constraints the buffer could be implemented much more simply. Alternatively, hardware support in the form of an appropriate DMA engine could be used to simplify the implementation.

6.2 Insets

When an input to an application is processed through two different paths in the application graph, the outputs of the two paths will differ in their iteration sizes if the kernels along the two paths generate different output halos. This was demonstrated in the differencing program in Figure 5.1. If the results of these two paths are then used together as inputs to the same method, an application inconsistency occurs as all inputs to a given method must have the same iteration size. The cause of this inconsistency is that the method using both inputs (the "subtract" kernel in the case of the differencing program) sees two input images of different sizes because the kernels produce outputs of different sizes. (See Figure 5.2.)

```
public void runBuffer() {
    while (isRunning()) {
        /*
         * As long as we are idle waiting for input just wait.
         */
        while (isRunning() && !validToReadBuffer() &&
               !readyToReadData() && !readyToReadToken()) {
            yield();
        }

        /*
         * Try to write into the buffer.
         */
        boolean write = tryToWriteBuffer();

        /*
         * Try to read a token from the input.
         */
        boolean token = tryToReadToken();

        /*
         * Try to read out from the buffer.
         * If this needs to generate an output token it will do so and try to
         *  read in more data if it is blocked in sending the output token.
         */
        boolean read = tryToReadBuffer();

        /*
         * If we didn't do anything then yield, otherwise go around the loop again.
         */
        if (!(write | read | token))
            yield();
    }
}
```

Figure 6.5: Run loop for the BufferKernel
Note that this kernel is not Input-triggered, and therefore must explicitly yield to other kernels if it is to be run in a time-multiplexed manner.

However, the logic of the program ("subtract these two intermediate results") is reasonable. Indeed the desired outcome is to realize that these two differently-sized outputs are related such that throwing out the outer pixel border on the "conv3x3" kernel's output, or zero-padding the input to the "conv5x5" kernel, will produce the correct (and desired) result. This fix is achieved by inserting an InsetKernel after the "conv3x3" kernel that removes that border, or inserting a ZeroPadKernel before the "conv5x5" kernel that enlarges the input sufficiently to produce valid outputs on the border. Determining that this is the correct application modification requires an additional data flow analysis that tracks where each application input is used and how its coordinates and halo have been transformed by the kernels through which it has passed.

6.2.1 Data Flow Analysis for Insets

The data flow analysis for calculating insets is similar to the analysis in Chapter 5: information regarding each DataInput is propagated through the application graph in topological order, and updated at each Input and Output. However, unlike the frame size, iteration size, and frame rate, the inset applies only to Inputs and Outputs. As the data flow analysis is performed, the insets are updated from source Output to sink Input, and then from method Input to method Output. For an application to be consistent, the Inputs to a method that share a common *ancestor Output* must have matching insets. If they do not, an InsetKernel is inserted after the kernel or a ZeroPadKernel is inserted before the kernel to correct the Input's inset[4]. In the differencing program example the common ancestor Output is the "out" Output from the "input" DataInput, which flows through both convolution kernels to the "subtract" kernel.

Insets are represented as the number of pixels the data is inset from the ancestor Output in each direction and scale factors for the x- and y-directions:

$$< ancestorOutput : (scale_x, scale_y)[left, top, right, bottom] > \tag{6.5}$$

The scale factor accounts for kernels that change the relative size of an image, such as downsampling. As an example, the inset for the Outputs from the "conv3x3" and "conv5x5" kernels in the difference program (Figure 5.1) are $< out : (1,1)[1,1,1,1] >$ and $< out : (1,1)[2,2,2,2] >$, respectively. These correctly reflect the 1- and 2-pixel wide halos relative to the ancestor Output "out" from the ancestor DataInput "input" shown in Figure 5.2.

The inset data flow analysis begins by creating initial insets of $< (1,1)[0,0,0,0] >$ at the Outputs for all DataInputs to the application. From there they are propagated through the application in a topological order. There are two transformations that are applied to the insets. The first is from a source Output to a sink Input. This

[4]The decision as to whether to zero-pad or inset the data must be made by the programmer due to its effect on the final output. The implementation presented here automatically inserts InsetKernels as needed, but does not implement zero-padding.

transformation calculates the base inset at the sink Input, divides it by the inset scale at the source Output, and then adds it to the inset from the source Output.

Inset calculation for the differencing program

From the difference program, the kernel's sink Input to the "conv5x5" kernel has a size of (5×5) with a step size of $(1, 1)$, for a halo of $(5 - 1 \times 5 - 1) = (4 \times 4)$. To calculate the base inset for this Input, an *offset* (O_X, O_Y) is needed to indicate how the (4×4) halo $(H_X \times H_Y)$ is distributed. The base inset at the Input is then calculated as:

$$inset_{base} = [O_X, O_Y, H_X - O_X, H_Y - O_Y] \tag{6.6}$$

The offset can be though of as specifying how the method's Output is offset from an Input. Figure 6.6 shows the effect of three different offsets for the Input to the "conv5x5" kernel. Offsets are defined for Inputs when they are created and are denoted as $[x.x, y.y]$ in the application graph.[5] For the difference program's "conv5x5" kernel with an offset of $(2, 2)$, the base inset at the Input is then:

$$inset_{base} = [2, 2, 4 - 2, 4 - 2] = [2, 2, 2, 2] \tag{6.7}$$

This is scaled by the source Output's scale ($(1, 1)$ in this case) and added to the source Output's inset of $[0, 0, 0, 0]$, resulting in the inset of $< out : (1, 1)[2, 2, 2, 2] >$.

$$[2, 2, 2, 2] = [0, 0, 0, 0] + [\frac{2}{1}, \frac{2}{1}, \frac{2}{1}, \frac{2}{1}] \tag{6.8}$$

The final Output $\rightarrow$ Input transformation is then:

$$input_{inset} = sourceOutput_{inset} + \frac{inset_{base}}{sourceOutput_{scale}} \tag{6.9}$$

The second transformation that is applied during the inset analysis is from the source Input to a method to its sink Output. As the step size and frame size are the

[5] Offsets have been omitted from most examples before this point for simplicity. The need for fractional offsets will be discussed later.

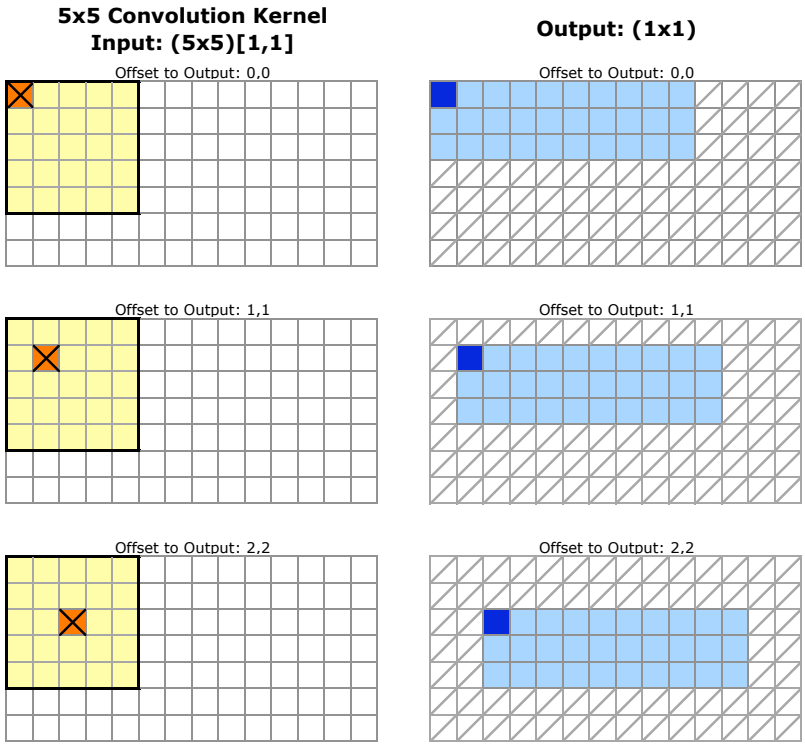


Figure 6.6: Offset example for a 5×5 convolution kernel

same for an Output, its halo is necessarily (0×0). Therefore the only part of the inset that changes is the scale, which is determined by dividing the step size of the Output by the step size of the Input.

In the case of the difference application, this analysis concludes that the two inputs to the "subtract" kernel have insets of $< out : (1,1)[1,1,1,1] >$ and $< out : (1,1)[2,2,2,2] >$ and iteration sizes of (38×38) and (36×36). (See Figure 5.1.) The inset data analysis makes it clear that to fix this inconsistency, an InsetKernel needs to be inserted along the "conv3x3" path to inset the output by an additional $[1,1,1,1]$ to bring them both to $[2,2,2,2]$. Alternatively, a ZeroPadKernel needs to be inserted before the "conv5x5" kernel to bring them both to $[1,1,1,1]$ by enlarging the input image.

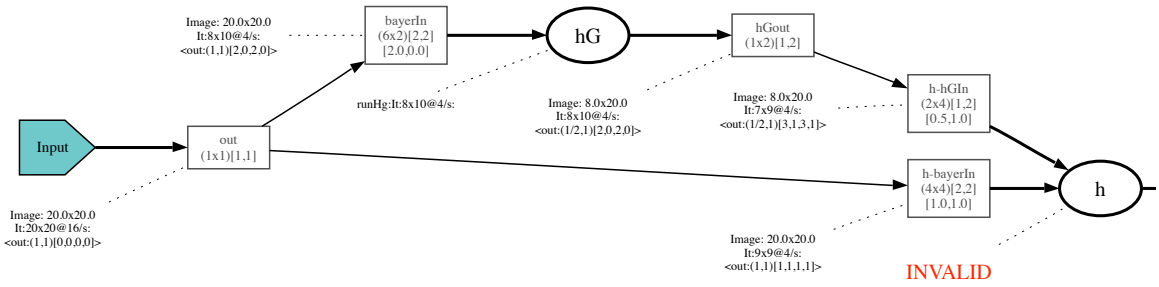


Figure 6.7: Bayer demosaicing program inconsistency
The $[2.0, 0.0]$, $[0.5, 1.0]$, and $[1.0, 1.0]$ values in each Input are the offsets from that Input to the kernel's Output.

Inset calculation for the Bayer demosaicing program

A more complicated example from a Bayer demosaicing application is shown in Figure 6.7. This application demonstrates a similar problem with insets as the differencing program whereby the two inputs to the "h" kernel have different iteration sizes and insets, thereby rendering the application invalid. In addition, this example demonstrates how scaling comes into play with insets and how fractional offsets may be required to properly define an Input. To analyze this program, the data flow analysis first proceeds along the top half of the graph ("out" $\rightarrow$ "bayerIn" $\rightarrow$ "hG" $\rightarrow$ "hGout" $\rightarrow$ "h-hGIn" $\rightarrow$ "h") and then the bottom half ("out" $\rightarrow$ "h-bayerIn" $\rightarrow$ "h").

The inset calculations for the top half of the application, through the "hG" kernel, are shown in Figure 6.8. The calculation starts at the Input "bayerIn" to the "hG" kernel. Here the inset from the source is $< out : (1, 1)[0, 0, 0, 0] >$ and the base inset for the Input is $< (1, 1)[2, 0, 2, 0] >$ because the Input has a halo of (4×0) and an offset of $(2.0, 0.0)$, which gives $inset_{base} = [2, 0, 4 - 2, 0 - 0] = [2, 0, 2, 0]$. The final inset for the Input is then $< out : (1, 1)[2, 0, 2, 0] >$, which is then propagated through the "hGout" Output. As this is a propagation from and Input $\rightarrow$ Output, only the scale changes. Here the scale is the Output step size divided by the Input step size, or $(1 \div 2, 1 \div 1) = (0.5, 1.0)$. This scale is then used at the next point when adding the base

inset from the "h-hGIn" Output to the inset at the "hGout". This changes the "h-hGIn's" base inset from $< (1,1)[0.5,1,0.5,1] >$ to $< (0.5,1)[1,1,1,1] >$, which is then added to the inset coming into the Input for the final result of $< (0.5,1)[3,1,3,1] >$. The iteration size, which is closely related to the inset, is (7×9).

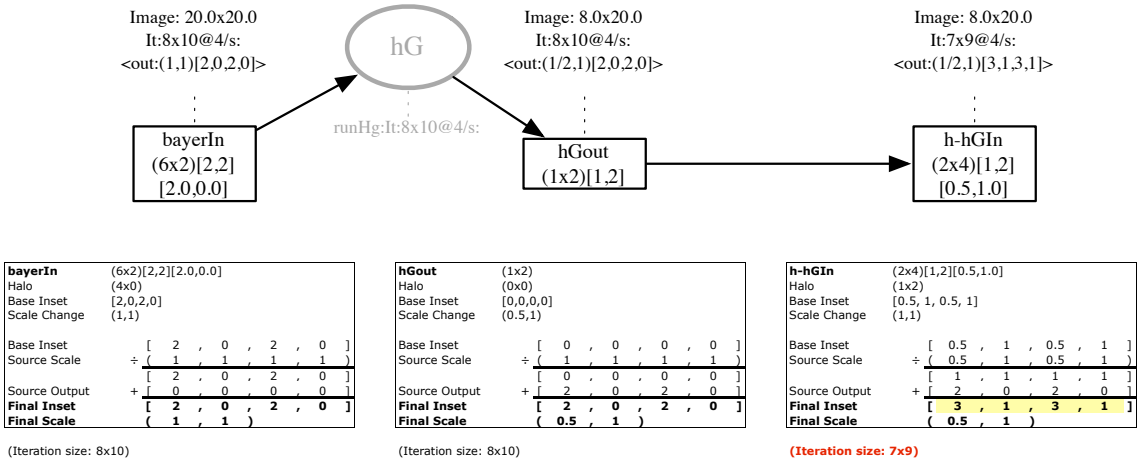


Figure 6.8: Bayer "hG" kernel inset calculations
The data flow calculation proceeds from left-to-right, with the result of the first calculation feeding into the next one as the "Source Output". The calculations follow the transformations discussed in the text.

Note that the "h" kernel's "h-hGIn" input has an offset of $[0.5, 1.0]$. This fractional offset is necessary because the kernel before it, "hG", effectively downsampled its output relative to the input. This can be seen by observing that the output step size for the "hG" kernel is $(1, 2)$ while its input step size is $(2, 2)$. The 0.5 offset ends up being divided by the 0.5 scale to produce the correct result. Graphically this downsampling is shown in Figure 6.9, where the output is shown to be half as wide as the input. The fractional offset here is required to make the application consistent, and must, unfortunately, be manually entered by the programmer. The insets can not be automatically calculated as they are determined by how the generated data is related to the input data, which is a function of how the kernel has been implemented.

The lower path in Figure 6.7 is through the "h-bayerIn" Output. This path's calculations are shown in Figure 6.10. The calculation here is much simpler, and

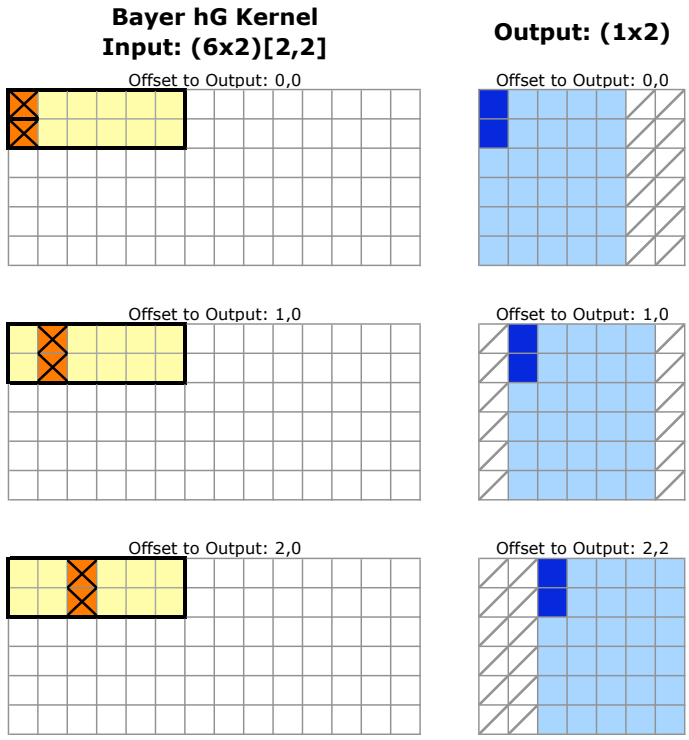


Figure 6.9: Offset example for the Bayer hG kernel
Note that the Output has a scale factor of $(0.5, 1)$ relative to the Input, which leads to the output frame size being half that of the Input.

reveals that the the inset after the "g-bayerIn" kernel is $< out : (1,1)[1,1,1,1] >$ and the iteration size is (9×9). As can be seen from the analysis, this results in two inconsistent Inputs at the "h" kernel. However, while it is much less clear why this inconsistency occurs than in the simpler differencing example, the solution is similarly clear from the analysis as it was in the differencing program: insert an InsetKernel with inset $[2,0,2,0]$ to bring both insets to $[3,1,3,1]$. The calculation after adding such an InsetKernel is shown in Figure 6.11.

Note that scaling also comes into play when determining if two Inputs to a method are consistent. Adding a $[2,0,2,0]$ inset to the bottom path in the aforementioned application will result in the top path having an inset of $< out : (0.5,1)[3,1,3,1] >$ while the bottom one has a scale of $< out : (1,1)[3,1,3,1] >$. (See Figure 6.14(a).)

This requires that the analysis to determine if the Inputs are consistent scale the Input's step size by the inset scale when comparing them. In this example, the upper path's setph size of 1 is scaled by (divided by) $\frac{1}{2}$ which matches the lower path's step size of 2 scaled by 1.

6.2.2 Zero Padding

A standard method for dealing with differing insets between two kernels' Outputs is to zero-pad the input to the smaller of the two rather than throw out the data from the larger. This approach keeps the frame size the same by adding dummy data as needed, as shown in Figure 6.12. The data analysis required to zero pad is the same as that required to insert InsetKernels. The difference is that ZeroPadding kernels are placed before the kernel with the smaller output, rather than after the kernel with the larger output.

ZeroPadding kernels can take advantage of the end-of-frame and end-of-line Control-Tokens to know when to generate zeros on their outputs vs. passing through the data they receive. As their Output frame size is not merely their iteration size times their Output size (due to the addition of the zero padding), they must implement

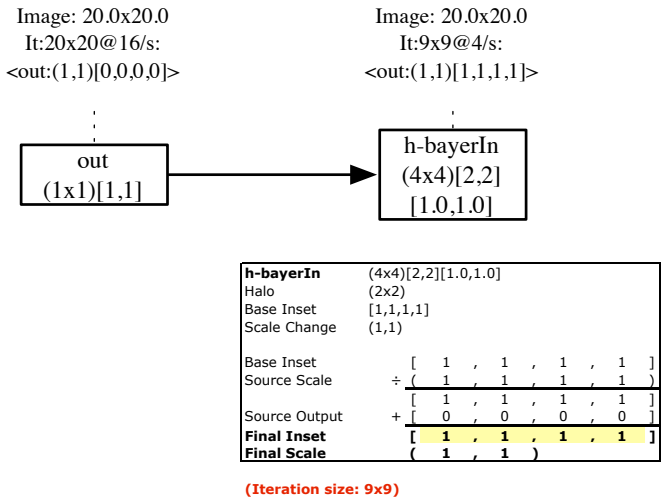


Figure 6.10: Bayer "h-bayerIn" Input inset calculations

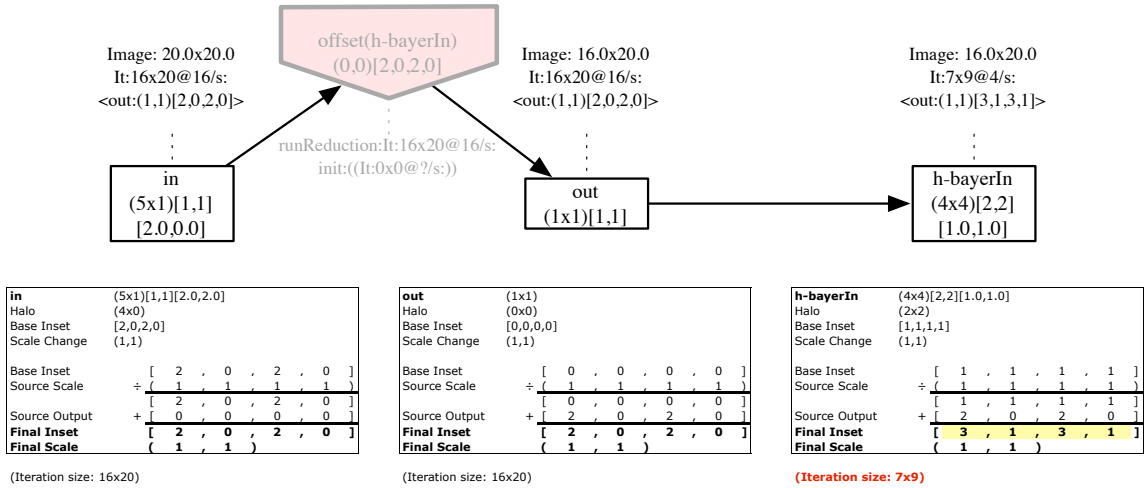


Figure 6.11: Bayer "h-bayerIn" Input inset calculations with appropriate InsetKernel With the addition of this InsetKernel, the inset and iteration size now match that of the upper path, as is seen in Figure 6.9.

an appropriate transfer function for the data analysis discussed in Chapter 5. When executing, ZeroPadding kernels simply keep track of their state based on receiving data and control tokens.

6.3 Automatic Insertion of Buffers and Insets

The data flow analysis presented here and in Chapter 5 enable the compilation system to automatically insert BufferKernels and InsetKernels/ZeroPadKernels to correct the inconsistencies discussed here. The procedure for inserting the buffers and insets starts by inserting InsetKernels. This is accomplished by analyzing the application's insets until inconsistent node is encountered. At that point, an appropriate InsetKernel is inserted to correct the inconsistency. The analysis is then restarted and repeated until no more insert inconsistencies are found. The application is then similarly analyzed for buffering, and appropriately-sized BufferKernels are inserted as needed. Since the addition of BufferKernels can change the insets in the application, this pattern of inserting insets and buffers is repeated until the application is

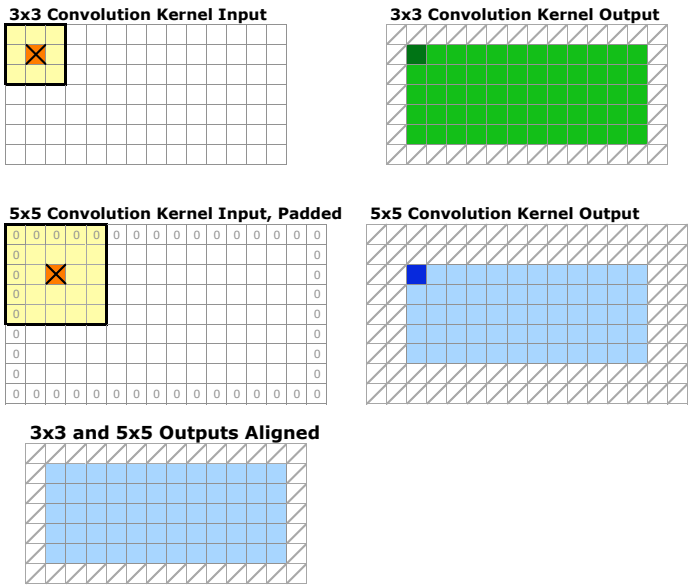


Figure 6.12: Adjusting insets by zero-padding inputs.
Zero-padding the Input to the 5×5 convolution kernel results in both kernels having the same output size. This method is similar to the removing the additional data generated by the 3×3 convolution kernel as seen in Figure 5.2.

consistent.

The results of automatically analyzing and inserting InsetKernel as needed for the differencing program can be seen in Figure 6.13(a). Automatically inserted kernels are shaded pink. As expected from Section 6.2.1, a [1,1,1,1] InsetKernel ("offset(in0)") has been added after the "conv3x3" kernel, resulting in the both Inputs to the "subtract" kernel having insets of $< out : (1,1)[2,2,2,2] >$, and consistent iteration sizes of (36×36).

The automatic insertion of buffers (Figure 6.13(b)) similarly does what one would expect, and adds buffers before both the "conv3x3" and "conv5x5" kernels to buffer enough lines of the (1×1) input chunks to output data in the required (3×3) and (5×5) chunks required by the kernels. In addition, a buffer is added after the "conv3x3" kernel to supply the required (3×3) input chunks to the InsetKernel. This is inefficient, but correct[6]. The full program graph after inset and buffer insertion is

[6]A more efficient implementation would move the InsetKernel close to the beginning of the

shown in Figure 6.13(c). This application is complete, consistent, and executable.

The Bayer demosaicing application is shown in Figure 6.14(a) after inset insertion. As expected, a $[2, 0, 2, 0]$ InsetKernel has been added to the lower branch to bring the $[1, 1, 1, 1]$ inset up to $[3, 1, 3, 1]$ to match the top branch. Note that the scales of the Inputs to the "h" kernel, however, do not match. The "h-hGIn" Input has a scale of $[0.5, 1.0]$ and the "h-bayerIn" Input is $[1.0, 1.0]$. This is addressed by comparing the step size divided by the scale to determine if the inset's scales match rather than the raw scale. This results in both having effective scales of 2, which is consistent.

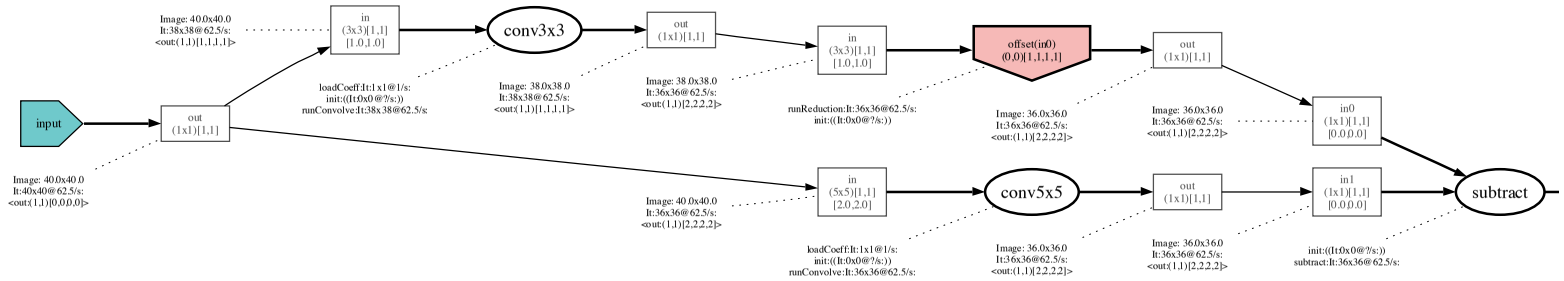
Inserting buffers into the Bayer application (Figure 6.14(b)) results in a very similar structure to that seen in the differencing program. Once the buffers and insets have been inserted (Figure 6.14(c)), the application is consistent and executable.

6.4 Discussion

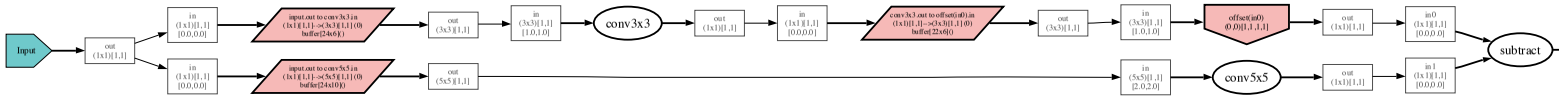
Buffer sizing for one-dimensional SDF applications is a well-studied problem. In particular, there is a tradeoff between the size of the schedule and the minimum steady-state buffering required between kernels [5, 33][7]. StreamIt inserts buffers implicitly at compilation, instead of representing them as kernels and inserting them into the application. This makes the application description less general as there is implicit buffering hidden in the structure. It is claimed that the memory sizes for the buffers generated by StreamIt are sufficiently small to fit on-chip, but they are mapped into off-chip memory for simplicity [15]. For streaming languages with only one-dimensional stream types, buffering tends to be sub-optimal for two-dimensional data as the compiler does not have full information about the stream access patterns. StreamC/KernelC is somewhat of a hybrid in this regard because the stride information for derived streams provides some information to the compiler when multi-dimensional data is mapped to one-dimensional streams.

application as possible to minimize the computation, rather than throwing out computed results as is demonstrated here. Either provide correct application execution, however.

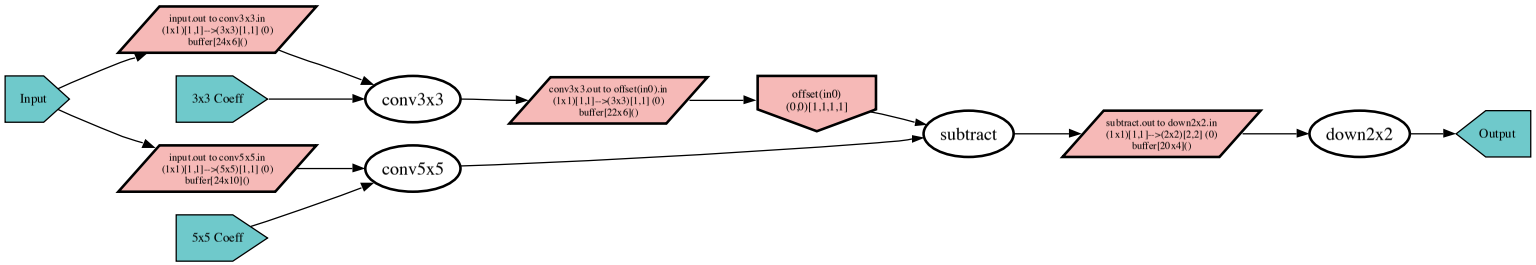
[7]The work on minimum steady-state buffer sizing is directly applicable here, but the use of buffers to adapt different input and output sizes often subsumes this role. In cases where it does not, more traditional buffer size analysis may be required.



(a) Difference program with data flow analysis and InsetKernels (first half enlargement)

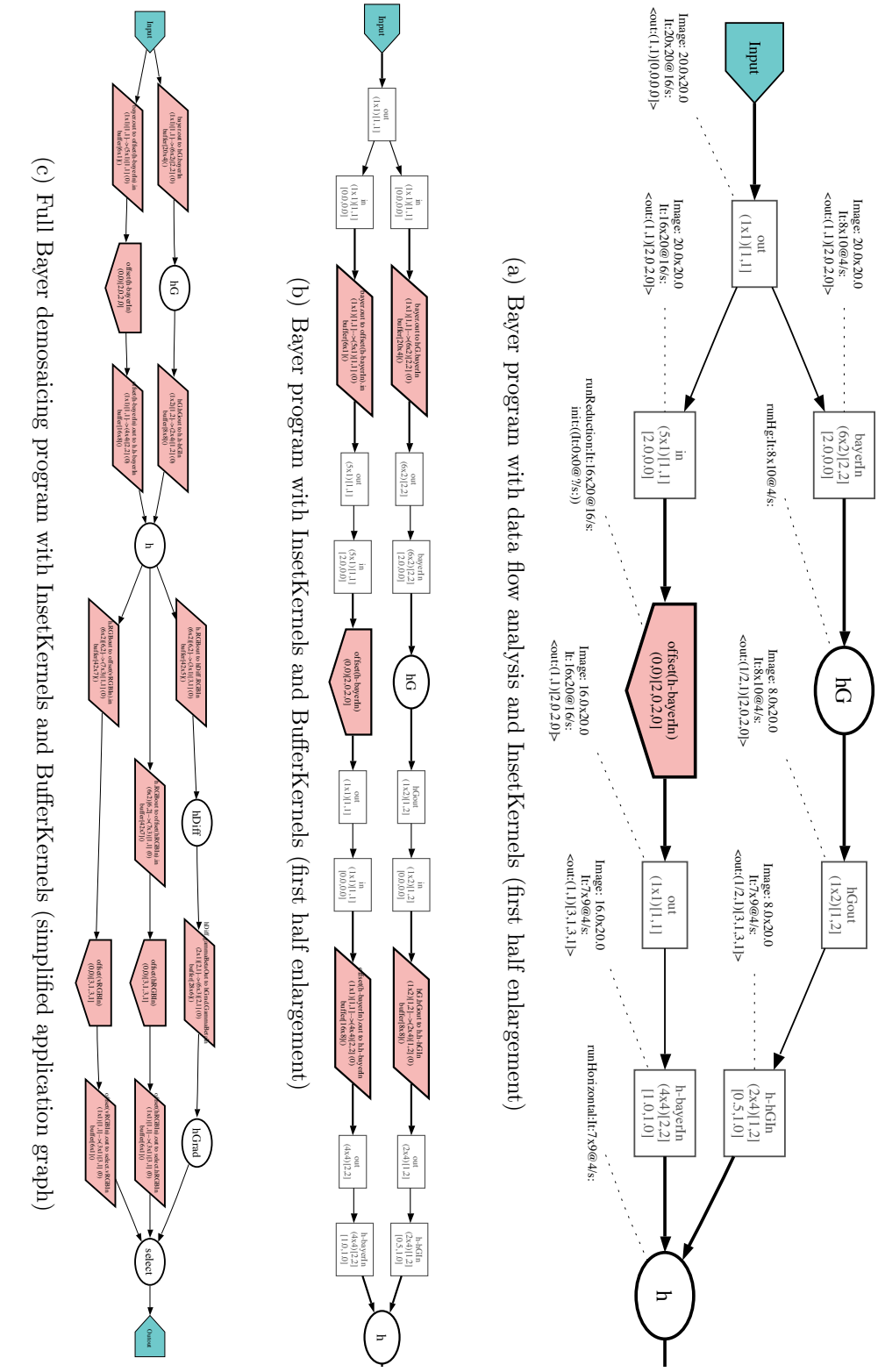


(b) Difference program with InsetKernels and BufferKernels (first half enlargement)



(c) Full difference program with InsetKernels and BufferKernels (simplified application graph)

Figure 6.13: Difference program with InsetKernels and BufferKernels
Note that 6.13(c) includes the inputs for the coefficients to the convolution filters.



(a) Bayer program with data flow analysis and InsetKernels (first half enlargement)

(b) Bayer program with InsetKernels and BufferKernels (first half enlargement)

(c) Full Bayer demosaicing program with InsetKernels and BufferKernels (simplified application graph)

Figure 6.14: Bayer program with InsetKernels and BufferKernels

Windowed SDF uses virtual tokens (combinations of output tokens) to adjust data sizes to match outputs and inputs as is done here with buffers. However, the lack of an implicit data ordering in WSDF makes buffer sizing an expensive search over both orderings and sizings. The implementation presented in [25] demonstrates that this search is possible (although slow) for two kernels at a time with a single stream between them. By specifying a scan-line ordering, the calculation of buffer sizes is greatly simplified in the implementation presented here.

The issue of insets or borders is only mentioned in the context of WSDF. They claim that their model supports the addition of border pixels to correct the same type of application inconsistencies discussed here. However, it appears that this must be done manually. Their approach should be similar to the automatic zero-padding discussed here, although it is possible that the lack of a implicit execution order may severely complicate the analysis.

Placing InsetKernels in the manner described here throws out results that will not be used later on. This is inherently inefficient as it first computes the values and then disposes of them. Instead of disposing of this data after it has been processed, it would be more efficient to push the InsetKernels as far towards the beginning of the application as possible. To do this, they would have to be adjusted to compensate for the halos of the kernels through which they were pushed, and could only be pushed forwards if the application was consistent for subsequent uses of the data long the path on which they are pushed. To achieve the best results, the analysis would need to be able to merge InsetKernels that were pushed together at merge points in the application appropriately.

Combined with the analysis from Chapter 5, the inset analysis presented here allows the compilation system to ensure that the application is consistent, and automatically adjust the application behavior to the logical behavior specified by the programmer. This relieves the programmer of the majority of the burden of keeping track of the relative sizes of data at any point in the program, and instead allows him to concentrate on the behavior of individual kernels independently. However, as seen with the need for fractional offsets in the Bayer demosaicing application, this does not completely remove the burden of understanding application details.

Chapter 7

Parallelization

The goal of writing programs in the model presented here is to allow rapid development of programs that can be readily analyzed and manipulated by a compilation system. The most important manipulations are automatic parallelization and placement of the application so as to meet its real-time constraints. The application analysis discussed in Chapter 5 enables analysis of a consistent application to determine the iteration rate of each kernel in the application and the communications bandwidth required between each kernel. Combined with the application's description of the resource requirements for each kernel method (operations per invocation and words of storage), and a description of the resources provided by each processor (operations per second and words of storage), computation of the number of processors required for each kernel is relatively straightforward.

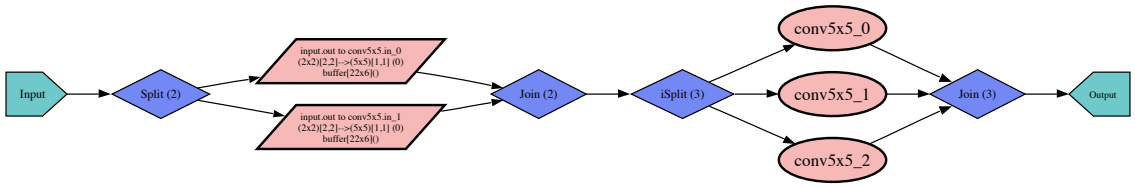


Figure 7.1: Simplified application graph for a parallelized application

The number of processors required for a kernel may be set by either its operation rate or its memory requirements, depending on the kernel and the resources available

on each processor. Parallelizing the application consists of modifying the application graph by replicating the kernels as required and inserting Split and Join kernels to distribute and collect the data in a manner appropriate to the kernels being parallelized. (See Figure 7.1.) However, the resulting application may be inefficient due to the centralized nature of the Split and Join kernels, which will be addressed in Section 7.6.2. Kernels may also require less than a whole processor to execute, in which case they should be considered for time-multiplexing with other kernels on the same processor to increase the utilization of the processor resources. This is particularly true for the Split and Join kernels themselves as they require very little computation to execute, and is discussed in Chapter 8. This chapter presents a flexible Split/Join kernel implementation and discusses how it is used to distribute data to kernels that exhibit full data parallelism or limited data parallelism.

7.1 Split/Join Kernels

The Split and Join kernels used to parallelize applications are programmable data distribution and collection finite state machines. When they are instantiated, their input (Join) or output (Split) degree is specified, and their behavior is defined by sequentially adding states and specifying how many pieces of data should be processed in each state and to or from which Input or Output the data should be sent or received. This allows for nearly complete flexibility in the use of these kernels. This capability is necessary as the type of data distribution depends on the type of parallelism available in the kernel.

Split/Join kernels handle ControlTokens in the same manner as any other kernel. Split kernels pass tokens on to all of their outputs when they receive a ControlToken. For end-of-line and end-of-frame tokens, for example, this ensures that all of the parallelized kernels receive the ControlTokens when they have completed their last iteration for the given line or frame. Join kernels pass ControlTokens on to their output when all of the inputs have the same ControlToken. For the end-of-line or end-of-frame tokens, this ensures that all of the parallelized kernels have had a chance

to complete their work on the current line or frame in order before the Join passes on the ControlToken[1].

By default, kernels use a parallelization algorithm that assumes pure data parallelism, whereby each iteration is assumed to be independent. The degree of parallelism allowed is then limited only by the presence of data dependency edges in the application (as in Figure 4.11 and Figure 4.5(c)). Kernels that are not fully data parallel can easily specify the use of their own algorithm for parallelization. This flexibility is used for parallelizing the BufferKernels as is discussed in Section 7.4, and could be used for other irregular parallelizations such as FFTs.

7.2 Data Parallel Kernels

Kernels that use the default parallelization algorithm default to a purely data-parallel parallelization. This consists of replicating the kernels and inserting a Split/Join pair around the replicated kernels that distributes the incoming data in a round-robin pattern. The finite state machine for the Split/Join kernels in this case simply sends (Split) or receives (Join) one chunk of data to/from each of the Split/Join kernel's Outputs/Inputs in order. Table 7.1 shows the FSM for the data-parallel parallelization of the convolution kernel in the simple convolution test (Figure 4.5(a)).

Table 7.1: Purely round-robin Split/Join FSM

State	Next State	Input(s)/Output(s)	Number to send/receive
0	1	0	1
1	2	1	1
2	0	2	1

In this example, the programming system has determined that three instances of the convolution kernel are required ("conv5x5_0", "conv5x5_1", and "conv5x5_2").

[1]This approach to token distribution will not have the desired behavior if ControlTokens apply to particular data outputs and not the stream as a whole. In such cases, where the intent is to "tag" particular pieces of data with ControlTokens, the ControlTokens should only be sent to the kernel that last received data. Such functionality has not been explored in this work.

This calculation was based on taking the iteration size and rate from the application analysis, multiplying it by the number of operations required by the kernel's method, and then dividing the number of operations provided per second by the target processor by that number.[2] Since the convolution kernel is data parallel and there are no data dependency edges connected to the "conv5x5" kernel in the original application graph, the kernel is replicated three times, and Split and Join kernels are added to the graph to distribute the data as shown in Figure 7.2.

The FSMs for the Split and Join kernels on either side of the parallelized convolution kernels are highlighted in the application graph shown in Figure 7.2. The notation `State: x n@[a,b,c]` indicates that state x sends or receives n Inputs or Outputs to/from Inputs or Outputs a, b, and c. The FMSs defined here are as described in Table 7.1. They serve to distribute every new input to the next convolution kernel in a purely round-robin fashion. However, this simple approach may forfeit some of the data reuse between sequential iterations as they are distributed to different kernels. This effect and a means to address it are discussed in Section 7.6.1.

Because this is a purely round-robin distribution, the data analysis keeps the iteration size and image size the same at each of the outputs of the Split kernel, but assigns each a rate of $\frac{1}{n} \times originalRate$. The Join kernel, as it is similarly purely round-robin, combines the rates of its inputs to calculate its output rate. Therefore, to floating-point accuracy, neither the rate, nor the iteration size nor image size change when comparing the `Split` $\rightarrow$ `(conv5x5_0, conv5x5_1, conv5x5_2)` $\rightarrow$ `Join` parallelization to the single `conv5x5` kernel. This is as expected because parallelizing the kernel should reduce the work load per kernel, but should not change the total work or the size of the work.

The data-parallel parallelization discussed here is quite broad and covers most kernels whose work is independent. In addition to the image processing examples presented here (where the pixels are processed independently of other pixels), operations such as motion vector searches and block encodings can be parallelized in the

[2] A more careful accounting adds in the cycles required to access the Input and write the Outputs as well as time for propagating any DataTokens that are not handled by the kernel.

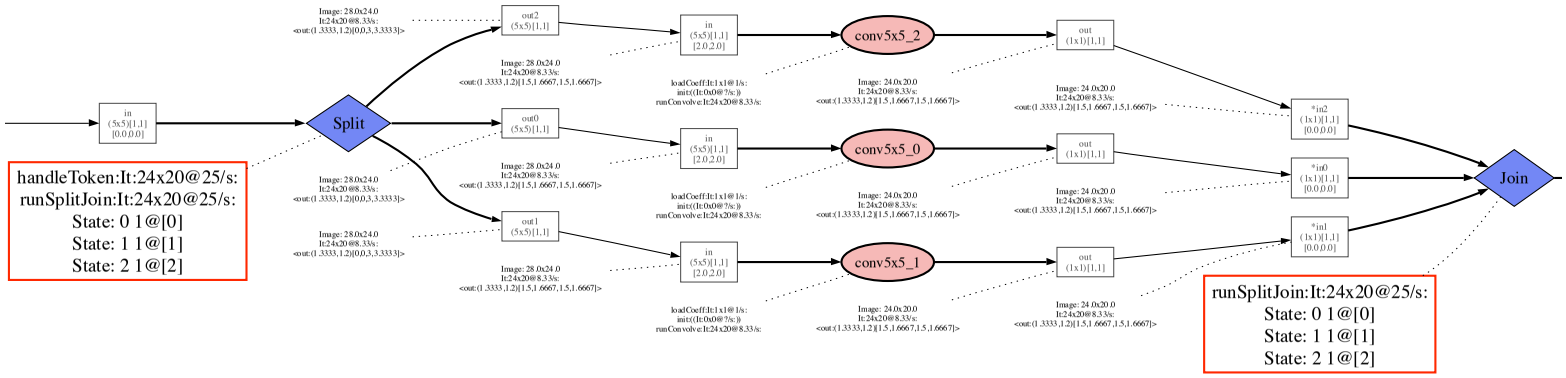


Figure 7.2: Round-robin parallelization of a convolution kernel

The FSMs for the Split/Join kernels are shown in the red boxes, matching Table 7.1. They distribute the data in a purely round-robin fashion to the three instances of the "conv5x5" kernel. The dataflow analysis shown here indicates each output of the Split kernel is running at $\frac{1}{3}$ of the input rate, (8.33/s vs. 25/s) as expected, but the final joined output is back to the full 25/s rate. The coefficient inputs to the convolution kernels (not shown here) are connected using a Replicate kernel which merely copies the data on its inputs. This is done because the coefficient Input was marked as replicated in the kernel definition. (See Figure 4.10).

same manner. All that is required for this approach is that the kernels be independent, and that the kernel resource requirements and input sizes and rates be known in advance.

7.3 Kernels with limited parallelism

The presence of data dependency edges (shown with blue dashed lines) in the application graph indicate that a given kernel's parallelism is limited by that of the edge's source. This constraint simply forces the number of instances of the edge's sink to be equal to that of the source. Therefore, if the sink requires a certain degree of parallelization to meet the required rates, the source must be equally parallelized. If the application analysis indicates that this will not meet the computation rate required, or the source can not be parallelized, then the application is incapable of meeting its constraint on the targeted hardware as the parallelism is at its maximum.

Data dependency edges allow the programmer significant flexibility in expressing the available parallelism in an application. For example, by adding a data dependency edge from an application input to a kernel, as in the Histogram program (Figure 4.11), the programmer can specify that a kernel must be serialized for each input image. The result of enforcing this constraint on the Histogram program is shown in Figure 7.3. If the program is parallelized across multiple images, this kernel could still be parallelized.

A more sophisticated example is shown in Figure 7.4. Here a serial pipeline is defined by connecting kernels B, C, and D with data dependency edges. This enforces the constraint that processing of the output of B must be done serially by C and D, but as B itself may be processed in parallel, the compiler can choose the legitimate parallelization shown in Figure 7.4(b). This parallelization may be dictated by the resource requirements of kernels B, C, or D, with the other two kernels being parallelized to the same degree to respect the constraint. By specifying such constraints, the programmer has dictated that the data generated from each iteration of B must be processed by only one instance of both C and D. This type of processing could be useful for such things as local contrast enhancement where the

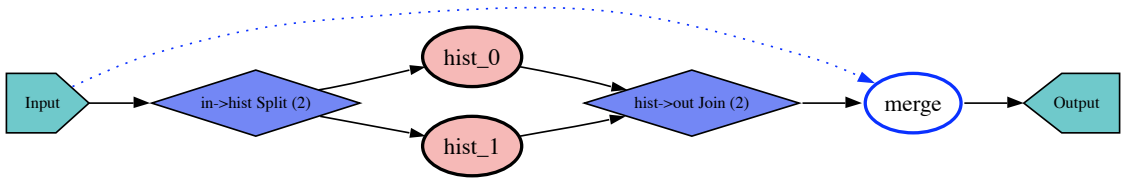


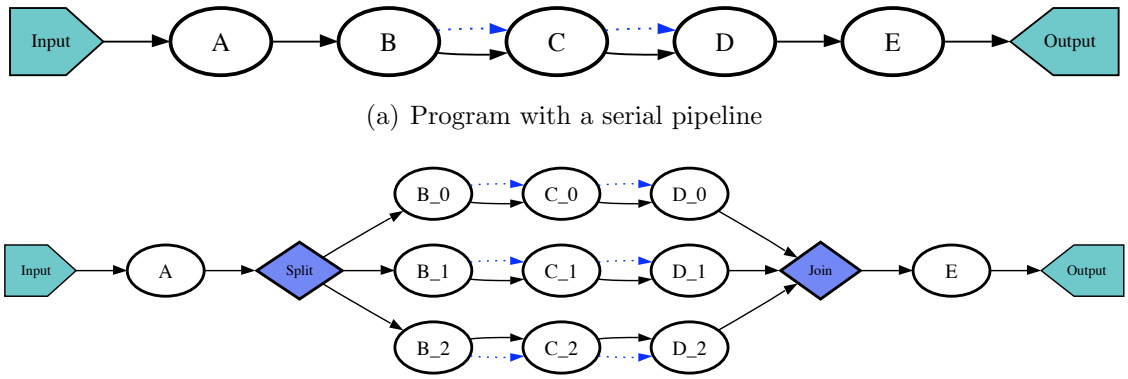
Figure 7.3: Simplified parallelized histogram with data dependency edge
The presence of the data dependency edge in this application restricts the degree of parallelism for the "merge" kernel to be equal to or less than that of the DataInput "Input". The "hist" kernel has no such limitation on its parallelism, and so it is replicated as needed.

B kernel evaluates the contrast of a block, and then tells the C and D kernels what processing to do on that block. By adding the data dependency edge, the programmer can force C and D to operate over each iteration's output from B, regardless of the particular Input and Output sizes of each kernel.

The compiler framework further enables kernels to specify their own parallelization algorithm. For data-parallel kernels, such as the "hist" kernel in Figure 7.3, the default round-robin parallelizer is used. BufferKernels use the column-replicating parallelization discussed in Section 7.4. For commutative reductions, such as the "merge" kernel in this case, it would be possible to implement a tree-reduction parallelization. The resulting tree could then be annotated with data dependency edges to ensure that those kernels are not touched by subsequent compiler manipulations. This flexibility allows the compiler framework to be extended to handle new types of parallelism, but requires that they be implemented by the programmer.

7.4 BufferKernels

Unlike the convolution kernel discussed in Section 7.2, BufferKernels are not purely data parallel, and therefore require a different type of parallelization. For the BufferKernels to function, they must collect data in sufficiently large chunks that they can



(a) Program with a serial pipeline

(b) Parallelized serial pipeline

Figure 7.4: Parallelization of a serial pipeline

The serial pipeline $B \rightarrow C \rightarrow D$ in Figure 7.4(a) can be parallelized, but the serial dependencies between the kernels as indicated by the data dependency edges must be maintained. Figure 7.4 demonstrates this parallelization while maintaining the dependencies.

generate the required output data chunks. If the data were to be distributed round-robin to parallelized BufferKernels, each one would end up with a vertical interleaving of the data which would result in incorrect output. Instead, the BufferKernels must be partitioned so that they chop up the input data in vertical slices. That is, if the BufferKernel needs to be parallelized in two, the first one should take the first half of the columns of the input and the second one the last half. However, if the Output driven by the BufferKernel has a step size that is smaller than its size (e.g., it reuses some data on each horizontal iteration), the halo data must be replicated between the split BufferKernels at the edges.

Figure 7.5 demonstrates the need to replicate data between BufferKernels. In the case of a BufferKernel feeding a (3×3) Output (the top examples in Figure 7.5), the two columns around the split point must be replicated and sent to each BufferKernel. This enables both BufferKernels to have the data that is reused across the last Output from the first buffer and the first Output from the second buffer. Similarly, if the buffer is split more than two ways, these replicated regions must be placed on both ends of any intermediate buffers as well as at the ending and beginning of the first

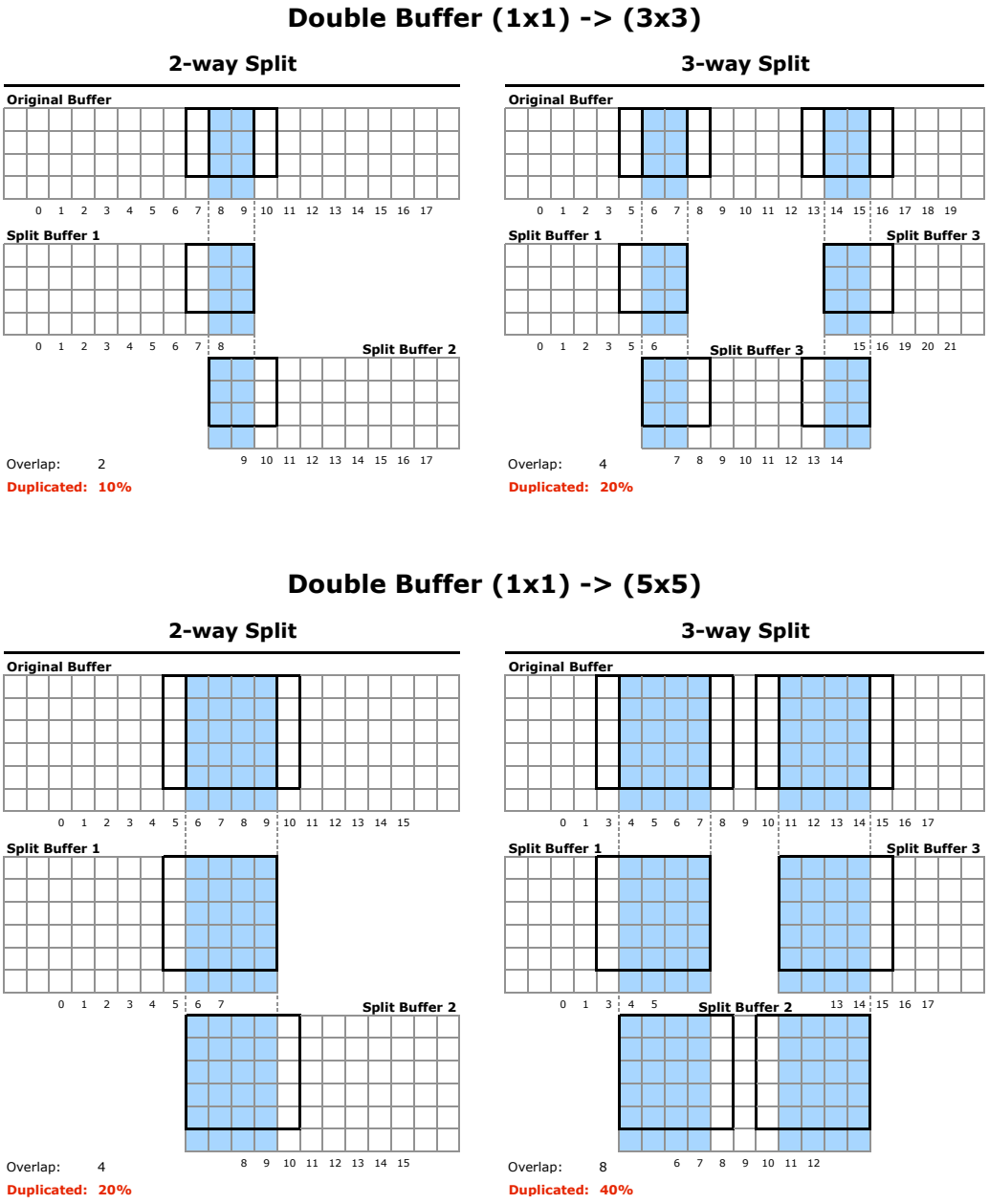


Figure 7.5: Data replication for parallelization of BufferKernels
The shaded region indicates the data that must be duplicated when the BufferKernel is parallelized. The outlines of the (3×3) and (5×5) Outputs are shown to indicate where the duplicated data is used. The iteration number is displayed under the buffer.

Table 7.2: Split and Join kernel FSMs for a (5×5) Output BufferKernel

SplitKernel

State	Next State	Output(s)	Number to send
0	1	0	8
1	2	0,1	4
2	0	1	8

JoinKernel

State	Next State	Input	Number to receive
0	1	0	8
1	0	1	8

and last buffers, respectively. Similar results hold for a BufferKernel feeding a (5×5) Output. As the halo is larger ((4×4) vs. (2×2)) for the (5×5) Output, the region that must be replicated is larger, and the percent of duplicated data increases. In general, the percent of replicated data increases with the halo size and the degree of parallelization of the buffers.

To parallelize a buffer, therefore, the new buffers must be large enough to hold the replicated data, and the Split/Join kernels must be constructed to correctly distribute and collect the data. Figure 7.5 shows the data replication for the case of two-way parallelized BufferKernel with a (1×1) input and a (5×5) Output. For each line, the Split kernel must send the first 8 values to the first buffer, the next 4 values to both buffers, and finally the last 8 values to the second buffer. When reading the data out, the the first 8 outputs come from the first buffer and the last 8 from the second buffer. The FSMs for these Split/Join kernels are listed in Table 7.2. The actual application can be seen in Figure 7.6. The full, automatically parallelized application is shown in Figure 7.7(a).

The data flow analysis for the irregular Split and Join kernels for BufferKernels is different from that of the purely round-robin ones for the data parallel kernels. In this case the frame rate stays the same, but the iteration size and image size are appropriately adjusted. This adjustment takes into account the replicated data in the parallelized BufferKernels. Between the Split and Join kernels it appears

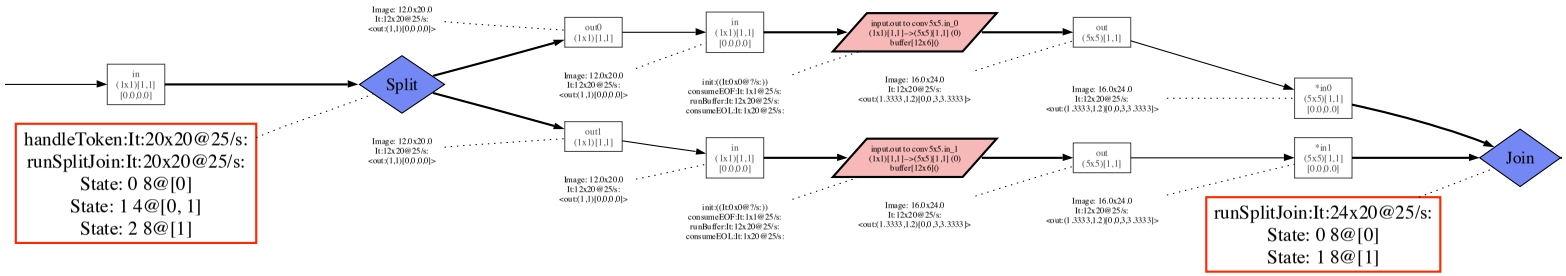


Figure 7.6: Split/Join kernels for KernelBuffer parallelization

The FSMs for the Split/Join kernels are shown in the red boxes, matching Table 7.2. They distribute the data by sending vertical slices of the input image to each of the two instances of the BufferKernel with appropriate data replication. The dataflow analysis shown here indicates that the rate remains the same for the parallelized BufferKernels but that their iteration sizes are larger than the input (two 12×20 from one 20×20) due to the replicated data.

that the frame size has increased due to this replicated data. However, the Outputs from the BufferKernels generate the same aggregate number of output iterations as they did before the parallelization. Therefore, the Join kernel adds the iteration sizes horizontally and its output is then the same as that of the non-parallelized BufferKernel. This requires that the Split/Join kernels implement different data flow analyses depending on whether they are regular "pure" round-robin distributions or irregular ones.

Calculating the size and required data overlap for BufferKernels is straightforward for buffers whose Input size is an even divisor of the Output size, as shown in Figure 7.5. For more complex buffers, the the easiest way to determine the buffer sizing and data distribution is to simulate one row worth of inputs and outputs to the buffer, and to use this simulation to determine the best splits and the required amount of replicated data. This allows the buffer sizes and Split/Join FSMs to be determined together.

7.5 Results

The full application graph for the convolution example is shown in Figure 7.7(a), and simplified application graphs are shown for it and the Bayer demosacing applications in Figures 7.1 and 7.7(b). These applications were transformed from the original application descriptions fully automatically, including inserting InsetKernels and BufferKernels (as discussed in Chapter 6) and parallelizing the computation kernels to meet the input requirements.

Further examples of the automatic analysis, buffering, and parallelization are seen in Figures 7.8-7.11. These four application graphs show the result of changing the input parameters to the differencing example to force the application to parallelize for memory, computation, or both, as described in Table 7.3.

(a) Full automatically parallelized convolution program

(b) Simplified automatically parallelized Bayer demosaicing program

Figure 7.7: Automatic parallelization examples

The replicated coefficient inputs to the convolution kernels in Figure 7.7(a) are indicated by dashed edges.

Table 7.3: Automatic parallelization examples

	Small Input Size	Large Input Size
Low Input Rate	Figure 7.5 "Small/Slow"	Figure 7.5 "Big/Slow" *storage-limited*
High Input Rate	Figure 7.5 "Small/Fast" *compute-limited*	Figure 7.5 "Big/Fast" *compute- and storage-limited*

The same application description was used for all four instances. Only the input size and rate was changed.

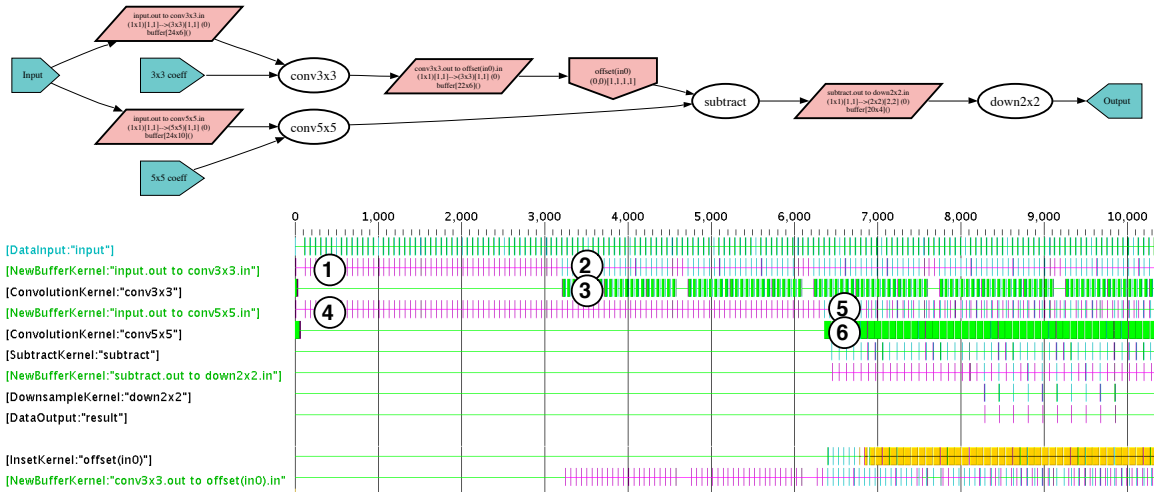


Figure 7.8: Baseline differencing program ("Small/Slow")

For this baseline combination of processor and input size and rate no parallelization is required. A key for the color-coding used in the timeline displays can be found in Figure B.3. For Figures 7.8-7.11: 1) magenta: writing into the 3x3 buffer(s), 2) cyan/magenta: reading/writing from/to the 3x3 buffer(s), 3) green: "conv3x3" kernel execution, 4) magenta: writing into the 5x5 buffer(s), 5) cyan/magenta: reading/writing from/to the 5x5 buffer(s), 6) green: "conv5x5" kernel execution. Yellow indicates stalls waiting for output.

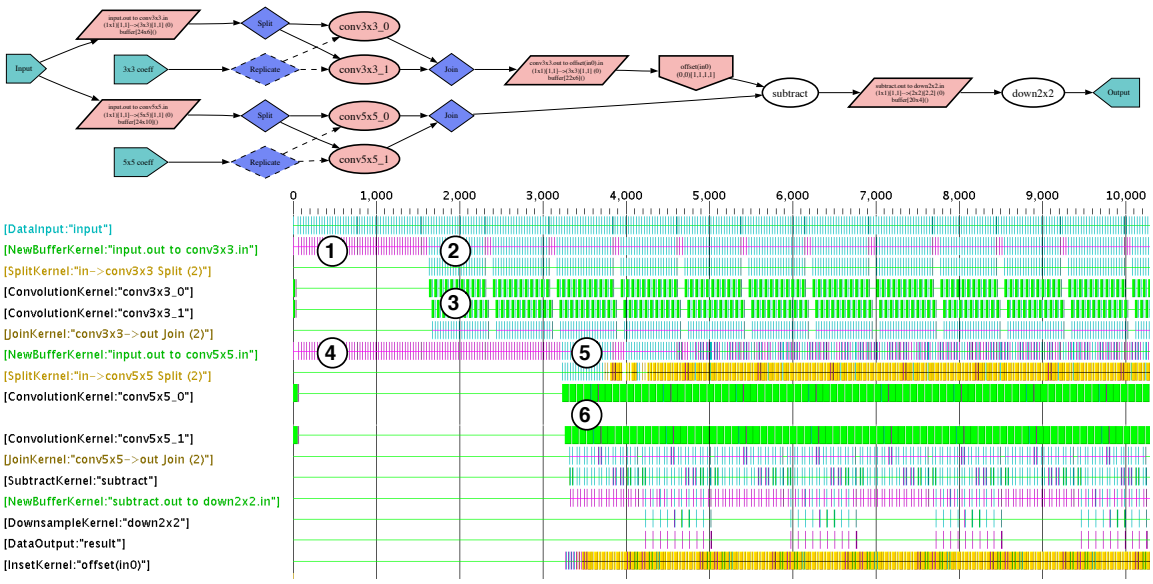


Figure 7.9: Differencing program with increased input rate ("Small/Fast")

Doubling the input rate over the baseline in Figure 7.8 requires that the limiting computation kernels (in this case the "conv5x5" and "conv3x3" kernels) be parallelized to meet the increased input rate.

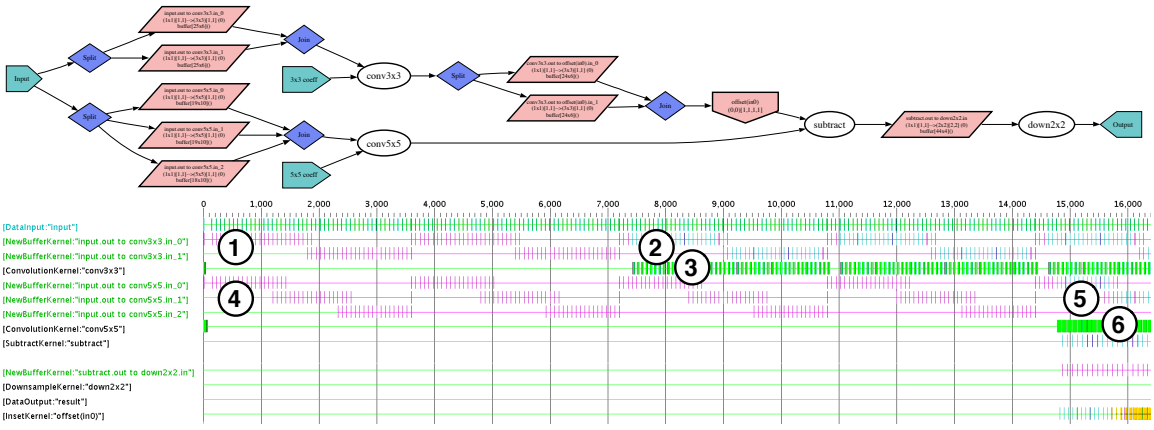


Figure 7.10: Differencing program with increased input size ("Big/Slow")

Keeping the throughput constant but doubling the input size over the baseline in Figure 7.8 requires that the limiting storage kernels (in this case all three of the buffers) be parallelized to fit within the limited memory resources of each processor. Note that due to the required overlap between buffers as seen in Figure 7.5, three buffers are required for the "conv5x5" kernel when the input size is doubled.

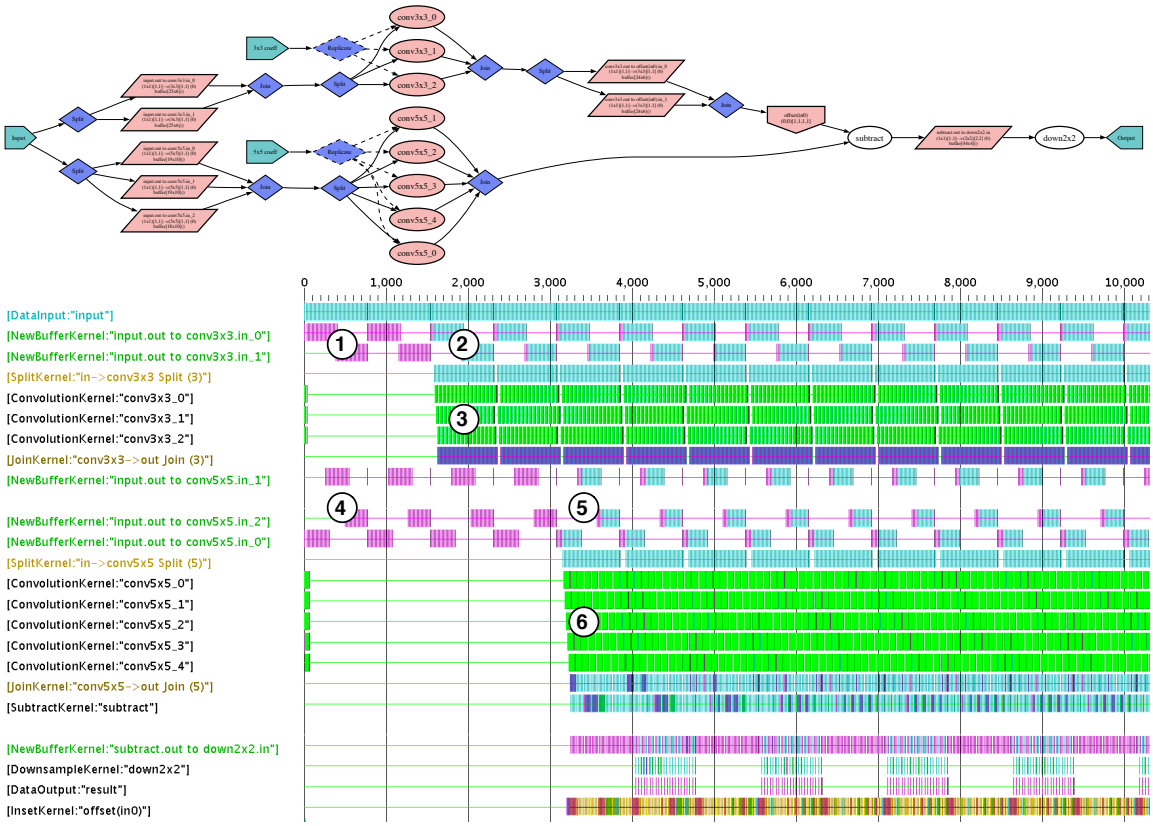


Figure 7.11: Differencing program with increased input size and rate ("Big/Fast")

Doubling both the input size and rate over the baseline in Figure 7.8 requires that both the limiting storage and computation kernels be appropriately parallelized.

7.6 Discussion

7.6.1 BufferKernel Data Reuse

The parallelization of buffers discussed in Section 7.4 may limit an application's ability to take advantage of data reuse between iterations of computation kernels. This is due to the round-robin distribution of the data to the parallelized computation kernels, which effectively destroys data locality between kernel iterations by spreading the spatially local iterations across physically disparate processors. For example, the simple convolution kernel shown in Figure 7.12, when parallelized (Figure 7.13), sends every other convolution to alternating convolution kernels. The resulting data reuse is shown in Figure 7.14. The two-way round-robin parallelization results in a data reuse for each of the convolution kernels of 15 of the 25 input elements, as each iteration must receive the five new elements for that iteration, and the five elements that were new for the previous iteration, but which were sent to the other convolution kernel. If subsequent iterations of the "conv5x5" kernel were executed on the same processor instead, the data reuse would be 20 of the 25 input elements in steady-state. Similarly, a round-robin parallelization across three processors would result in a reuse of only 10 of 25. For six or more processors the reuse is eliminated entirely, since each kernel executes every sixth iteration, and the input size for the kernel is only five.

To avoid eliminating this inter-iteration reuse, the buffers can be parallelized to match the kernels which they are feeding. This would change the round-robin distribution of data to the vertical-slices required for parallelizing buffers, as described in Section 7.4. For the example presented above, this results in the application graph in Figure 7.15. This transformed application has the buffer kernel split in two, with appropriate Split and Join kernels. The resulting data distribution pattern sends the first 8 iterations to "conv5x5_0" and the second 8 to "conv5x5_1". As the iterations that each kernel processes are now both spatially local (iterations 1-8 for "conv5x5_0" and 9-16 for "conv5x5_1"), nearly the full data reuse across the 8 iterations can be utilized. To do so, a static copy elimination would use the known data usage patterns from the kernels' inputs and the buffer's output to determine that only the new data on each iteration was required to be transmitted between the buffers and the

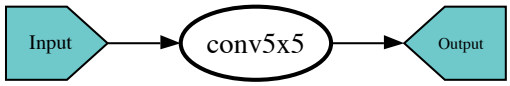


Figure 7.12: Simple 5×5 convolution application graph

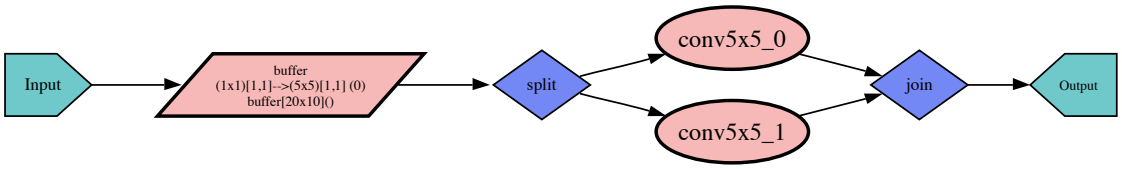


Figure 7.13: Automatically buffered and parallelized 5×5 convolution program
The basic parallelization applied here to the program in Figure 7.12 uses a purely round-robin Split kernel to distribute the work to two convolution kernels. The buffer is not parallelized because it is not resource constrained.

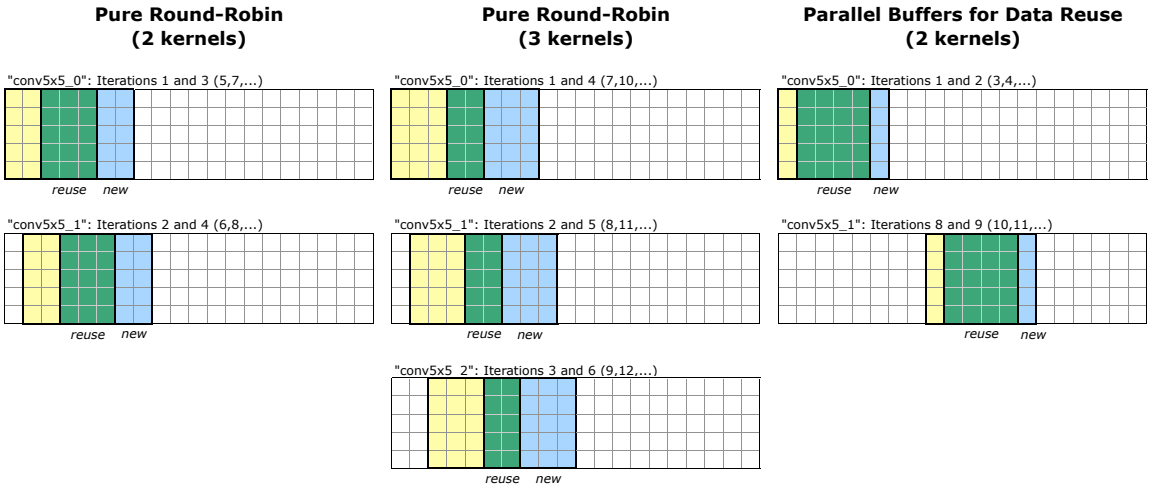


Figure 7.14: Data reuse options for a 5×5 convolution
The data reuse for a 5×5 convolution kernel is shown. The new data for each iteration is shown in light blue, while the data that can potentially be reused is shown in dark green. The pure-round-robin parallelization reduces data reuse by sending consecutive iterations to different processors (left and middle). A more intelligent distribution (right) can maximize reuse by sending consecutive blocks of iterations to the same processor.

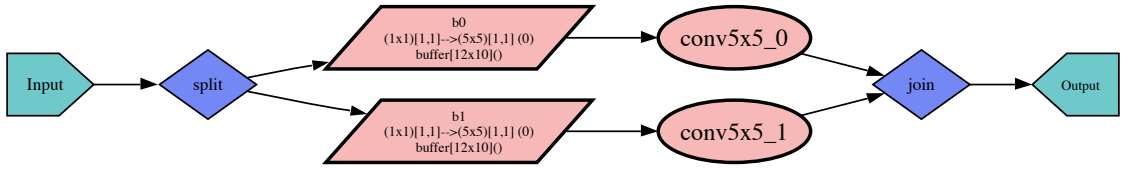


Figure 7.15: Naïve buffer parallelization for reuse
The buffer feeding the convolution kernels in Figure 7.13 has been replicated to allow each convolution kernel to receive data from one buffer directly, thereby increasing the potential for data reuse.

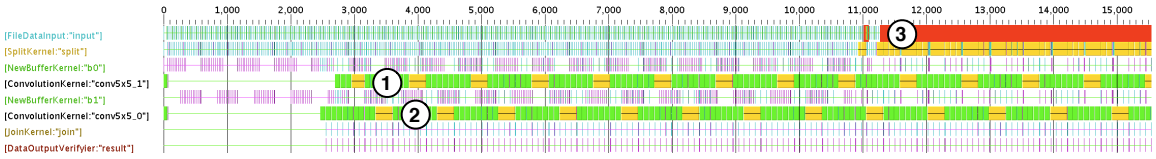


Figure 7.16: Simulation trace of naïvely parallelized buffers
The results of running the program shown in Figure 7.15 indicate that the program input is forced to stall (3) at cycle 11,000 due to the lack of buffering between the convolution kernels and the Join kernel. The convolution kernels show low utilization as indicated by the large amount of time spent waiting to write their outputs (1 and 2, yellow with black horizontal lines) compared to time spent executing (green).

convolution kernels. The data overlap between the two buffers, however, would still require data duplication as in Figure 7.5. Given known costs for data movement, storage, and duplication, it would be possible to choose the optimal distribution for each buffer/kernel pair. Determining the optimal distribution for an entire application would be significantly more difficult as the local buffer/kernel choice has an impact on the global result.

While parallelizing the buffer to match the required parallelization of the computation kernels enables the exploitation of greater data reuse, it does not work in all situations. Indeed, the example presented in Figure 7.15 will not operate as expected. The problem is that the Join kernel expects to receive iterations 1-8 from convolution kernel "conv5x5_0" before receiving any data from "conv5x5_1". This means that "conv5x5_1" will stall after its second iteration because it will have filled its Output's

buffer and the buffer in its Input to the Join kernel. From that point on, it will not be able to proceed until all 8 of the outputs from "conv5x5_0" have been processed. This results in a steady-state behavior wherein each of the parallelized kernels must wait for the other to execute before it can continue, which will not meet the real-time constraints of the application if two were required in the first place. This behavior can be seen in the simulation trace in Figure 7.16. The yellow blocks with the black lines through them for the two convolution kernels are the time they spend stalled waiting for the Join kernel to receive their outputs. The result is that the application fails to process data at its input rate, and therefore stalls the input at cycle 11,000 when the buffers fill.

To address the issue seen in Figure 7.16, there must be an output buffer for each of the parallelized input buffers. Such buffers ensure that both computation kernels can be kept busy at all times regardless of the state of the Join kernel at the end. These output buffers can either be inserted for this purpose, or any down-stream buffer in the application can be used. By inserting such buffers, the application is transformed as seen in Figure 7.17, and the runs correctly as demonstrated by the simulation trace in Figure 7.18.

The transformations described here can be thought of two ways: either as separately parallelizing the buffers and then merging the Join/Split kernels between the buffers and the computation kernels, or as moving the Split for the computation kernels through the application graph in front of the buffer kernel. In either case, the round-robin data distribution for the computation kernels needs to be replaced with the vertical-slice data distribution required for the buffer, which is how the data locality reuse is realized. In general, multiple computation kernels can exist between the input and output buffers. As long as no Buffer Kernels are required between the computation kernels this approach will work without modification. However, if the Input and Output sizes of the kernels are such that buffering is required, the implementation becomes more complicated, requiring either replication of computation or complex distribution of intermediate results.

These transformations enable buffers to be parallelized to maximize data reuse to the kernels they feed. When combined with time-multiplexing the buffers on the

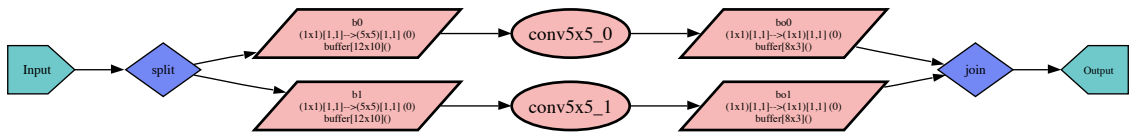


Figure 7.17: Correctly parallelized buffers for reuse

The addition of buffers after the convolution kernels allows the second convolution kernel to continue processing while the Join kernel takes results from the first convolution kernel. This allows maximum data reuse across the convolution kernels, but at the cost of additional buffering.

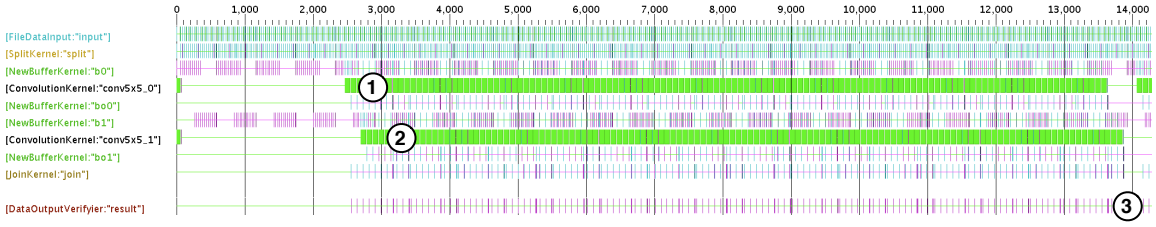


Figure 7.18: Simulation trace of correctly parallelized buffers for reuse

The results of running the program shown in Figure 7.17 indicate that the program does not stall and in fact completes (3) as expected on cycle 13,878. Note the high utilization (lack of output stalls) of the convolution kernels at (1) and (2) in this program trace compared to that of Figure 7.16.

same processor as the kernels, this enables maximum data reuse by keeping the data local to the processor that will consume it. The tradeoff is the need to parallelize the buffers which results in increased storage and computation overhead. In addition to enabling increased reuse, these configurations can reduce the required communications bandwidth between kernels. In the example in Figure 7.17, both buffers can simultaneously output data to the computation kernels. Compared to the default implementation of Figure 7.13, this doubles the bandwidth available to feed the computation kernels as well as reducing the required bandwidth by increasing the reuse.

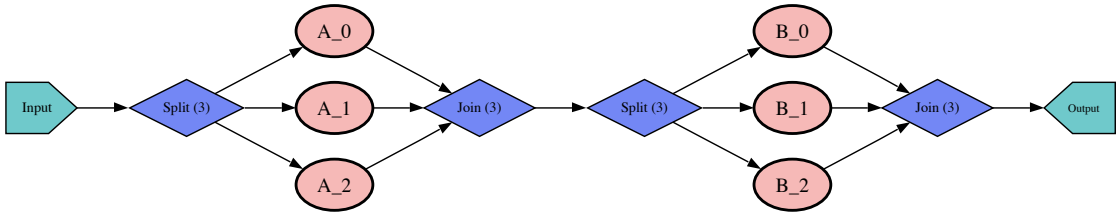
7.6.2 Split/Join Inefficiencies

While the parallelized applications presented in Chapter 7 are all correct in the sense that they produce the desired output and meet the required input rates given the specified hardware resources, they are clearly not optimal. As can be seen in all of these graphs, and is explicitly illustrated in Figure 7.19(a), the approach of simply replicating a kernel as required and then inserting Split and Join kernels around it leads to an overall program structure which consists of Split→Kernel→Join followed immediately by another Split→Kernel→Join. This program structure can be inefficient because of its forced centralization of data movement and the difficulty of load-balancing the Split/Join kernels.

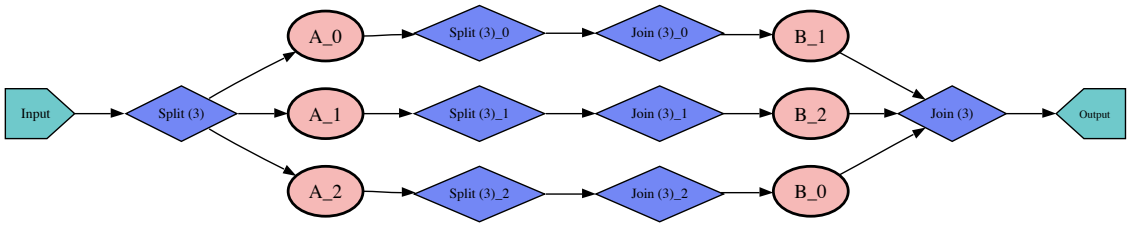
For example, in Figure 7.19(a), all data from the parallelized kernels is forced to come together at centralized collection and distribution points, even if the pattern of the data movement would not require it. Here kernels A and B in are both replicated 3 times, so it would seem be wasteful to have the outputs from the As collected by a centralized Join kernel immediately before distributing them to the Bs via a second Split kernel. Such distribution can be inefficient as well as they can waste interconnection bandwidth by moving all data through a central point. One solution, if valid, would be to eliminate the intermediate Join/Split kernels and directly connect the A and B kernels, as illustrated in Figure 7.19(c).

Along similar lines, having centralized Split and Join kernels makes load balancing difficult. The Split and Join kernels require very little storage and processing time, which makes them prime candidates for sharing a processor with another kernel in a time-multiplexed manner. However, pairing the single Split or Join kernel with one of the multiple parallelized processing kernels on a single processor results in a load imbalance. If the processing kernels have been parallelized such that their expected utilization is high, then the addition of the Split or Join kernel to any of them may exceed the processor's resources.

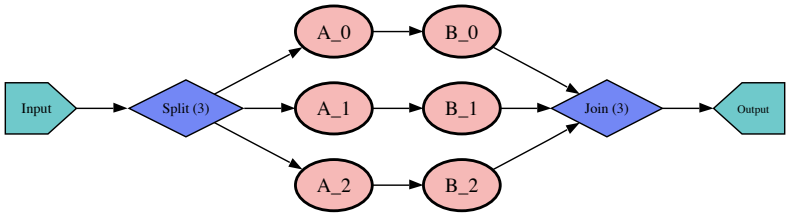
Both of these issues can be addressed by distributing the Split and Join kernels among the computation kernels they serve. This effectively partitions the finite state machines in the original Split and Join kernels such that a separate Split or Join kernel can be created for each computation kernel. The result of this transformation



(a) Inefficient Split/Join structures from simple parallelization



(b) Distributed Split/Joins

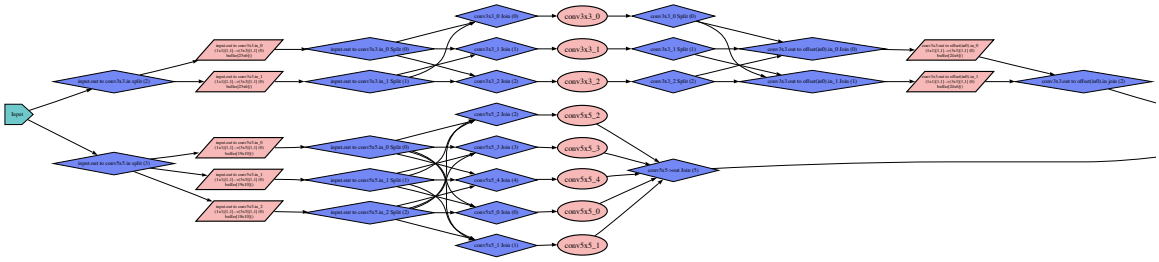


(c) Elimination of redundant Split/Joins

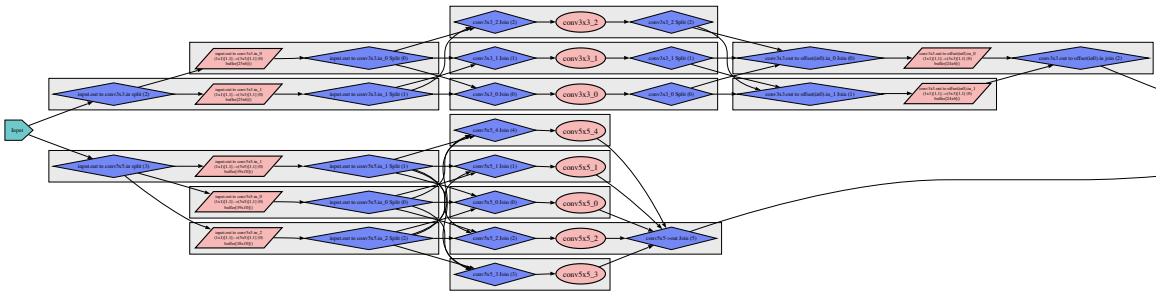
Figure 7.19: Split/Join optimization examples

is that each data-parallel kernel has its own Split and Join kernel, which can be easily time-multiplexed in a load-balanced fashion and distributed for optimal bandwidth utilization. Furthermore, it may be possible to eliminate the resulting individual Split/Joins if their data movement is fixed, as shown in Figures 7.19(b)-7.19(c).

While this approach evenly distributes the Split/Join overhead among the computation kernels, it also been increases the overhead as there are more Split/Join kernels that must be executed. This proliferation is demonstrated in Figure 7.20(a), where the program from Figure 7.11 has had its Split/Join kernels automatically distributed. Alternatively, for data-parallel computations, the distribution could be adjusted to be not perfectly round-robin, thereby reducing the processor utilization of one of the



(a) Proliferation of Split/Join kernels when distributed



(b) Time-multiplexing of distributed Split/Join kernels

Figure 7.20: Split/Join distribution and time-multiplexing

The results of automatically distributing the Split/Join kernels from the parallelized program in Figure 7.11 (top) and using the distributed kernels for a load-balanced time-multiplexing (bottom). The gray boxes in Figure 7.20(b) indicate the kernels that have been selected to be time-multiplexed on the same processor. As the Split/Join kernels in this program do not execute a static distribution they can not be eliminated, thereby incurring a significant overhead compared to the program in Figure 7.11.

parallelized kernels, which could then be time-multiplexed with the non-distributed Split/Join kernels without incurring as much of a load imbalance.

7.6.3 Analysis

The parallelization problem addressed here is most similar to that targeted by the StreamIt compiler [16, 15]. StreamIt attempts to maximize throughput by optimizing load balance and minimizing synchronization across a fixed number of processors. The approach presented here is to determine the minimum number of processors to meet

the real-time constraints imposed by the application's input sizes and rates. While these may appear fundamentally different, they both have the same issues of optimally utilizing resources. For an application with real-time constraints, the goal is to utilize the fewest number of resources, while a maximum throughput approach attempts to make the best use of a fixed number. Either approach can be crudely mapped to the other by wrapping the compilation process in a loop that either adjusts the real-time constraint until the desired number of processors are utilized, or adjusts the number of processors until the desired real-time constraint is achieved.

StreamIt's splitjoin filters differ from the Split/Join kernel presented here in two important ways. First, they are regular kernels and not hierarchical containers. This implies that the computation and storage resources required to implement the data distribution and collection can be incorporated transparently in the overall application analysis. Secondly, the finite state machines in the these Split/Join kernels are not limited to round-robin data distribution, and can replicate data, for example to distribute data to parallelized buffers. This functionality could be reproduced with a significant degree of complexity in StreamIt by using multiple splitjoins and data replications. The tradeoff is that it is possible to create Split/Join kernels that are hard to analyze, and the application analysis that inserts them may need to keep track of its intentions separately, as extracting them from the constructed Split/Join kernels may be difficult.

The basic parallelization presented here is sufficient to ensure the application meets its real-time constraints, but it is unlikely to result in an efficient implementation. This is due in large part to the proliferation of Split/Join kernels with their low CPU and memory utilization, which need to be load-balanced with the high-utilization computation kernels. (See Section 7.5.) More generally, this simple approach maps each resulting kernel in the parallelized application to a separate processor tile which is unlikely to result in high utilization. StreamIt deals with this issue by applying varying degrees of kernel fission to pipelined data-parallel kernels, and using software pipelining to enable flexible scheduling of pipelined, but data dependent, kernels. The end result is a time-multiplexing of processors wherein each processor executes multiple kernels over time, to potentially improve load balancing. Implementing such

time-multiplexing requires that the buffering between the kernels to be multiplexed be reasonable, and reasonably calculable. Both StreamIt [15] and Array-OL [4] found this to the case for certain combinations of filters/kernels, but not for others. Chapter 8 discusses such time-multiplexing to improve utilization.

7.6.4 Other Access Patterns

Matrix multiplication is a common operation, but has a very different reuse pattern from that of the image processing kernels discussed heretofore. This difference makes it difficult to map matrix multiplication to the block-parallel framework. The fundamental problem is that matrix multiplication accesses data in both column- and row-order, which is awkward within the fixed scan line ordering presented here. In addition, matrix multiplication re-uses the entirety of one matrix for each row (or column) in the other matrix, which is not readily expressible with the given Input/Output parameterization.

The simplest implementation of a two-dimensional matrix multiplication, $C = AB$, would consist of a single kernel to implement the matrix multiplication with two inputs A and B, each the size of the matrices to be multiplied. This would result in sufficient input buffering being instantiated for the kernel to hold the entirety of each input matrix A and B, and would not be data-parallel, thereby preventing parallelization. A slightly more sophisticated implementation might add a kernel to replicate B a sufficient number of times to allow each row of A to be multiplied in parallel by a matrix-vector multiplication kernel. With the addition of this replication kernel, the rest of the analysis and parallelization would be supported by the system presented here. However, such an implementation might exhibit poor data locality as B would be replicated and sent out once for each row in A.

A proper matrix multiplication implementation would provide the compiler with a parallelization algorithm which could determine how to parallelize the matrix multiplication, and generate the correct, potentially heterogenous, series of kernels to implement that parallelization. Such an algorithm would need to block the input data appropriately to fit in on-chip memory, and transfer intermediate results to and

from off-chip storage as needed. However, this complex data movement is a function of the algorithm, and not a requirement of the programming system. While this system can support such an implementation, writing the particular parallelization algorithm is roughly as complex as in other programming environments.

Another common operation is a fast Fourier transform, or FFT. These operations are typically parallelized using a butterfly structure of alternating data exchanges and computation, and can map well to pipelined implementations. As with matrix multiplication, such non-scan line data ordering is difficult to map to the Input/Output parameterization supported here. To enable FFT kernels to be parallelized, an appropriate parallelization algorithm must again be implemented for the compilation system.

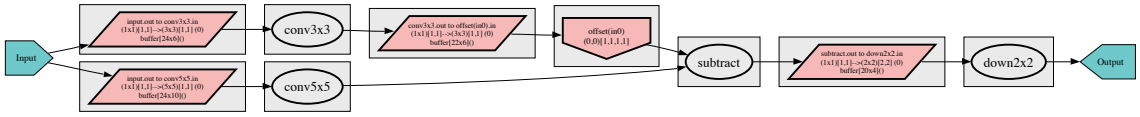
These two examples of non-windowed data access demonstrate that while this framework can support arbitrary algorithms, the limitations of the data movement parameterization supported by the block-parallel approach may require that their analysis and parallelization be implemented specifically for the algorithm. This trade-off acknowledges that trying to provide either a too general automatic parallelization capability or too general input parameterization makes implementing an effective compiler extraordinarily difficult. Instead, this framework provides the capabilities to automatically and efficiently handle simple two-dimensional windowed access data-parallel kernels, and allows the programmer to extend it to handle more specific cases.

Chapter 8

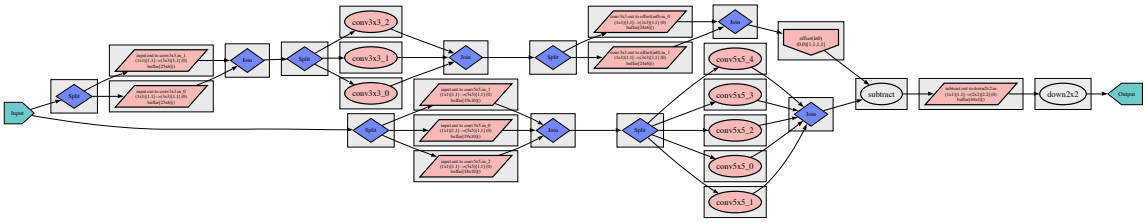
Time Multiplexing

Once a program has been analyzed and parallelized it must be mapped to physical processor resources. This mapping has two interdependent aspects: physical placement of kernels on processors (see Appendix A) and time-multiplexing of multiple kernels on one processor. The physical mapping and multiplexing are not fully independent as the selection of which kernels to time-multiplex will have a direct impact on the communications patterns, which will therefore change the optimal physical placement. However, if the communications costs are assumed to be independent of the physical mapping, as is the case for the simulated results presented here (see Appendix B), these two steps can be treated as independent.

This chapter discusses the need to time-multiplex kernels to achieve higher overall processor utilization, and presents a simple greedy algorithm for doing so. The algorithm is first motivated by examining the utilization of the naïve 1:1 mapping of kernels to processors implicitly presented in the previous chapters. These results are used to determine how low-utilization kernels can be time-multiplexed with other kernels to increase the overall utilization, without exceeding the available computation or storage resources. Finally, the result of applying all the analyses and transformations discussed in this thesis, including the time-multiplexing of the kernels, is compared across a range of programs.



(a) "Small/Slow" program from Figure 7.8



(b) "Big/Fast" program from Figure 7.11

Figure 8.1: 1:1 kernel-to-processor mapping
The implied naïve 1:1 processor-to-kernel mappings for two of the applications from Chapter 7 are shown. The individual kernel groups are shown as gray boxes around the kernels included within them. Each kernel group indicates the kernels that will be executed in a time-multiplexed manner on a single procoressor. Coefficient inputs are not shown for simplicity.

8.1 Naïve Mappings

The previous chapters have implicitly assumed a naïve 1:1 mapping of kernels to processors. That is, in each of the examples presented so far, it was assumed that each computation kernel, split/join kernel, and buffer kernel was mapped to its own processor. This can be clearly seen in Figure 7.11 where each of the kernels in the application graph has a corresponding trace in the simulator output. The explicit 1:1 mapping for that example is shown in Figure 8.1(b), where each of the kernels in the program has been placed in its own *kernel group*. The kernel groups for an application determine which kernels will be time-multiplexed together on a given processor. The time-multiplexing is implemented as a round-robin cooperative scheduling of the kernels, where each kernel yields to the scheduler when it finishes invoking a method or is stalled waiting for an input or output. (See Appendix B for more details on the simulator implementation.)

While this 1:1 mapping results in a functionally correct application, it is unlikely

to result in a high-efficiency implementation. This is due to the inherent low utilization of the processors dedicated to executing simple kernels, such as splits and joins. Having underutilized processors in the implementation is inefficient due to their power consumption and area requirements. While it is standard practice to put idle processors in low-power sleep states, the transitions to and from sleep incur penalties in both performance and power for the time and power it takes to enter and exit sleep. Furthermore, underutilized processors increase the die area required to implement the program, which increases chip cost. To both minimize power and area it is important to keep the minimum number of processors as highly utilized as possible[1].

The efficiency of these 1:1 mappings can be seen in Figure 8.2[2] for the programs shown in Figure 8.1. The overall results are not impressive. The "small/slow" and "big/fast" programs achieve an average of just 20% and 32% utilization, respectively. (The "Big/Slow" application has higher overall utilization only because it has a higher percentage of computation kernels due to its higher data rate.) In the case of the "small/slow" program, the single "conv5x5" kernel achieves 93% utilization, while the "conv3x3" kernel requires roughly only a processor. The "big/fast" program shows that the "conv5x5" kernel was parallelized into five instances, with each instance operating at 91% utilization. Similarly, the "conv3x3" kernel was parallelized three times, with each instance running at 75% utilization. When taken by themselves (see "Computation Average" in Figure 8.2), the computation kernels have a significantly higher average utilization (38% and 71%) for the two programs, even including the "subtract" and "down2x2" kernels which do very little work. In both of these cases, the computation kernels are correctly parallelized given the constraint of an integer number of evenly-load balanced kernel instances and the real-time requirements.

The low overall average utilization ("Average" in Figure 8.2) is due to two issues:

[1]It should be noted that other issues may change this formulation. For example, if instruction storage is highly constrained, it might be more energy efficient to put a processor to sleep and execute other kernels on different processors than to reload instructions from a higher-level memory in order to multiplex multiple kernels on the same processor.

[2]Figure 8.2 has the amusing property that it acts as an optical illusion. To see this, tilt the page away from yourself while examining the long striped lines in the graph. They will appear to bend as you tilt the page.

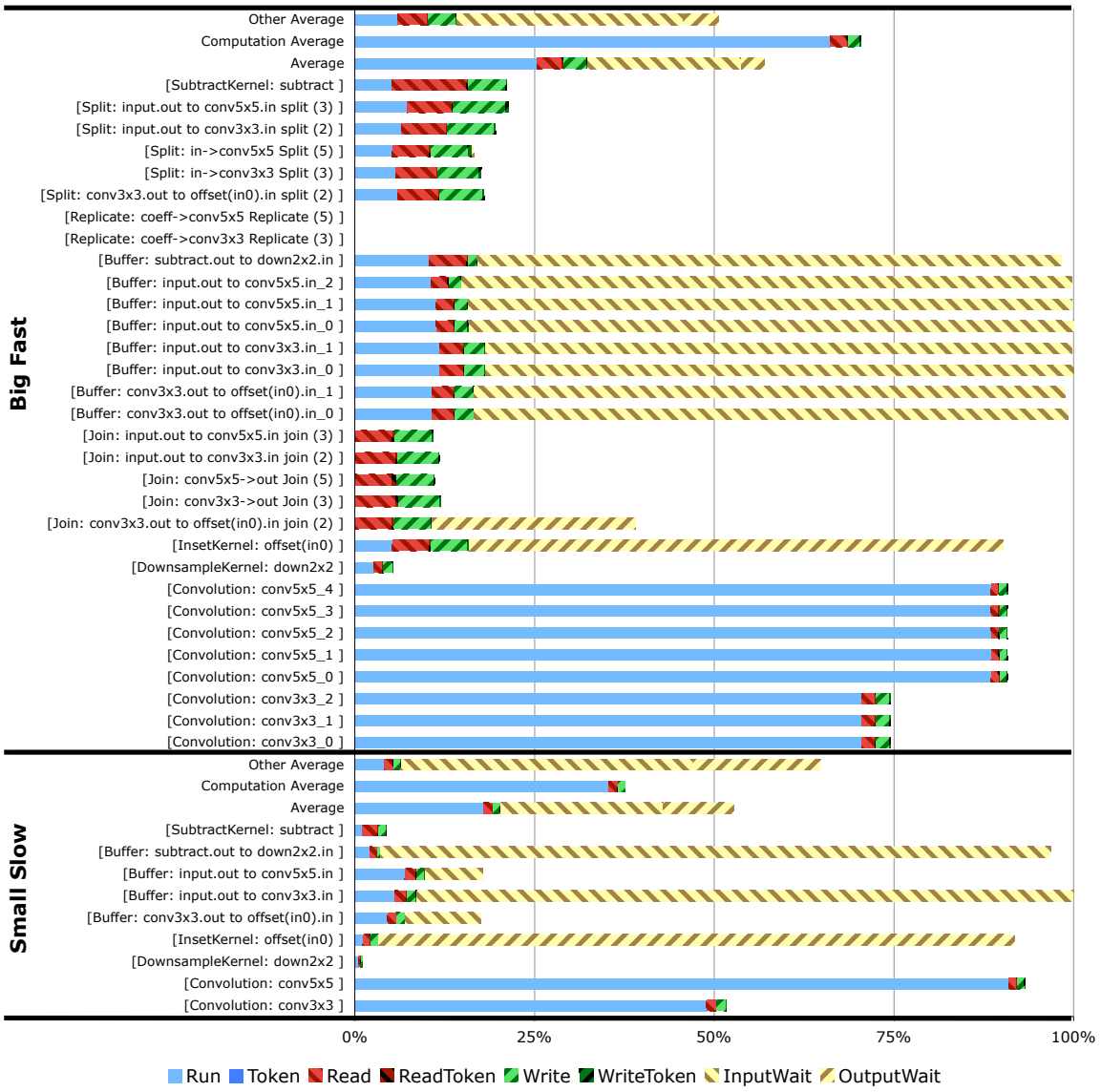


Figure 8.2: 1:1 mapping utilization

Processor utilization for a 1:1 kernel-to-processor mappings of the programs shown in Figure 8.1. The run time (light blue) is the time spent doing useful work. The InputWait and OutputWait (striped yellow) bars represent time the kernels are waiting to receive or send data, respectively. "Computation Average" shows the utilization averaged over the computation kernels, and "Other Average" shows the average across the remaining (split/join, buffer) kernels. Note that the two coefficient replication kernels in the "Big/Fast" program show zero utilization because they only execute once at the beginning of the application to load the coefficients into the convolution kernels.

1) inherently low-utilization kernels such as splits, joins, and buffers that are each mapped to a dedicated processor, and 2) the round-robin distribution of work to parallelized computation kernels, which only achieves high utilization if the amount of work required is close to an integer multiple of the capabilities of an individual processor. The inherently low-utilization kernels are in general non-computation kernels. Taken by themselves, these "other" kernels ("Other Average" in Figure 8.2) have net utilizations of 7% and 14% for the two programs, respectively, which significantly reduces the average utilization. To ameliorate this problem, the kernels with low utilization should be time-multiplexed with other low-utilization kernels on a reduced set of processors. Doing so will result in fewer processors with a higher utilization per processor.

8.2 Greedy Merge Algorithm

To overcome the poor utilization demonstrated with 1:1 kernel-to-processor mappings, kernels should be time-multiplexed on the same processor. This effectively builds up a higher utilization for that processor and reduces the total number required. However, the choice of which kernels to time-multiplex is a fundamentally difficult problem as it is not independent of the choice of parallelization in the first place. For example, it may be better to parallelize a kernel more than is required to meet the real-time requirements in order to enable time-multiplexing it with another kernel, the combination of which may result in increased utilization and/or reduced communications.

The "greedy merge algorithm" presented here is a simple greedy algorithm for finding a reasonable set of kernels to time-multiplex given a *fixed* initial parallelization. This algorithm examines neighboring kernel groups in the application graph and merges the ones with the lowest utilization if their combined utilization does not exceed 100% of the processor CPU or memory.

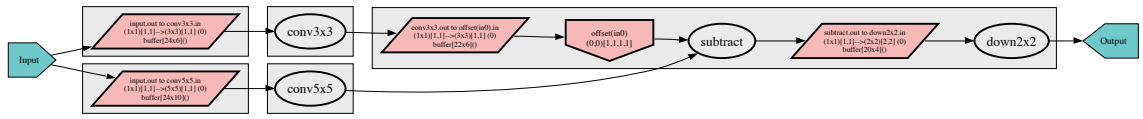
The greedy merge algorithm starts by building a set of 1:1 candidate kernel groups that map each kernel to its own group. It then loops over the candidate groups, starting with the group with the lowest utilization. For each candidate group, the

neighboring groups in the application graph are identified. The neighboring groups are then sorted in order of increasing CPU utilization, and the algorithm merges the candidate with the neighbor with the lowest CPU utilization whose combined grouping does not exceed either the CPU or memory limitations of the processor. If a valid merging is found, the old candidate and neighbor groups are removed from the set of candidates, and the new merged group is added. If a group's utilization exceeds a threshold (95% for these examples) of the CPU or memory it is removed from the set of candidates. If a group can not be merged with any of its neighbors, or has no neighbors, it is removed from the set of candidates as well. Buffers connected directly to application DataInputs are not considered for merging because they must immediately buffer the input data to insure the DataInput does not stall and drop data. Time-multiplexing these buffers can result in other kernels being executed on that processor when a DataInput generates data, which causes the input data to be dropped. The algorithm terminates when there are no more candidate/neighbor merge pairs to consider. The result of applying this algorithm to the programs in Figure 8.1 are shown in Figure 8.3.

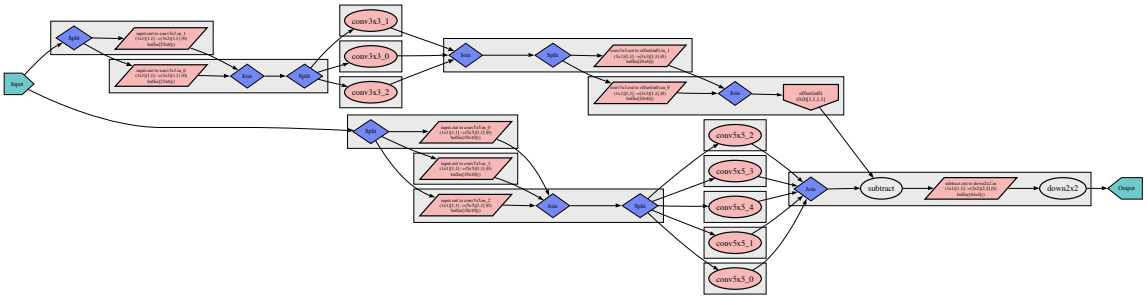
8.3 Results

8.3.1 Greedy Mapping Results

The results of applying the greedy merge algorithm are shown in Figure 8.3. The corresponding utilizations for these programs are shown in in Figure 8.4. The utilization for the "Small/Slow" and "Big/Fast" programs increased from 20% to 37% and 32% to 63%, with the greedy merge algorithm, respectively. Both of these applications have their utilization limited by only considering neighboring kernel groups to merge. The "Small/Slow" program has so few kernels to start with, so the two non-merged input buffers significantly decrease the average utilization. The "Big/Fast" program is limited by the inability to merge kernel groups across the convolution kernels due to their high inherent utilization. In this case, choosing a different initial parallelization which reduced the utilization for the convolution kernels might have allowed



(a) "Small/Slow" program from Figure 7.8



(b) "Big/Fast" program from Figure 7.11

Figure 8.3: Greedy kernel mapping
The results of the greedy kernel mapping algorithm when applied to the programs from Figure 8.1 are shown. Coefficient inputs are not shown for simplicity.

the time-multiplexing to merge the convolution kernels. Alternatively allowing non-neighboring kernel groups to merge or changing the round-robin data distribution would have improved utilization.

8.3.2 General Results

The results of applying the program dataflow analysis, inset and buffer insertion, parallelization, and greedy time-multiplexing discussed in this thesis are shown for a range of programs in Figure 8.5. The new programs introduced here, "Parallel Buffer" and "ConvAB", are shown in Figure 8.6 along with the details of the time-multiplexing for the "Bayer" and "Histogram" programs. The "Parallel Buffer" program tests the buffer kernel parallelization for improved data reuse that was discussed in Section 7.6.1. "ConvAB" is a test case for the Split/Join distribution mentioned in Section 7.6.2. The "Bayer Fast" and "Histogram Fast" programs are versions of the "Bayer" and "Histogram" programs, respectively, where the input rate has been roughly doubled.

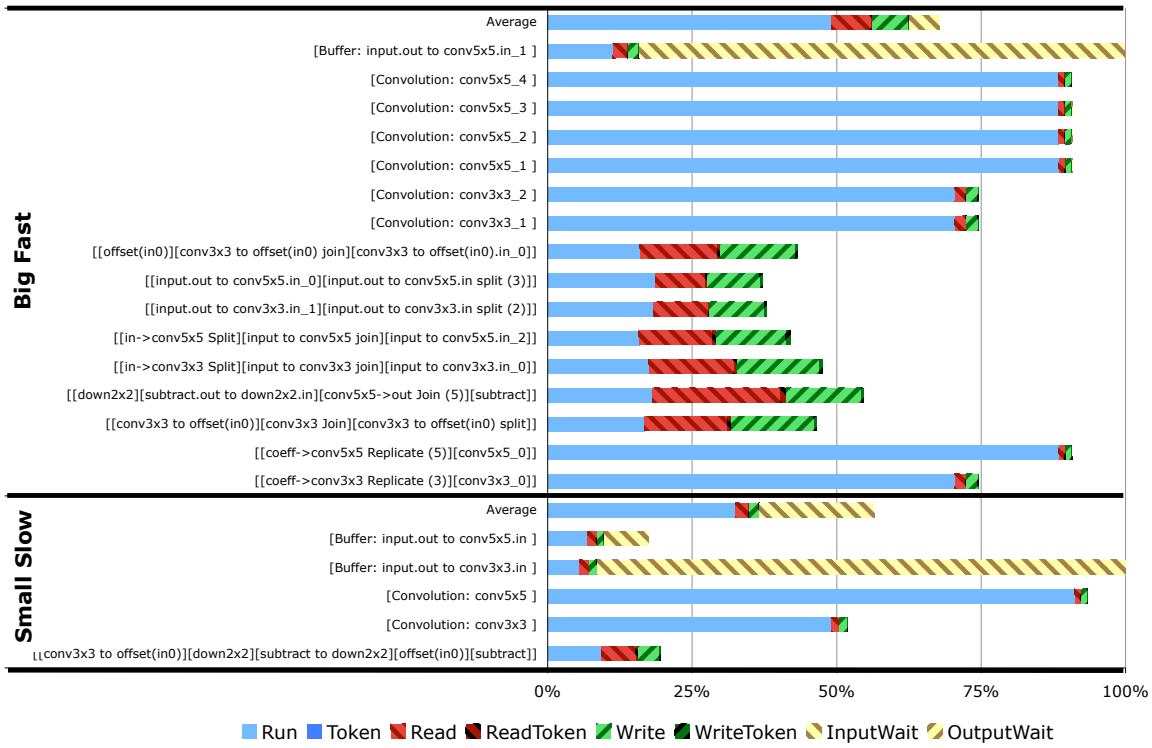


Figure 8.4: Greedy mapping utilization

On average, the time-multiplexing selected by the greedy merging algorithm resulted in a 1.5× improvement in processor utilization, including the overhead for cooperative scheduling of the multiplexed kernels. This improvement was greatest for the "Big/Slow" program (2.1×) and smallest for the "Histogram Fast" program (1.1×). These results correlate closely to prevalence of high-utilization computation kernels to low-utilization kernels in the applications. The "Big/Slow" program (Figure 7.10) has only four computation kernels (two of which, "subtract" and "down2x2", do very little), but eight buffers and six split/join kernels. By time-multiplexing these together it is able to reduce the number of processors required from 19 to nine. Conversely, the "Histogram Fast" program only saves one of its nine processors when time-multiplexed, but because four of the original nine were already running high-utilization computation kernels, it still maintains a high utilization. Given this high sensitivity to the constitution of the program, the average improvement shown here

should be quoted with care.

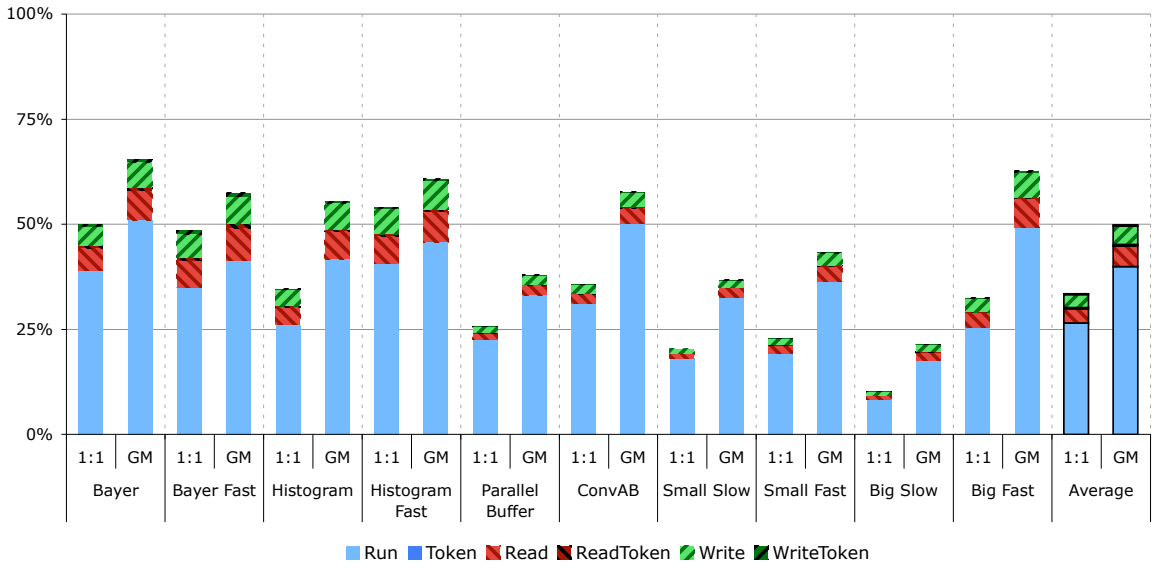
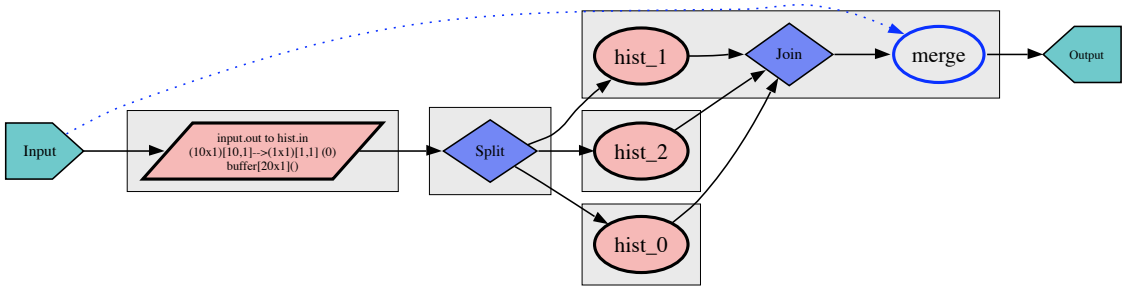


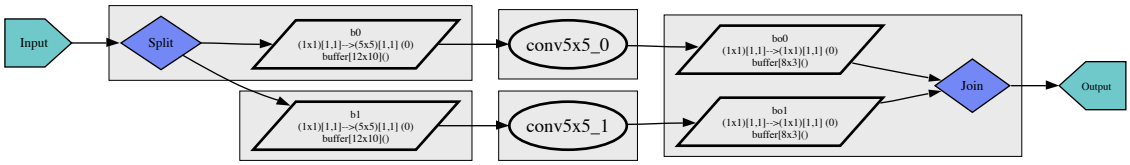
Figure 8.5: Average utilization for naïve (1:1) and greedy (GM) mappings

The average utilization is shown across a variety of test programs for the 1:1 kernel-to-processor mapping and the greedy merge.

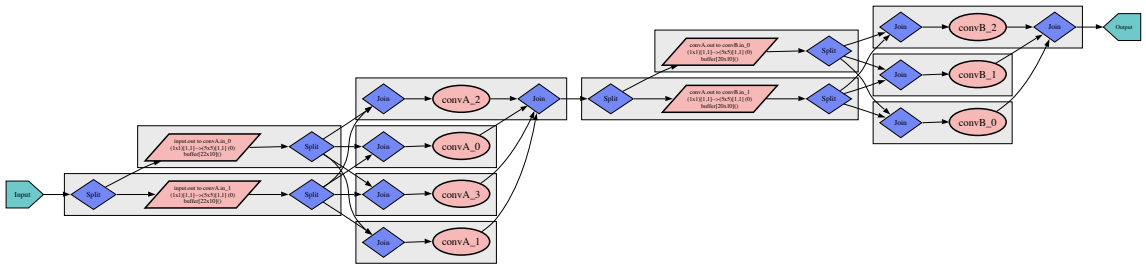
However, Figure 8.5 is most significant because it validates the analyses and transformations presented in this thesis. By showing these results across a range of programs, it can be seen that this approach is general enough to handle a range of programs and can automatically adjust to varying input constraints. The programs shown here range in size from only 11 kernels in the "Histogram" program to more than 50 in "Bayer Fast", demonstrating that the approach scales to many tens of processors. The correctness of the "Histogram" application demonstrates the usefulness and capabilities of the data dependency edges and ControlTokens discussed in Chapter 4. The "fast" versions of the test programs and the four "Small/BigSlow/Fast" test programs all demonstrate the ability of this system to automatically adapt the program to changes in input rate requirements, by appropriately propagating information through the application and the parallelization.



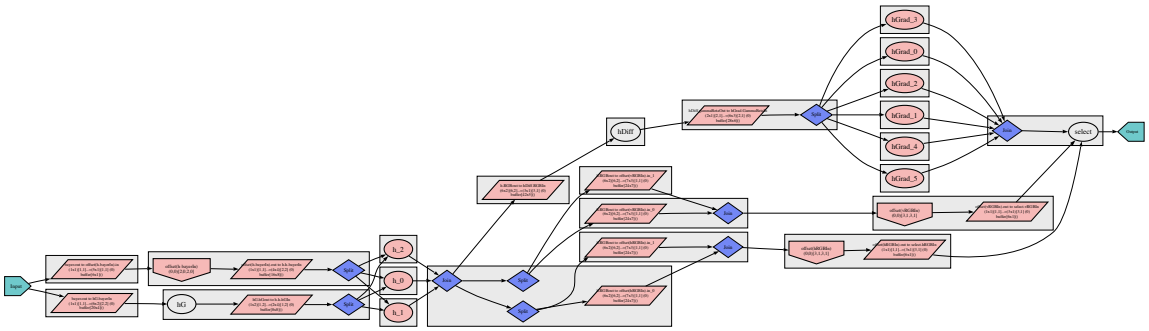
(a) "Histogram" program greedy mapping



(b) "Parallel Buffer" program greedy mapping



(c) "ConvAB" program greedy mapping



(d) "Bayer" program from Figure 7.7(b)

Figure 8.6: More greedy kernel mappings
These mappings correspond to the remaining applications in Figure 8.5.

8.4 Discussion

The greedy merge algorithm presented here implements a simple heuristic (merge the two kernel groups with the lowest utilization first) to select kernels to time-multiplex. Despite this simplicity, it provides a significant improvement in utilization over a simple 1:1 kernel-to-processor mapping. The resulting low overall utilization, however, is partially a result of the simplicity of the algorithm. In particular, the inability of the greedy merge algorithm to change the initial parallelization based on the time-multiplexing limits its ability to explore the potential mapping space.

The problem of time-multiplexing kernels on processors is a fundamentally hard one that has been explored in detail in the Raw backend for StreamIt [15, 16]. The StreamIt compiler takes a more sophisticated approach by trying to determine the correct degree of parallelism to enable the right amount of kernel fission to achieve the highest possible utilization. This differs from the problem here in that the minimum degree of parallelism is not dictated by the real-time constraints. However, StreamIt's approach of integrating the decision as to how much a kernel should be parallelized with the decision as to with which other kernels it should be time-multiplexed provides the benefit that more possible mappings can be explored. The algorithm presented here does not reevaluate the initial parallelization while determining the time-multiplexing.

The concept of kernel merging is similar to time-multiplexing and has been explored in both ArrayOL [4] and StreamIt. When merging kernels the functionality of the kernels is combined into one larger kernel, with the time-multiplexing of the different kernel operations implicit in the static merging of their code. Kernel merging has the benefit of removing the need to cooperatively schedule time-multiplexed kernels, but it has the downside that the buffering between the kernels must be statically determined as well. For example, merging kernels with 3×3 and 5×5 inputs would require determining the number of iterations of both kernels that consume and produce the same amount of input, and statically replicating the kernels appropriately to meet them. The result would be a single kernel that would execute on each

iteration some number of both sub-kernels in a static manner. For the more complex input and output patterns allowed by ArrayOL, finding such mergings has been shown to be decidedly non-trivial. StreamIt fusses kernels in this manner whenever it can profitably do so and statically bound the buffer size required between them.

In addition to fusing kernels, StreamIt took kernel merging one step further by implementing a linear state space analysis of the kernels to determine if their internal computations can be profitably merged [1]. Such analysis is only applicable to kernels of a particular form, but when it can be applied it enables merging kernels in the most efficient manner possible.

The Greedy Merge Algorithm

The greedy merge algorithm presented here is greedy in the sense that it takes the first valid choice each time. This simplifies implementation and provides for a very fast runtime, but is quite likely to get caught in local minima. The implementation of the algorithm relies on accurate static estimates[3] of the CPU and memory utilization of each kernel, and the monotonic behavior of merged kernels. (E.g., if the CPU or memory utilization of two merged kernels decreased in merging them this greedy approach would be less likely to find the best match without examining all of its neighbors.)

This algorithm has two significant limitations that appear in the results presented here. The first is the limitation of only considering neighboring kernel groups for merging. This prevents merges across kernels with high utilization such as the convolution kernels in the "Big/Fast" and "Small/Slow" programs. The second limitation is in not changing the initial parallelization to make the time-multiplexing more efficient. This is particularly a problem for the parallelization of kernels where the even round-robin distribution of work results in a lower utilization across the computation kernels. This effect can be seen in the "Big/Fast" application. The "conv3x3" kernel requires 2.25 processors to meet its required rate. This forces the parallelization to allocate 3 processors to ensure the real-time requirements are met, which reduces

[3]The estimates provided by the compiler for runtimes appear to be accurate to within 5% of the actual runtimes

the average utilization to only 75% with a round-robin distribution of work. If a non-round-robin distribution were chosen that sent 44% of the data to the first two parallelized kernels, the third kernel would have a utilization of only 13%, which might make it a much better candidate for time-multiplexing with other kernels.

The algorithm further avoids time-multiplexing buffers that immediately follow a DataInput. If such "inflexible" buffers are time-multiplexed, the cooperative nature of the scheduling can prevent them from running for long enough to cause the application to miss an input, and thereby not meet its real-time requirement. This can significantly reduce the average utilization for programs with few kernels, such as the "Small/Slow" program. Indeed, Figure 8.3(a) suggests that if the initial buffers were not explicitly excluded from merging, the greedy merge algorithm would have merged one of them with the "conv3x3" kernel, thereby removing the processor with the lowest utilization.

To make this approach more accurate it would be important to include the benefit of placing two kernels on the same processor. This would be seen through the reduction in communications costs between two processors, and is not visible in a non-placed simulation. However, because this algorithm only considers merging neighboring groups in the application, each merging will reduce the net communications between processors and replace it with a local memory transfer or pointer exchange.

Chapter 9

Conclusions

As architectures with tens to hundreds of processor cores become more common, programming approaches will need to automatically handle the parallelization of computation across the cores and, more critically, the streaming of data through them in order to achieve decent utilization. To this end, this thesis has introduced a statically parameterized streaming language and compiler framework for analyzing and mapping real-time embedded applications to many-core architectures, and demonstrated the correctness and effectiveness of this approach with a variety of test programs.

The language presented here implements a variety of features to improve the flexibility of stream programming, thereby making it easier to write and analyze such programs. The three major additions are: 1) the use of two-dimensional data to make image processing easier and enable the automatic analysis and buffering of two-dimensional data, 2) the use of control tokens to integrate control flow within a streaming model in a manner that is both flexible and analyzable, and 3) the addition of data dependency edges to explicitly specify the degree of parallelism allowed, thereby enabling the merging of serial and parallel computations without compromising the ability of the compiler to analyze the program. Taken together, these features make for a more intuitive and powerful programming model than previous streaming languages.

The compiler framework utilizes the static parameterization of the application

data movement and resource requirements to enable powerful automatic manipulations. These include data insetting or zero-padding to correct intuitive, but inconsistent, application descriptions, insertion and sizing of two-dimensional circular buffers to interface kernels with different data shape requirements, and parallelization with appropriate data distribution and collection to meet real-time requirements. These transformations are enabled by a series of dataflow analyses which propagate the application's input size and rate through the application to determine the required processing rate of each kernel.

This complete process is shown in Figures 9.1 through 9.8. The simplified application description is shown in Figure 9.1 with the details of the parameterized inputs and outputs shown in Figure 9.2. The dataflow analysis for data insetting and buffering is shown in Figure 9.3, with the results from automatically inserting buffers and insets displayed in Figure 9.4. The fully valid application after inset and buffer insertion is shown with a complete dataflow analysis in Figure 9.5. This analysis is then used to parallelize the application to meet the required input rates, as seen in Figure 9.6. The kernels from the parallelized application are then grouped for time-multiplexing to increase utilization (Figure 9.7), and the results from the final cycle-accurate simulation are shown in Figure 9.8.

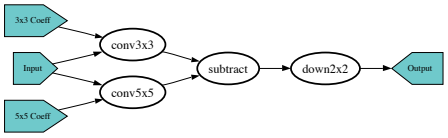


Figure 9.1: Input: Simple program representation

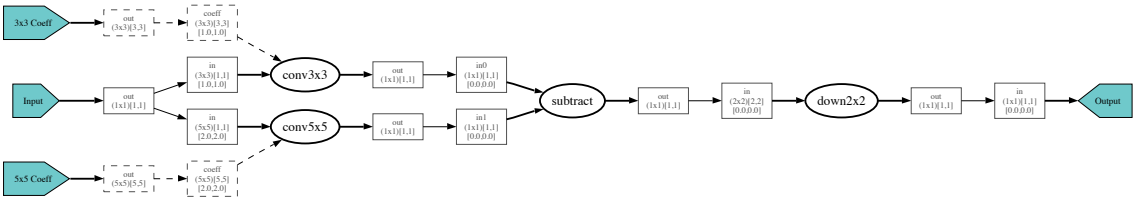


Figure 9.2: Input: Full parameterized program representation

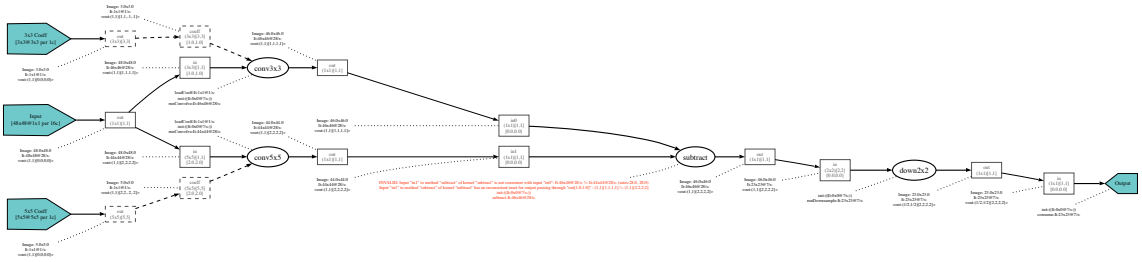


Figure 9.3: Step 1: Partial dataflow analysis for inset/buffer insertion

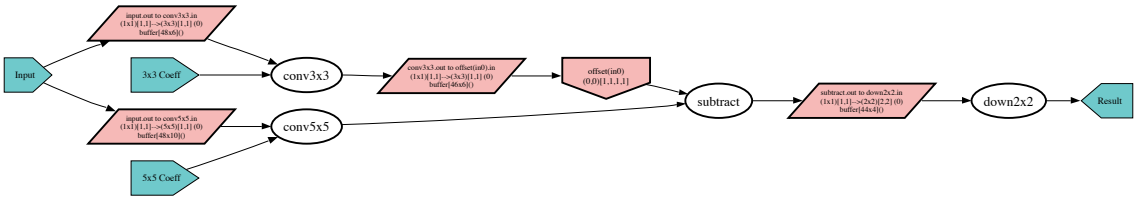


Figure 9.4: Step 2: Automatic insertion of buffers and insets for correctness

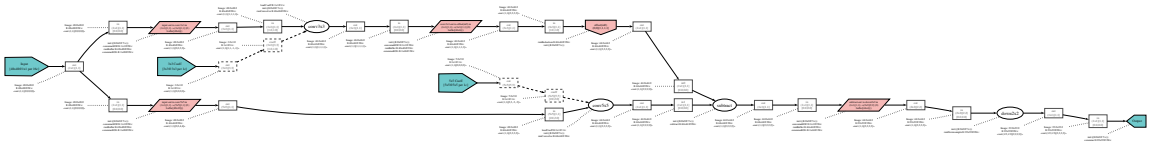


Figure 9.5: Step 3: Dataflow analysis for automatic parallelization

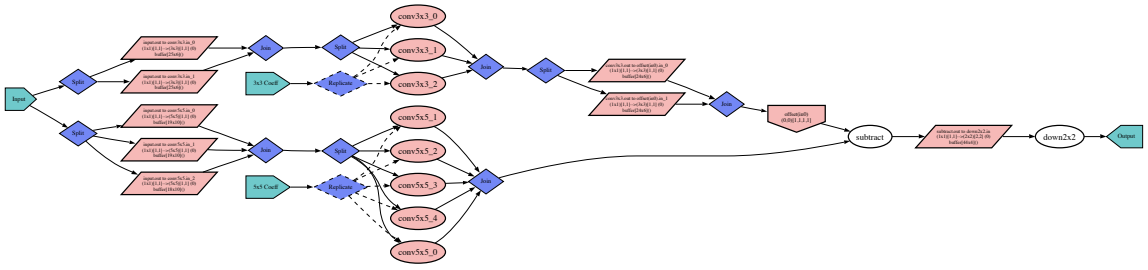


Figure 9.6: Step 4: Automatic parallelization to meet real-time constraints

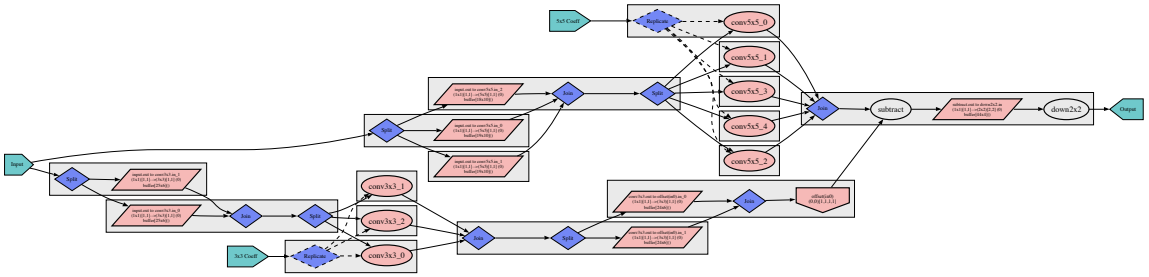


Figure 9.7: Step 5: Automatic time-multiplexing to increase utilization

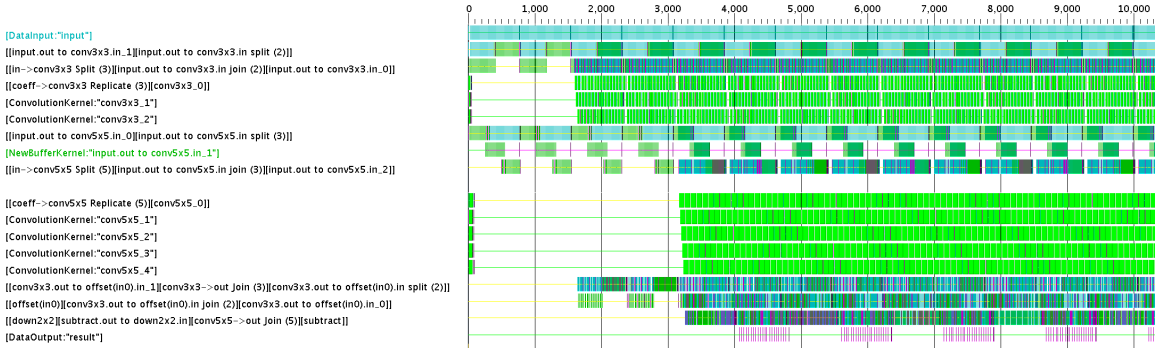


Figure 9.8: Step 6: Simulated application execution

The data movement model presented here simplifies application analysis and manipulation by assuming a scan line ordering of both incoming data and execution and two-dimensional windowed reuse patterns. This approach works well for data-parallel, and therefore order-agnostic, image processing algorithms. For algorithms that require arbitrary scatter/gather data access, column ordering, or higher-dimensional data, this parameterization and its associated analyses are of limited value. In these cases, the programmer must configure the data movement and parallelization, but such applications can still be implemented within the provided framework. This can be done either manually be instantiating appropriate kernels or automatically by writing a parallelization algorithm which the compiler can then use. Either way, the framework is capable of incorporating, and subsequently placing, kernels that do not

adhere to the basic scan line ordering of the system, but it is unable to analyze and manipulate them for automatic parallelization.

However, the Achilles heel of this work is its reliance on a static parameterization of an application's processing rates and sizes. While this enables straightforward calculation of the required computation rates and buffer sizes in the application, it limits the applicability of the language to less regular applications. The general problem of resource allocation to meet real-time resources constraints under variable workloads is non-trivial, and fundamentally requires either provisioning resources for the worst case or provisioning for some statistical guarantee with explicit exception handling. This work takes the former approach as all rate information is static. To enable provisioning for a statistical guarantee, the system must be able to handle the infrequent cases where the computation or storage requirements exceed those allocated. Such event handling requires that the runtime system generate appropriate exceptions and that the programmer handle them as needed. The addition of this statistical approach to this framework would significantly increase the system's applicability, but reduce its analyzability.

In conclusion, the language presented here is a practical and flexible approach to enabling the automatic mapping of applications to many-cored architectures. The success of this approach stems from exposing the static requirements of each kernel to the compiler in a parameterized method that enables straightforward analysis of the data movements and computation requirements, and using an underlying data model that imposes a simple processing order. Structuring the programming system in this manner avoids the difficulty of extracting the data movement and computation kernels from arbitrary code (e.g., the analysis of imperative code) and the complexity of having to determine the correct ordering as well (e.g., finding the optimal transformations to apply). By simplifying these two tasks, the problem of efficiently manipulating and mapping an application can be more readily addressed as demonstrated in this work.

Appendix A

Placement

Once the application is parallelized, placement on the processor array can be achieved by using simulated annealing [29] with an appropriate cost function that takes into account the communications cost for a given placement. This procedure can be applied to a variety of different time-multiplexings (e.g., mappings of multiple kernels to the same processor) to choose the end result with the lowest overall cost.

A.1 Simulated Annealing

Simulated annealing is the process of applying random changes to a system and accepting changes that produce worse system costs with a progressively decreasing likelihood. This approach attempts to avoid local minima by allowing the system sufficient flexibility to make "bad" choices at the beginning, which should allow it to move far enough to escape local minima. As the simulation progresses, fewer and fewer "bad" choices are allowed, thereby encouraging the system to settle down into a final form, and explore the remaining local optimizations.

A.2 Cost Function

To implement simulated annealing, a cost function and a perturbation function must be provided. For the problem of placing kernels (or groups of time-multiplexed kernels) on processor tiles, the perturbation function is simply randomly changing the placements by some amount. The cost function, however, is more complicated.

For this application, the cost function looks only at the communications as the computation is fixed by the particular parallelization provided to the placement module. The communications cost is determined by the application analysis which describes the amount of data sent over each edge in the application graph. From this, the cost of a given placement can be determined by computing how far the data must move given the physical location of each kernel, and the routing resources between its source and destination.

For this work, the cost function included two different communications networks on the processor array. A local communications path provided cheap access to nearest neighbors, while a global communications network was more expensive, but provided arbitrary connectivity. The global network's cost included an initial startup cost, and then a per-hop cost along the dimension-ordered route between the source and destination. Such a cost function encourages the placement of kernels with significant producer-consumer locality close to one another. The cost function could be further extended to model the effect of interfering traffic from other communications if the particularities of the routing on the processor array were sufficiently well known.

A.3 Results

The simulated annealing was implemented by using the Graphviz `dot` [17] graph layout program to produce an initial layout for a variety of time-multiplexings of kernels. These initial layouts were then annealed for a fixed amount of time. The results from this process are shown in Figure A.4, with the initial placements in Figure A.3. Figures A.1 and A.2 show the initial and post-annealing placements for

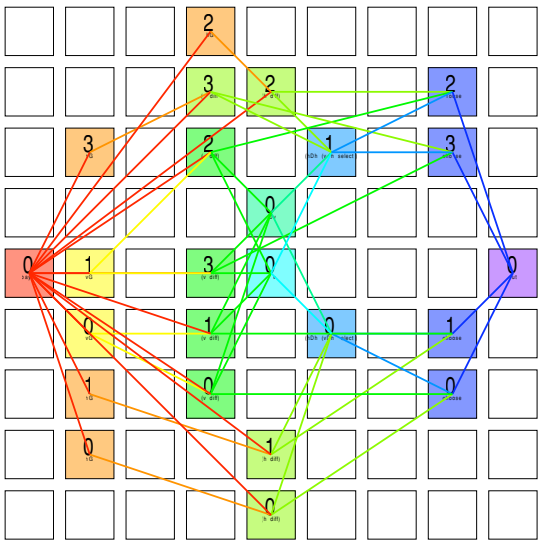


Figure A.1: Initial kernel placements for Bayer demosaicing before annealing

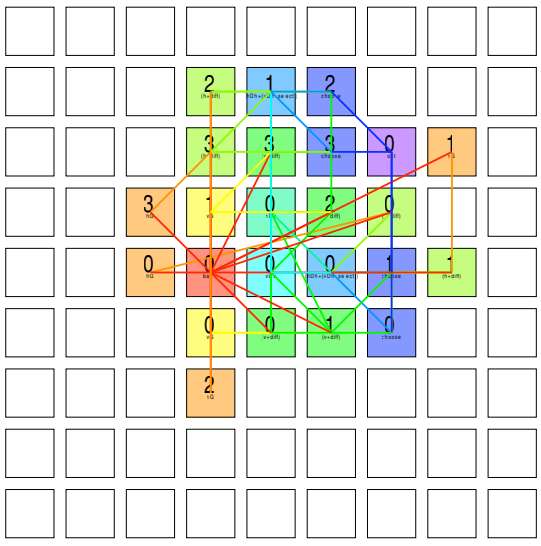


Figure A.2: Final kernel placements for Bayer demosaicing after annealing

the Bayer demosaicing application. Simulations were based on un-placed applications, as discussed in Appendix B.

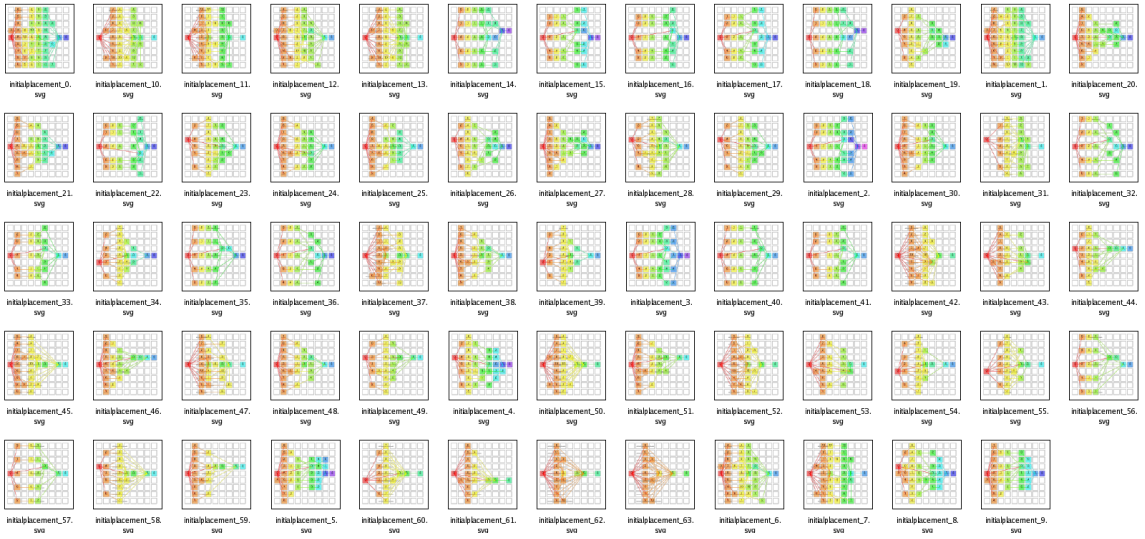


Figure A.3: Initial JPEG kernel placements before annealing

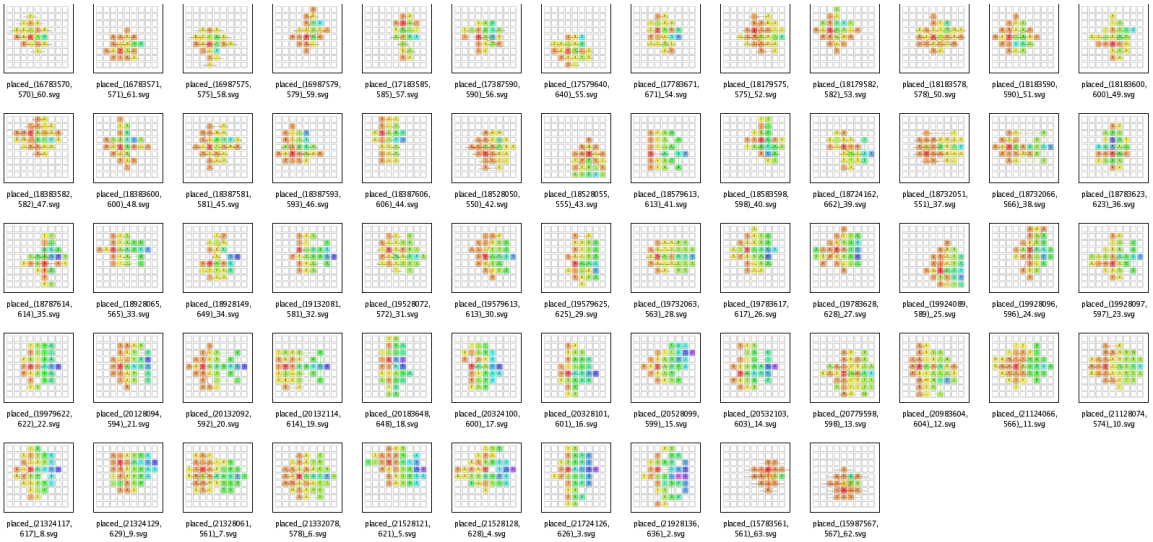


Figure A.4: JPEG kernel placements after annealing

Appendix B

Simulator Implementation

The simulator was designed to provide a functionally correct execution of the applications with as little effort as possible. By using the native reflection and thread capabilities of the language, it was possible to implement a "cycle-accurate" and highly flexible simulator without needing any hardware emulators or external compilers. This design incurred the tradeoff that performance is limited by the reflection capabilities of the language and the thread synchronization performance of the system.

B.1 Functional Simulation via Threads

The functional simulator is implemented by mapping each computation kernel (including Buffer kernels, DataInput, DataOutput, and Split/Join kernels) to their own thread. The run loop for each thread simply checks the inputs to determine if a valid combination of data or ControlTokens has arrived to fire a method, and, if so, it invokes the method. Methods are invoked using the built-in reflection facilities of the underlying language. This enables kernels to be written specifying the name of the method to execute as a string, which the runtime can then use to lookup the corresponding method and execute it. By specifying methods so, their description can further be used for the data analysis which operates on the same description used by the runtime system.

The Inputs and Outputs for kernels are implemented with blocking queues, which cause the thread attempting to read or write them to block, if they are empty or full, respectively, until the queue is appropriately serviced. The thread's run method detects if a ControlToken has been received but not processed and handles moving it to the method's Outputs.

This approach places the burden of tracking the internal kernel state on the underlying OS thread implementation, thereby making it trivial for a kernel to block trying to write an output without halting the simulation. While this greatly simplifies writing the simulator, it incurs the enormous performance penalty of relying on system/library thread synchronization primitives to orchestrate the execution.

By itself this implementation is functionally correct. Applications run "freely", with kernels blocking when they have no available data, and stalling when the downstream kernels are not ready. Applications that do not deadlock will execute correctly in this model, but applications that do deadlock may exhibit inconsistent, and difficult to debug, behavior. This is due to the "functional" nature of the simulation wherein each kernel's execution time and order depends on the system's scheduling of its thread. This invariably leads to different execution orders on each run, which complicates debugging.

B.2 "Cycle-accurate" Simulation

The functional thread-based simulation approach is easy to implement and can demonstrate the correctness of an application. However, it is inadequate for debugging as it does not produce reproducible execution orders, and does not provide accurate utilization information as each thread is scheduled and executed arbitrarily by the system. To address both of these issues, the simulator was made to be "cycle-accurate" by adding a single global barrier that synchronizes each kernel thread on each virtual clock cycle.

By using a global clock barrier, the simulator can ensure that the program executes consistently as a fixed set of operations will occur between each clock, regardless of how the operating system schedules the individual threads. Further, by modifying

the run loop of each kernel to specify that its thread should wait for a given number of cycles after each method invocation, methods can effectively simulating "running" for arbitrary lengths of time. However, some work must be done to ensure this illusion. When a method executes it fires on a given cycle, but effectively executes instantly as the global clock can not advance until the method returns. This means that any outputs that are written are written in that initial cycle, even if the method is scheduled to take multiple cycles. To ensure correct operation, the Inputs and Outputs are modified such that they will block if the data written into them is not valid at the time it is accessed.

This approach enables a "cycle-accurate" simulation, where the cycle count for each operation can be programmatically defined by the simulator. In addition to reproducibility, this provides significant flexibility over a micro-architectural simulator in that any aspect of a kernel can be set to take zero cycles to analyze its effect on the application performance. However, this approach does require that the kernel and simulator programmers enter appropriate values for all operations to obtain reasonable results.

B.3 Enabling Time-multiplexing

Adding the ability to simulate multiple kernels running in the same execution context requires that each kernel's thread run method be augmented to yield control to a cooperative scheduler after each method invocation. This was accomplished by defining a cooperative thread scheduler which uses a synchronization lock to cause the time-multiplexed threads to wait until the currently executing one yields. At that point it then enables the next appropriate thread and allows it to continue. Between each thread switch, the cooperative thread scheduler waits for the global clock barrier to ensure consistent execution.

B.4 Parameters

The simulator used for this work makes several assumptions about data transport times. The most significant is that data transfers between Inputs and Outputs are assumed to take zero cycles. This does not accurately reflect the performance of any realistic communications architecture, but provides a good abstraction for analyzing the raw performance of an application independent of its physical placement. The most significant effect that is hidden by this assumption is the centralized nature of the Split/Join kernels may saturate the available communications bandwidth. In order to more accurately estimate the communications latency, a full placement for the application and a detailed communications architecture description would be required.

The simulator further models the time to move data from Input and Output buffers as taking only one cycle. This is reasonable if the buffers are local to the processors and the transfer is merely a pointer change rather than a buffer copy.

Cooperative scheduling for time-multiplexing kernels on the same processor is modeled to take one cycle per scheduling iteration, regardless of whether a kernel method is invoked. For back-to-back invocation, this incurs an overhead of one cycle per multiplexed kernel, which is not unrealistic for a hardware-assisted implementation.

B.5 Application Correctness

While not technically part of the simulator, the verification of the application's correctness is an essential part of the simulator. The correctness is verified automatically in through comparison of outputs to known correct values and automatic detection of input stalls. For the majority of the test applications presented here, MATLAB simulations were run to compute the expected final and intermediate outputs for the application. These outputs were verified by inserting additional DataOutputVerifiers which compare their received outputs to the results calculated by the MATLAB simulations. (See Figure B.1.) Additional runtime checks are made to verify that the DataInputs to the application are always able to send their data to the downstream

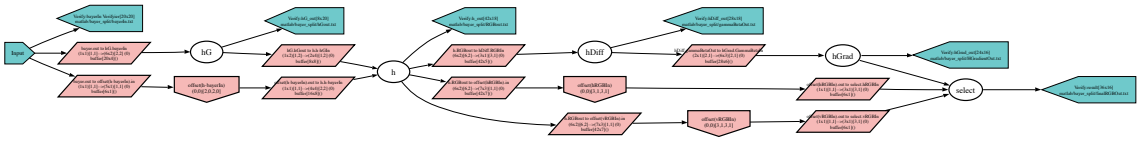


Figure B.1: Bayer application with output verification

kernels on-time. This ensures that the input data is never stalled, which, in the application's steady-state, indicates that it is meeting its real-time requirements.

B.6 Simulation Traces

The output of the simulator is viewed through a trace viewer application (Figure B.2) which displays the execution state for each processor simulated. For time-multiplexed applications, multiple kernels show up on a given line, but for non-time-multiplexed kernels each line represents a separate kernel. The traces display the state of the processor in a color-coded fashion as described in Figure B.3.

Figure B.2: Simulation timeline viewer application

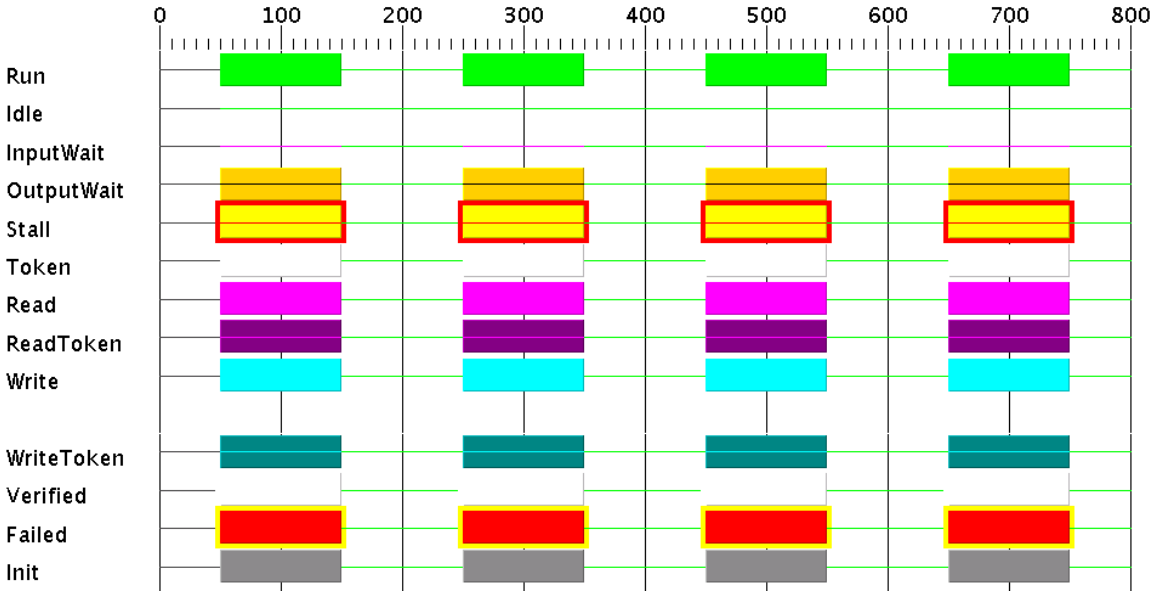


Figure B.3: Simulation timeline key

Run (green bar) indicates a kernel is executing. Idle (green line) indicates the processor is not running anything. InputWait (red line) and OutputWait (yellow bar with black line) indicate a kernel is waiting to either read from its input or write to its output. These occur when the Input buffer is empty or the Output buffer is full. Read (magenta bar) and write (cyan bar) indicate time spent accessing the Input and Output buffers, respectively. ReadToken (dark magenta bar) and WriteToken (dark cyan bar) indicate time spent reading and writing ControlTokens, respectively. Verified (white bar) and Failed (red bar with yellow edges) indicate the completion status of an application when simulated with DataOutputs that compare the results to known values.

Appendix C

Future Work

C.1 Variable Rates and Sizes

One of the most severe limitations of the block-parallel approach presented here, and SDF-like approaches in general, is their insistence on statically known rates and data sizes. While such data does enable simple and accurate compiler analyses, it also limits the general applicability of the programming system. This limitation is particularly apparent in applications that deal with variable data sizes (such as variable numbers of motion vectors in video compression) or variable computation rates (such as different runtimes for the encodings of different symbols). The issue of variable rates and sizes comes into play in two areas: static application analysis and dynamic resource allocation.

Static Application Analysis with Variable Rates and Sizes

Implementing static application analysis with variable rates and sizes involves replacing the known distinct size and rate values used in the data analysis with a representation of their variable nature. For example, the computation cycles required for a kernel method might be replaced with a distribution representing the frequency with which the kernel takes a given number of cycles. Similarly, the output size for an encoding kernel might be represented with a distribution that tells the frequency with

which a given number of outputs are generated. These distributions could be simple averages, parameterized Gaussians, or discrete distributions acquired from profile analysis. The application analysis would then need to convolve these distributions to propagate the information through the application, and the calculation of the required resources at a given point would then need to be done by integrating along the distribution to the desired level of certainty. For example, if the programmer requests 95% assurance of meeting the real-time constraints, the appropriate distribution would be integrated to 95%, and that value would be used for determining the amount of resources to allocate.

In addition to calculating the required resources, the runtime system would need to provide a means to generate and process exceptions for the (hopefully rare) occasions when the allocated computation or storage resources are insufficient. Handling these exceptions can be difficult as the exceptions themselves require resources, and it is likely that the appropriate action can not be taken at the level of the individual kernels, but must instead be taken at a higher level in the application. Adding these capabilities to the system presented here would complicate the analysis (as it is no longer static), and require explicit out-of-band (and hence unanalyzable) exception handling, but would enable a much broader range of applications.

Dynamic Resource Allocation

The other side to variable computation rates and data sizes is the desire to dynamically allocate resources to meet the dynamic resource demands. As a given number of resources must be allocated to meet the real-time constraints in the first place, the utility of this capability is largely due to its ability to take advantage of very-low power states which may take a long time to enter and exit, and not its potential to reduce the absolute resource requirements. If the runtime can determine that the current dynamic load will not require the use of some resources for an extended period of time it can afford the time to suspend them and resume them as needed.

C.2 Phased Computation

Many application workloads consist of distinct phases which are dynamically determined, but statically bound. Such an example would be a color copier which analyses each page it processes and applies different image processing techniques based on the content of a page. While the application would spend a fixed amount of processing resources per image, the type of processing done would vary dynamically. To efficiently integrate this type of processing into the system presented here, it would be necessary to enable the programmer to specify which segments of an application were dynamically executed, and how their resources are interrelated. The compiler system could then analyze these segments and determine the resources required for the worst-case without over-provisioning for segments which would never need to execute concurrently. The problem of mapping a program with such dynamic execution styles could be tackled by developing a range of static mappings for each of the expected dynamic workloads and then choosing the correct one at runtime, or by using a dynamic runtime environment that swaps in-and-out pre-determined application segments as needed.

C.3 Dynamic Data Fetch

The ability to gather (and scatter) dynamic data is critical for many applications. The canonical image processing example is a motion vector search where each motion vector determines the location in the current and previous frame to be fetched for the beginning of the search. Unfortunately, data-dependent fetches, by their nature, are almost always unanalyzable by a static compiler. The best approach for dealing with such references is to use a hardware or software data cache, and to try and buffer enough requests to cover the memory fetch latency with computation.

So while such references are not likely to be analyzable, they can certainly be integrated into the type of system described here. For example, a motion vector search might contain an image fetch kernel which would take in the motion vector to be processed and would then access the memory containing the image data. The kernel's

output would be the vector and the search image data, which could then be distributed to multiple search kernels to execute the search. This would enable the motion vector search to fit into a block-parallel framework quite cleanly, as the compiler would simply be unaware of the unanalyzable data transfers. (When mapping the kernels to processors, however, it would be important to take this data transfer into account.) Such an approach could be extended by allowing the image fetch kernel to choose where to send each motion vector so as to maximize the data reuse within the search kernels, with minimal modification to the analyses presented here.

C.4 Merging Buffers and Kernels

The differencing program example discussed *ad nauseum* in previous chapters (Figures 4.3 and 6.1), is an excellent example of an application which can be dramatically improved by the merging of buffers and kernels. In this case, all three of the computation kernels are affine functions of the input data, which means they could be reduced to one kernel, and have all internal buffering eliminated. Furthermore, the two buffers required to supply the 3×3 and 5×5 convolutions from the 1×1 DataInput are clearly redundant: the 5 lines of input data held by the buffer for the 5×5 convolution kernel clearly subsumes the 3 lines in the buffer for the 3×3 kernel.

Merging the buffers is fairly straightforward. The result will be a buffer that is the larger of the two, modulo any differences in input or output step sizes, which is just a logical extension of the buffer size calculations done in Section 6.1.1. The buffer implementation becomes more complex as each buffer must now ensure that all relevant outputs are done with a given piece of data before determining that it is stale.

Merging of kernels, however, is significantly more difficult. At its simplest, merging kernels aims to maximize data reuse. For example, merging a 5×5 convolution kernel with a 3×3 convolution kernel would result in a kernel that output one 5×5 result and three 3×3 results on each iteration. In addition, the kernel would have special behavior at the beginning and end of each line as the horizontal halos differ for the two kernels. Enumerating the possible ways to merge such kernels is not particularly

difficult, but doing so will change the rest of the application, making it difficult to evaluate the benefits of the merge independently. For purely linear kernels, a more sophisticated approach can be taken to merge their actual computation, as discussed in [2].

C.5 High-level Blocking

The explicit parameterization of the image size of DataInputs and the data reuse for all kernels should enable a fairly straightforward high-level blocking of input data into the available on-chip resources. For example, if an input image is large, a straightforward mapping of the application to a processor array may require more on-chip storage than is available. The compiler can readily re-map the application for a portion of the input until it determines that the application fits in the available resources. At that point the edge conditions for each kernel would have to be evaluated to determine how much data would need to be replicated or stored across each block, but because all of these values are explicit and parameterized in the application description, this analysis would be quite straight forward.

C.6 Higher Dimensional Data

The choice to use two-dimensional data for this language was a tradeoff between utility and complexity. As discussed in this work, mutli-dimensional data is accessed in one-dimensional streaming languages by requiring the programmer to manually map the higher-dimensional data into a one-dimensional stream. This manual mapping increases the complexity of the code and reduces the ability to analyze the use of the data because it is obfuscated by the manual mapping to and from a higher-dimensional data set. Conversely, providing a more general facility for accessing n-dimensional data natively incurs a tremendous overhead in terms of the complexity of the compiler analyses needed to interpret and manipulate the programs.

Two-dimensional streaming represents a "sweet spot" between the complexity of fully n-dimensional data and the limitations of one-dimensional data. This choice

makes sense given the large number of algorithms that operate on dense one- and two-dimensional data sets (virtually all signal and image processing) and the relative paucity of algorithms operating on dense higher-dimensional data sets. Indeed, most three-dimensional data sets are so large that they are represented using more complex structures, such as irregular grids or surfaces, which are not efficiently amenable to regular dense representation.

The complexity required to support higher dimensional data in this language would come primarily from the data flow analysis for propagating rates and sizes and the added overhead of keeping track of data usage in higher-dimensional circular buffers. Within the confines of maintaining a windowed data access parameterization (e.g., instead of the more general parameterizations supported by other languages) this complexity would be manageable, although the benefits might be limited. For example, the use of a third dimension to store RGB data for image processing adds very little over providing a separate input for each channel or manually multiplexing the RGB values in one frame for most uses. Providing an additional dimension to store previous frames for inter-frame encoding is again of marginal benefit over manually implementing such buffering in a dedicated kernel. The benefits of a native higher-dimensional data representation would be most significant when the compiler analysis could use the description to take advantage of reuse, or combine it with an automatic high-level blocking to deal with larger data sets.

Appendix D

Thesis Writing Progress

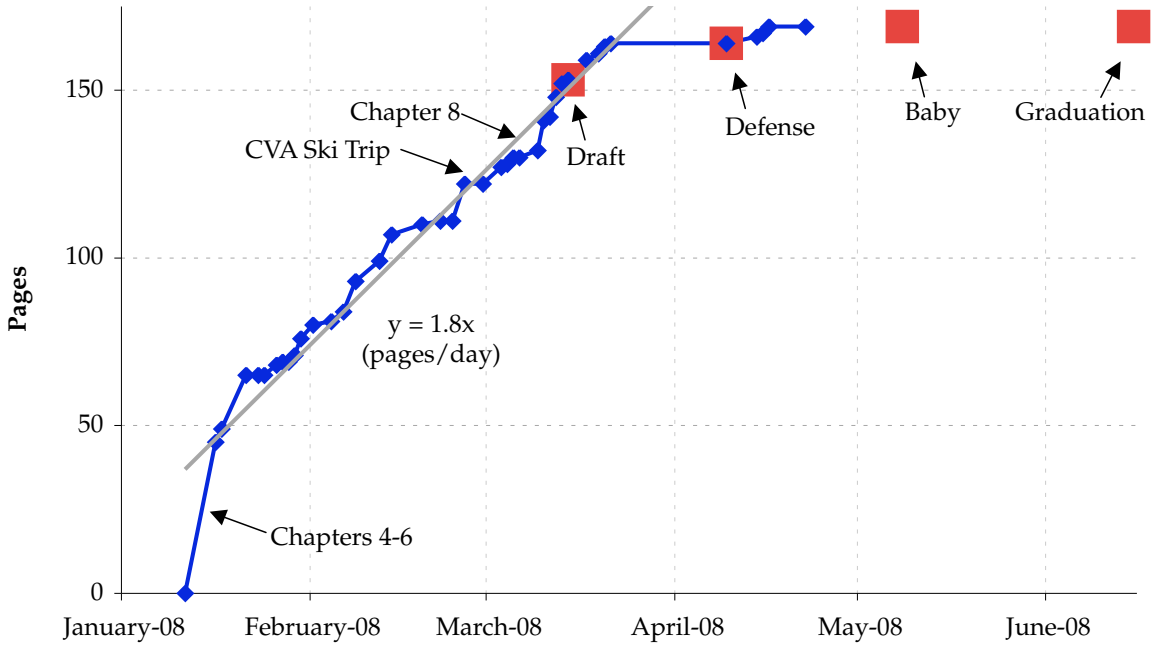


Figure D.1: Thesis writing progress
This graph is provided for entertainment only; no academic merit or suitability for any purpose is implied.

Bibliography

[1] S. Agrawal, W. Thies, and S. Amarasinghe, "Optimizing stream programs using linear state space analysis," in *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems.* New York, NY, USA: ACM, 2005, pp. 126–136.

[2] ——, "Optimizing stream programs using linear state space analysis," in *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems.* New York, NY, USA: ACM, 2005, pp. 126–136.

[3] A. Amar, P. Boulet, and P. Dumont, "Projection of the Array-OL specification language onto the kahn process network computation model," *Parallel Architectures,Algorithms and Networks, 2005. ISPAN 2005. Proceedings. 8th International Symposium on Parallel Architectures, Algorithms and Networks*, pp. 6 pp.–, 7-9 Dec. 2005.

[4] ——, "Projection of the Array-OL specification language onto the kahn process network computation model," INRIA, France, Research Report RR-5515, March 2005. [Online]. Available: http://www.inria.rf/rrrt/rr-5515.html

[5] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "Synthesis of embedded software from synchronous dataflow specifications," *J. VLSI Signal Process. Syst.*, vol. 21, no. 2, pp. 151–166, 1999.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, 2004.

[7] D. Carmean, "Intel Larrabee," May 2007, presentation at Stanford University, CS448.

[8] A. Das, W. J. Dally, and P. Mattson, "Compiling for stream processing," in *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques.* New York, NY, USA: ACM, 2006, pp. 33–42.

[9] E. Demers, "AMD R600," May 2007, presentation at Stanford University, CS448.

[10] M. Drake, H. Hoffmann, R. Rabbah, and S. Amarasinghe, "MPEG-2 decoding in a stream programming language," *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 10 pp.–, 25-29 April 2006.

[11] A. Duller, D. Towner, G. Panesar, A. Gray, and W. Robbins, "picoArray technology: the tool's story," *Design, Automation and Test in Europe, 2005. Proceedings*, pp. 106–111 Vol. 3, 7-11 March 2005.

[12] D. Dunn, "Azul plans 48-core processor for 2007," *Informationweek*, March 2006.

[13] M. Engels, G. Bilson, R. Lauwereins, and J. Peperstraete, "Cycle-static dataflow: model and implementation," *Signals, Systems and Computers, 1994. 1994 Conference Record of the Twenty-Eighth Asilomar Conference on*, vol. 1, pp. 503–507 vol.1, 31 Oct-2 Nov 1994.

[14] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[15] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *ASPLOS-XII: Proceedings*

of the 12th international conference on Architectural support for programming languages and operating systems. New York, NY, USA: ACM, 2006, pp. 151–162.

[16] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, "A stream compiler for communication-exposed architectures," in *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2002, pp. 291–303.

[17] Graphviz. (2008, March) Graphviz graph vizualization software. [Online]. Available: http://www.graphviz.org/

[18] J. Gummaraju, M. Erez, J. Coburn, M. Rosenblum, and W. J. Dally, "Architectural support for the stream execution model on general-purpose processors," *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pp. 3–12, 15-19 Sept. 2007.

[19] J. Gummaraju and M. Rosenblum, "Stream programming on general-purpose processors," *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, pp. 12 pp.–, 12-16 Nov. 2005.

[20] J. Gustafson, "Using accelerators to escape the shackles of 20th century software," in *International Supercomputing Conference*, June 2007.

[21] H. P. Hofstee, "Power efficient processor architecture and the cell processor," *HPCA*, vol. 00, pp. 258–262, 2005.

[22] Y. Hoskote, S. Vangal, N. Borkar, and S. Borkar, "Teraflop prototype processor with 80 cores," in *Hot Chips 19*, August 2007.

[23] M. Karczmarek, W. Thies, and S. Amarasinghe, "Phased scheduling of stream programs," in *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference*

on Language, compiler, and tool for embedded systems. New York, NY, USA: ACM, 2003, pp. 103–112.

[24] J. Keinert, C. Haubelt, and J. Teich, "Modeling and analysis of windowed synchronous algorithms," *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, vol. 3, pp. III–III, 14-19 May 2006.

[25] ——, "Simulative buffer analysis of local image processing algorithms described by windowed synchronous data flow," *Embedded Computer Systems: Architectures, Modeling and Simulation, 2007. IC-SAMOS 2007. International Conference on*, pp. 161–168, 16-19 July 2007.

[26] ——, "Windowed syncrhonous data flow (WSDF)," University of Erlangen-Nuremberg, Institute for Hardware-Software-Co-Design, Germany, Technical Report 02-2005, 2005.

[27] B. Khailany, W. Dally, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, A. Chang, and S. Rixner, "Imagine: media processing with streams," *Micro, IEEE*, vol. 21, no. 2, pp. 35–46, Mar/Apr 2001.

[28] B. Khailany, T. Williams, J. Lin, E. Long, M. Rygh, D. Tovey, and W. Daly, "A programmable 512 GOPS stream processor for signal, image, and video processing," *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pp. 272–602, 11-15 Feb. 2007.

[29] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science, Number 4598, 13 May 1983*, vol. 220, 4598, pp. 671–680, 1983. [Online]. Available: citeseer.ist.psu.edu/kirkpatrick83optimization.html

[30] T. J. Knight, J. Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan, "Compilation for explicitly managed memory hierarchies," in *Proceedings of the 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2007.

[31] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded sparc processor," *Micro, IEEE*, vol. 25, no. 2, pp. 21–29, March-April 2005.

[32] E. A. Lee, "Mulitdimensional streams rooted in dataflow," in *PACT '93: Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism.* Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1993, pp. 295–306.

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

[34] D. Luebke, "The democratization of parallel computing," November 2007, tutorial: High Performance Computing with CUDA.

[35] P. Mattson, "A programming system for the imagine media processor," Ph.D. dissertation, Stanford University, March 2002.

[36] P. Mattson, W. J. Dally, S. Rixner, U. J. Kapasi, and J. D. Owens, "Communication scheduling," *SIGARCH Comput. Archit. News*, vol. 28, no. 5, pp. 82–92, 2000.

[37] E. Murthy, P.K.; Lee, "Multidimensional synchronous dataflow," *Signal Processing, IEEE Transactions on [see also Acoustics, Speech, and Signal Processing, IEEE Transactions on]*, vol. 50, no. 8, pp. 2064–2079, Aug 2002.

[38] NVIDIA Inc. (2008, January). [Online]. Available: www.nvidia.com

[39] T. M. Parks, J. L. Pino, and E. A. Lee, "A comparison of synchronous and cycle-static dataflow," in *ASILOMAR '95: Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers (2-Volume Set).* Washington, DC, USA: IEEE Computer Society, 1995, p. 204.

[40] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens, "A bandwidth-efficient architecture for media processing,"

in *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998, pp. 3–13.

[41] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, pp. 491–541, 1997.

[42] M. Taylor, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, and J. Kim, "Evaluation of the raw microprocessor: an exposed-wire-delay architecture for ilp and streams," *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pp. 2–13, 19-23 June 2004.

[43] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *CC '02: Proceedings of the 11th International Conference on Compiler Construction*. London, UK: Springer-Verlag, 2002, pp. 179–196.

[44] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe, "Teleport messaging for distributed stream programs," in *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2005, pp. 224–235.

[45] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to software: Raw machines," *Computer*, vol. 30, no. 9, pp. 86–93, Sep 1997.

[46] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *Micro, IEEE*, vol. 27, no. 5, pp. 15–31, Sept.-Oct. 2007.

[47] R. Wilson, "Cisco taps processor array architecture for NPU," *EE Times*, September 2004. [Online]. Available: http://www.eetimes.com/showArticle.jhtml?articleID=26806315

[48] X. D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe, "A lightweight streaming layer for multicore execution," in *Workshop on Design, Architecture and Simulation of Chip Multi-Processors*, Chicago, IL, Dec. 2007. [Online]. Available: http://cag.lcs.mit.edu/commit/papers/07/zhang-dascmp07.pdf