8-16-2024

Automated Data-Flow Optimization for Digital Signal Processors

Madushan Thilina Abeysinghe
University of South Carolina

Automated data-flow optimization for Digital Signal Processors

by

Madushan Thilina Abeysinghe

Bachelor of Science in Engineering
University of Moratuwa, Sri Lanka, 2011

Bachelor of Science
University of Greenwich, United Kingdom, 2011

Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy in

Computer Science and Engineering

College of Engineering and Computing

University of South Carolina

2024

Accepted by:

Jason D. Bakos, Major Professor

Marco Valtorta, Committee Member

Yan Tong, Committee Member

Ramtin Zand, Committee Member

Austin Downey, Committee Member

Ann Vail, Dean of the Graduate School

Dedication

To my beloved parents, Nirmala Pathirana and Upali Abeysinghe.

Acknowledgments

If there's a Research Scientist that I look up to, it is my PhD advisor, Dr. Jason D. Bakos, who had the most profound impact on my career as a Computer Scientist. I thank him for supporting me via financial assistance and academic advice throughout my ups and downs in graduate student life.

I would like to extend my thanks to my dissertation committee: Dr. Marco Valtorta, Dr. Yan Tong, Dr. Ramtin Zand, and Dr. Austin Downey for taking their time to serve on my committee.

Support for this research was provided by National Science Foundation under Grant No. 1910748 and through financial, hardware, and intellectual support from Texas Instruments Corporation. Specially I would like to thank Jesse Villarreal, Lucas Weaver and Todd Hahn from Texas Instruments for years of continued collaboration.

Finally I would like to thank my parents Nirmala Pathirana and Upali Abeysinghe for raising me to be the person I am today. And my brother Pubudu Abeysinghe and his family for their support.

Abstract

Digital signal processors (DSP), which are characterized by statically-scheduled Very-Long Instruction Word architectures and software-defined scratchpad memory, are currently the go-to processor type for low-power embedded vision systems, as exemplified by the DSP processors integrated into systems-on-chips from NVIDIA, Samsung, Qualcomm, Apple, and Texas Instruments. DSPs achieve performance by statically scheduling workloads, both in terms of data movement and instructions. We developed a method for scheduling buffer transactions across a data flow graph using data-driven performance models, yielding a 25% average reduction in execution time and a reduction of up to 85% DRAM utilization for randomly-generated data flow graphs. We also developed a heuristic instruction scheduler that serves as a performance model to guide the selection of loops from a target data flow graph to be fused. By strategically selecting loops to fuse, performance gains can be achieved by eliminating unnecessary transactions with memory and increasing functional unit utilization. This approach has helped us achieve up to 1.9x speedup on average for sufficiently large data flow graphs used in image processing.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Abbreviations

- ALAP ..... As Late As Possible
- ANN ..... Artificial Neural Network
- ASAP ..... As Soon As Possible
- ASM ..... Assembly Code
- BAM ..... Block Access Memory
- CISC ..... Complex Instruction Set Computer
- CPU ..... Central Processing Unit
- DAG ..... Directed Acyclic Graph
- DRAM ..... Dynamic Random Access Memory
- DSL ..... Domain Specific Language
- DSP ..... Digital Signal Processor
- FPGA ..... Field Programmable Gate Array
- GPGPU ..... General-Purpose Graphical Processing Unit
- GPU ..... Graphics Processor Unit
- II ..... Initiation Interval
- IL ..... Iteration Latency

- ILP ..... Instruction Level Parallelism
- IP ..... Iterations in Parallel
- IPU ..... Image Processing Units
- SIMD ..... Single Instruction Multiple Data
- SoC ..... System On Chip
- TIOVX ..... Texas Instruments OpenVX
- VLIW ..... Very Long Instruction Word
- VPU ..... Visual Processing Units

List of Symbols

B_n	Number of partitions for a set of n nodes
$\sum_{k=0}^{n-1} \binom{n-1}{k}$	Summation of $\binom{n-1}{k}$ choices for k items that remain after one set is removed
n_L	Number of instructions that can be executed on L functional units on the DSP
n_S	Number of instructions that can be executed on either the S units
n_D	Number of instructions that can be executed on either the D units
n_{LS}	Number of instructions that can be executed on either the L or S units
n_{LSD}	Number of instructions that can be executed on either the L, S, or D units
$\sum_{i=1}^{NI}$	Summation of values from 1 to Number of Inputs
$\sum_{i=1}^{NO}$	Summation of values from 1 to Number of Outputs

Chapter 1

Introduction

1.1 Digital Signal Processors

Image processing accelerators are now commonly integrated as part of embedded System-on-Chip (SoC) architectures. These types of accelerators go by different names, such as "Visual Processing Units (VPUs)" and "Image Processing Units (IPUs)", but are generally structured as Digital Signal Processor (DSPs)-type which use a hybrid Very Long Instruction Word (VLIW) and Single Instruction Multiple Data (SIMD) architecture, which themselves differ from general purpose processors and Graphics Processor Units (GPUs) in that they rely on compilation techniques to statically schedule all instructions, they have a complex instruction set, and use software-defined scratchpad memory and software-defined asynchronous pre-fetching and buffering of data blocks in the scratchpad using a direct memory access (DMA) controller as opposed to – or in addition to – traditional caches. Examples of such processors include the Google Pixel Visual Core [48], Qualcomm Hexagon DSP [13], Nvidia Programmable Vision Accelerator [17], and the Texas Instruments C66x & C71x DSPs [49], [37]. In addition, Field Programmable Gate Arrays (FPGAs) loosely fit this category when used with High Level Synthesis compilers [61], [25], [3].

Digital Signal Processors are a key technology for enabling the widespread deployment of Automated Driver Assistance (ADAS) systems that improve vehicle safety and save lives. DSPs are the processors that execute real-time computer vision algorithms that use camera-based sensors to detect road and lane boundaries, traffic signs

and signals, pedestrians, and other vehicles, and are instrumental for self-driving car technology. However, unlike GPUs, DSP design does not lend itself to an intuitive programming model (i.e. CUDA), which makes it difficult, costly, and time consuming for DSP programmers to meet performance requirements for new applications. This creates significant challenges for the deployment of new algorithms for use in vehicle safety systems [1].

1.2 OpenVX

OpenVX is a code-portable and performance-portable open programming interface to aid in the design of accelerated signal processing subroutines, and has widespread support on a variety of coprocessor architectures [46]. It was originally conceived as a domain specific API targeted at image processing, but it is extensible to other domains. OpenVX relies on a graph-based model that allows the programmer to compose processing pipelines by assembling a collection of cooperative kernel primitives. This way, each graph node represents a kernel and each edge represents the flow of one image or video frame between kernels or as top-level inputs or outputs. This representation potentially allows the runtime environment to identify opportunities for optimization. Examples of OpenVX vendor implementations are Intel OpenVINO [26], Nvidia VisionWorks [8], AMD OpenVX (AMDOVX) [5], and Texas Instruments Vision SDK [28]. In our work, we target the Texas Instrument's TDA2x System-on-Chip and our baseline results are given by the release version of the Texas Instruments Vision SDK with OpenVX framework (TIOVX) [10].

Kernels in an OpenVX graph exchange images, and a kernel cannot begin execution until it receives an image for each of its inputs. Executing an OpenVX graph on a DSP-type architecture requires that images be conveyed as a sequence of smaller fixed-sized tiles for the purpose of buffering in on-chip scratchpad memory. The size of the tiles impacts both the memory system performance and instruction throughput.

Additionally, tiles associated with internal edges may optionally be kept in scratchpad only or exchanged with DRAM between kernel invocations.

1.3 Image Processing Graph Optimization

The performance of a image processing graph can be improved by fusing kernels (graph nodes). This dissertation is attempting to realize this using two methods.

1. By sharing intermediate data between kernels via scratchpad which allows for intermediate results to be passed through L2 scratchpad memory and this avoids DMA transactions. This is refereed to as "Kernel Buffer Fusing".

2. By sharing intermediate data between kernels via processor registers which is the next step forward and totally cuts down even the L1 cache and L2 scratchpad memory transactions. This is refereed to as "Kernel Loop Fusing".

$$B_n = \sum_{k=0}^{n-1} \binom{n-1}{k} \times B_k \qquad (1.1)$$
$$B_0 = 1$$

The connected nodes in a graph can be grouped in various ways. A graph of n nodes may be decomposed into B_n groups, where B_n is the recursively-defined "Bell number", as defined in Equation 1.1. The Bell number scales exponentially, e.g. $B_6 = 203$, $B_7 = 877$, ..., $B_{12} = 4,213,597$. In practice, the total number of decompositions is typically small enough to be enumerated and evaluated using fast performance models as proposed by our methods above, but too large to enumerate and evaluate on the DSP.

1.3.1 Kernel Buffer Fusion

Like other domain-specific processors, DSPs favor the use of software-controlled on-chip memories instead of traditional caches. A scratchpad offers several advantages

over a traditional multi-level cache. A scratchpad combined with DMA controller operates asynchronously with the processor, allowing the programmer to overlap DRAM transfers with the corresponding compute workload.

Our method proposes new programming framework for DSPs that significantly simplifies the development of efficient DSP code. The framework is built on top of an existing open programming model, OpenVX, but uses a novel algorithm for identifying which memory structures to store intermediate results from the computation, and how to most efficiently orchestrate the set of processing stages that comprise the targeted image processing algorithm.

The developed framework includes two components. The first is a set of data-driven models for predicting the efficiency of the DSP's memory interface decomposing image data into sub-images of a given size, and a given schedule of buffering and processing each of the sub-images. The second is a model for predicting the hardware utilization and pixel throughput as a function of the number of processing stages used to process each image. Together, these models are used to evaluate a large set of possible software configurations to identify the one that will yield the best performance. This allows DSP developers to avoid manual, trial-and-error approaches for performance tuning and ultimately improves productivity for VPU programming.

One of the contributions of this dissertation is the development of performance models to predict both DMA bandwidth and instruction throughput given features of input and output tiles. Using these models, we select weakly-connected sub-graphs that exchange tiles instead of whole images. This causes the runtime environment to schedule successive kernels after processing each tile as opposed to processing each image. We refer to this as "kernel buffer fusing". Our approach performs tile size selection for all individual kernels and groups of fused kernels.

To evaluate the potential performance impact of our tuning methodology, we use large sets of randomly-generated graphs and compare their predicted performance

Table 1.1 OpenVX 1.1 Kernels That Support Fusing

Point to Point Kernels		Neighborhood Kernels
AbsDiffNode	PhaseNode	Box3x3Node
AddNode	TableLookupNode	CannyEdgeDetectorNode
SubtractNode	ThresholdNode	ConvolveNode
AndNode	MultiplyNode	Dilate3x3Node
XorNode	MagnitudeNode	Erode3x3Node
OrNode	ConvertDepthNode	NonLinearFilterNode
NotNode		IntegralImageNode
ChannelCombineNode		Median3x3Node
ChannelExtractNode		HalfScaleGaussianNode
ColorConvertNode		Sobel3x3Node
ConvertDepthNode		Gaussian3x3Node

improvement over that of the baseline software. Using this methodology, we achieve a speedup of 1.3 on average, with an average prediction error of less than 2%.

A subset of the OpenVX 1.1 kernel set supported by the fusion approach proposed in this paper is shown in Table 1.1.

1.3.2 Kernel Loop Fusion

The Texas Instruments VisionSDK includes an image processing library called VXLIB that targets their DSP architectures. The library consists of a set of primitive image processing kernels. Each kernel has a loop that operates on individual pixels or groups of pixels when multiple pixels are processed via SIMD instructions.

Fusing the loops of multiple VXLIB kernels instead of sequentially executing them allows the kernels to exchange intermediate results through registers instead of through the L1 cache and on-chip scratchpad memory [1]. This reduces the number of load and store instructions, reduces the number of cache misses, and allows the pool of functional units to be shared by multiple kernels within a basic block, increasing the utilization of functional units.

As a VLIW architecture, DSPs exploit instruction-level parallelism by scheduling multiple independent instructions in each clock cycle. Since data dependencies in

the loop body would normally make this impractical, DSP compilers perform modulo scheduling to transform the innermost loop in a way to reduce or eliminate intra-loop dependencies and maximize the distance between loop-carried dependencies.

In this dissertation we propose a technique for fusing VXLIB kernel loops prior to compilation, with the fusing decision being made in a closed-loop process in which candidate sets of loops to fuse are systemically fused and their resultant performance is estimated using a performance model that is capable of estimating the resultant functional unit pressure, register pressure, and achieved loop throughput.

There are two reasons why identifying the sets of kernel loops to fuse is a challenging problem. The first is that some combinations of loops, when fused, cause the compiler to fail when performing modulo scheduling. This is because longer loop bodies may be significantly constrained by register pressure and cause register allocation to fail during scheduling. Second, although the throughput of a fused loop is generally greater than that of the effective throughput of individual loops, our goal is to identify the sets of loops that, when fused, result in the greatest overall throughput improvement for the whole graph as compared to the baseline case of not fusing any loops.In this paper we propose a technique for fusing VXLIB kernel loops prior to compilation, with the fusing decision being made in a closed-loop process in which candidate sets of loops to fuse are systemically fused and their resultant performance is estimated using a performance model that is capable of estimating the resultant functional unit pressure, register pressure, and achieved loop throughput [2].

There are two reasons why identifying the sets of kernel loops to fuse is a challenging problem. The first is that some combinations of loops, when fused, cause the compiler to fail when performing modulo scheduling. This is because longer loop bodies may be significantly constrained by register pressure and cause register allocation to fail during scheduling. Second, although the throughput of a fused loop is generally greater than that of the effective throughput of individual loops, our goal is

to identify the sets of loops that, when fused, result in the greatest overall throughput improvement for the whole graph as compared to the baseline case of not fusing any loops.

For a given directed acyclic graph (DAG) of VXLIB kernels, our proposed tool searches for an optimal graph decomposition of weakly connected subgraphs that, when the loops of each subgraph are fused, achieves maximum effective throughput.

In order to generate compilable code for merged loops, we developed HeRCide, a library built using C++ meta-programming that simplifies the process of combining and connecting loop bodies. Additionally, we developed a fast, heuristic scheduler that can estimate the feasibility and performance of a given set of fused loops, based on a database of single-scheduled iterations for each supported VXLIB kernel.

We show that this approach can achieve up to a 1.9 speedup on average for a graph with 10 nodes as compared to equivalent non-fused baseline graph.

Chapter 2

Background

There are two important performance optimizations we can do for DSPs:

1. Scheduling DMA transactions to transfer I/Os between L2 scratchpad and DRAM in parallel to processing.

2. Scheduling more instructions in loops to increase functional unit utilization.

2.1 Scratchpad Memory Optimization

Like other domain-specific processors, Digital Signal Processors (DSPs) favor the use of software-controlled on-chip memories instead of traditional caches. A scratchpad offers several advantages over a traditional multi-level cache. A scratchpad combined with DMA controller operates asynchronously with the processor, allowing the programmer to overlap DRAM transfers with the corresponding compute workload. For applications where the access pattern is fixed, a scratchpad makes more efficient use of memory, as caches are susceptible to conflict misses when accessing multidimensional data. Scratchpads can also achieve higher throughput to off-chip memory by transferring larger blocks of consecutive words. For example, a scratchpad might transfer tiles of 128 by 96 four byte pixels, generating 512-byte consecutive transfers per row, larger than a typical cache line of 64 bytes.

The introduction of image tiling, DMA, and scratchpad into the OpenVX runtime on the Texas Instruments C66 DSP provided a benefit of a 2.3 speedup as compared to using cache [10]. This baseline implementation decomposes each input, output,

and intermediate image into 64x48-pixel tiles, each of which is transferred between the on-chip L2 scratchpad and off-chip DRAM using the integrated DMA controller. The DSP cores in our target platform (the TDA2x) have an L2 memory size of 288 KB, which the program code (usually the bootloader) can configure as a partial hardware-controlled L2 cache and partial software-controlled scratchpad. For our experiments, it is configured as a 128 KB scratchpad SRAM with the rest as cache.

Since the DSP is an in-order architecture, the latency of data exchanges between the cache and DRAM cannot be hidden and, as a result, a cache miss always causes the DSP to idle until the miss is complete. On the other hand, the latency of exchanges between the scratchpad and DRAM may be hidden by exposing concurrency between the DSP core and DMA controller. In this way, the DSP processes tile n concurrently with the outgoing transfer of output tile $n-1$ and incoming transfer of input tile $n+1$. This is referred to as "ping-pong buffering" or "double buffering". The scratchpad memory, while being an on-chip memory, exists at level 2 and is itself cached by a 32 KB 2-way set associative L1 cache.

When executing an OpenVX graph, all tiles comprising each of a kernel's output edges are transferred back from scratchpad to DRAM before the DSP begins executing any other kernel in the graph. For example, if a graph contains a kernel that has two inputs and one output, the software will allocate three scratchpad buffers–one for each edge–until the kernel has completed processing its output image and the DMA engine transfers the last of its tiles to DRAM. At this point, the software will re-allocate the scratchpad memory for the next kernel and start fetching tiles from DRAM.

As shown in Fig. 2.1, the software allocates two scratchpad buffers, ping and pong, for each input and output image of the kernel. Each buffer holds one tile and the software operates an outer loop that processes one tile. In each iteration, the software instructs the DMA controller to transfer the previously-computed output

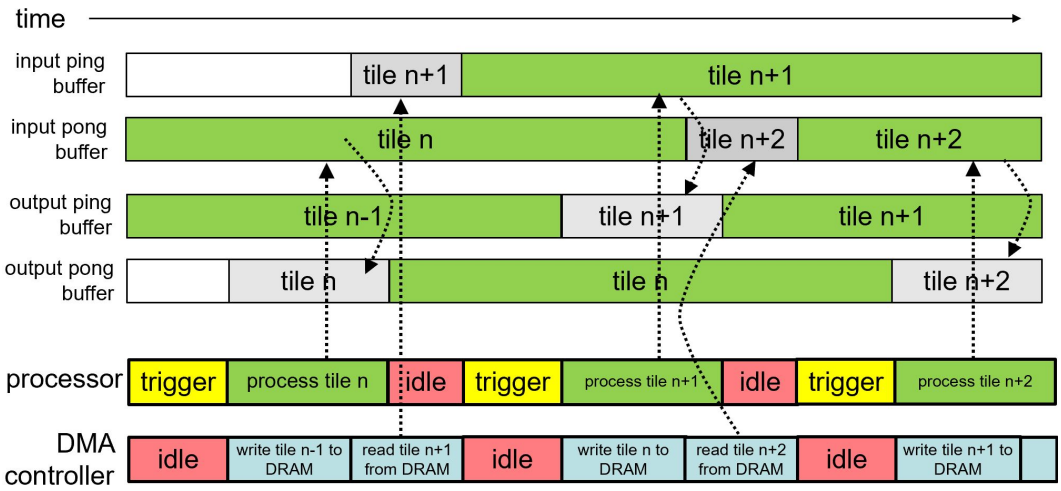

Figure 2.1 Example timing diagram for double buffering execution on the TI C66x DSP. In this example, the execution is memory-bounded, since the time required for the DMA transfers exceeds the time required for processing one tile. "trigger time" is the time needed for the processor to initiate concurrent DMA transfers.

tile into DRAM and transfer the next input tile into scratchpad. After computing, the processor might need to wait until the DMA transfer completes before processing the next tile.

The "trigger time" refers to the time required for the processor to initiate DMA transfers and the "waiting time" refers to the difference in total DRAM transfer time and compute time. Note that for smaller tiles, the cumulative trigger time across the frame can be significant since there are more blocks to trigger per frame than for larger tiles. The tile size can vary significantly, since the pixel size varies from one to four bytes and some kernels have more inputs and outputs than others. If a kernel's compute time exceeds its DMA transfer time then it is compute bound and its waiting time will be near zero. If a kernel's DMA transfer time exceeds its compute time, it will be memory bound.

Tile size selection has a potentially significant effect on performance. Also, although OpenVX's graph-based model decomposes the application into discrete operations, there are still opportunities for combining kernels within subgraphs to improve memory locality and avoid unnecessary transactions with off-chip memory.

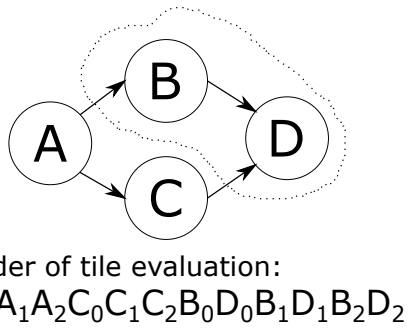

Figure 2.2 A graph with nodes A, B, C, and D, nodes B and D are grouped, and the image is comprised of three tiles. K_n denotes the execution of tile n for node K. In this case, the DSP must alternate execution of tiles within the set of grouped nodes.

For example, consider two connected kernels that each have one input and one output. During execution, each kernel requires that an input tile and output tile to be stored in scratchpad at any given time. The first kernel will transfer all its output tiles to DRAM before the second kernel begins retrieving each of these tiles for processing. Alternatively, if the first kernel conveys each of its output tiles as input to the second kernel through a scratchpad buffer and without transferring the tile to DRAM, the combination of both kernels will require that three total tiles are stored in scratchpad at any given time, comprising the input tile, intermediate tile, and output tile.

This approach reduces the total number of DMA transfers from three to two, which can improve performance if the sum of DMA transfer time of both kernels exceeds the sum of their execution time. However, storage of additional tiles will potentially limit the maximum tile size.

Also, as shown in Fig. 2.2, grouping kernels in this way increases the frequency at which the DSP must alternate its execution state between kernels, putting increased pressure on the L1 instruction cache. Likewise, processing multiple tiles at once increases the size of the working set, increasing pressure on the L1 data cache [3].

Further performance improvements with this method is presumably impossible due to,

1. The limitation of having to run the buffer merged kernels sequentially (we can't software pipeline across kernels).

2. All intermediate results need to pass through L2 scratchpad which also could cause L1 cache conflict misses.

3. Mandatory cache capacity misses if the buffer sizes are greater than L1 data cache size (32KB).

2.2 Loop Scheduling Optimization

One technique to overcome the problems described in the previous method is to convey the intermediate results between kernels through DSP core's registers. This will result in no L1 data cache misses because we are getting rid of all intermediate I/Os to cache or scratchpad memory and it will also save power because external memories have very high power usage. All the merged kernels will be a single loop which could probably be fully cached in L1 instruction cache. The only I/Os facing the memory are the global inputs of the first and last kernels that are merged if the intermediate kernels don't have global I/Os. One limitation to this method is the amount of instructions we could put in a software pipelined loop. If the instruction count is too high, it might not produce an efficient pipeline.

2.2.1 DSP Architecture

Fig. 2.3 illustrates the simplified microarchitecture of the Texas Instruments C66x DSP. Operating on a Very Long Instruction Word (VLIW) design, it emphasizes instruction-level parallelism by enabling the execution of up to eight instructions in parallel. To address the need for an excessive number of ports on the register file that would otherwise be required for this approach, the design is split into two datapaths, each featuring a register file and four associated functional units. Not depicted in

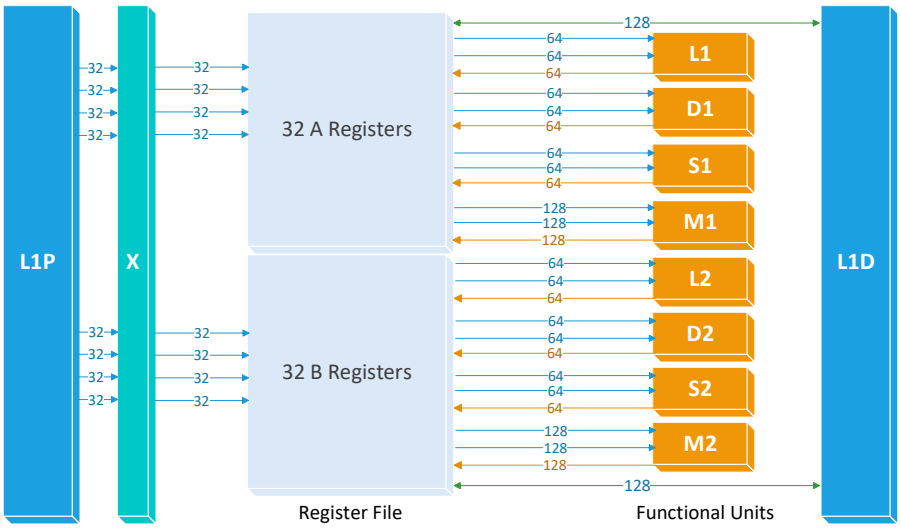

Figure 2.3 Simplified Texas Instruments C66x DSP Block Diagram [24]

the diagram is a multiplexer situated on the second input of each functional unit, enabling the use of one register from the opposite register file as a second operand when necessary.

In total, there are 64 32-bit registers available. Each functional unit can accept up to two 32-bit registers per operand for SIMD instructions, and the M unit can support up to four 32-bit registers as each operand. The D unit executes load/store instructions and both D units together can allow up to 128 bits to be loaded or stored per cycle to the L1D cache.

Each of the four types of functional units can execute a specific subset of instructions, while some instructions can be executed on more than one unit. For example LD (load) and ST (store) instructions can only be executed on the D units, while the ADD instruction can be executed on the L and S units and the MOVE instruction can be executed on the L, S, and D units. Note that the compiler determines which unit is used for each instruction.

2.2.2 Modulo Scheduling

To maximize the utilization of functional units on a VLIW DSP architecture, the compiler must transform the innermost loop in a way that exploits any iteration-to-iteration independence that may exist. To achieve this, compilers use modulo scheduling, a form of software pipelining, which overlaps successive iterations of a loop [47]. The resulting software-pipelined loop contains a prolog, software pipelined kernel ("piped kernel"), and epilog. The prolog, run once, initiates a number of iterations of the loop, "piping-up" the software pipelined loop. Execution then falls into the repeating piped kernel, which starts additional iterations, continues the work of some iterations, and completes some iterations. When no further iterations need to be started, executions falls into the epilog (run once), which completes any iterations that have started but not yet completed.

The objective of this transformation is to maximize the throughput of the loop by minimizing the number of cycles between initiations of each iteration of the loop, or "initiation interval (II)". A loop's total latency is decomposed into stages, each with length II [24], and the number of iterations that must be executed in parallel is determined by the ratio of loop latency to II, as shown in Eq. 2.1.

$$instruction\ throughput = \frac{1}{initiation\ interval} = \frac{iterations\ in\ parallel}{iteration\ latency} \tag{2.1}$$

The lower bound of the II is determined in part by the resource constraint, or the minimum number of cycles required to execute the instructions in the loop body based on the number of function units available, or $ResMII$ [47]. $ResMII$ depends on the number of instructions comprising the loop body that can only be executed on one of each of individual units (n_L, n_S, n_M, and n_D), which we refer to as single unit instructions, the number of instructions that can be executed on either the L or S units (n_{LS}), which we refer to as two-unit instructions, and the number of instructions

that can be executed on either the L, S, or D units (n_{LSD}), which we refer to as three-unit instructions (note that no instructions can execute on all four units). Since there are two of each unit type, the number of single-unit instructions must be divided by two, the number of two-unit instructions must be divided by four, and the number of three-unit instructions must be divided by six. Thus, the lowest II at which the software pipeliner can reasonably start attempting to schedule is determined in part by Eq. 2.2.

$$ResMII = max\left(\frac{n_L}{2}, \frac{n_S}{2}, \frac{n_D}{2}, \frac{n_{LS}}{4}, \frac{n_{LSD}}{6}\right) \tag{2.2}$$

The minimum II at which the software pipeliner can reasonably start attempting to schedule is also determined in part by data dependencies from iteration i to iteration $i + k$. For example, if a value is produced late in iteration i, but consumed early in iteration $i + 1$, the start of iteration $i + 1$ may need to be delayed so that the value produced from iteration i is ready by the time iteration $i + 1$ needs it. This iteration-to-iteration dependence is called a loop-carried dependence. Scheduling at a lower II than what the largest loop-carry dictates will result in an illegal software pipeline schedule, and thus the scheduling attempt at that lower II will fail as a result. Because of this, the compiler will compute the recurrence bound of a loop, or $RecMII$. The $RecMII$ is the minimum II in which a loop can be scheduled without violating the iteration-to-iteration data dependence. The actual minimum II is thus the larger of $ResMII$ and $RecMII$, i.e. $MinII = max\,(ResMII, RecMII)$.

The minimum number of parallel iterations is determined by the minimum loop iteration latency, or IL, which is determined by the critical path latency of the directed acyclic graph (DAG) that describes the dependencies in the assembly code representation of the loop body. However, the critical path delay does not include the effect of additional delays caused by resource constraints. Determining the minimum loop latency when including functional unit constraints requires a combinatorial

search for all potential mappings of instructions to functional units and cycles.

Adding to this complexity are two factors. First, various scheduling constraints can cause register lifetimes to be extended, increasing register pressure. When register allocation fails during modulo scheduling, the TI compiler will increase the II and retry modulo scheduling (and register allocation). Second, registers may become live-too-long as a result of the modulo scheduling process, which means a register is longer for II cycles. When registers that are written and read inside the loop are live-too-long, this condition results in a schedule that will not produce the correct results. The TI compiler typically attempts to split any live-too-long registers after a modulo scheduling attempt with live-too-long-split-moves [52], but this does not always work. If live-too-long splitting does not work, ii will need to be increased and modulo scheduling tried again.

Chapter 3

Related Work

Several prior efforts have been made to optimize the performance of data flow graphs for various types of architectures. Our approach is unique because of the use of data driven performance models to optimally chose the configurations parameters for a given image processing graph to achieve the highest speedup. Our models can also be directed to optimize DRAM utilization while maintaining a reasonable speedup for a given graph.

3.1 OpenVX Platform-Specific Deployments

Yang et al extended Nvidia's VisionWorks libray [8] to create a real-time OpenVX implementation [63]. ADRENALINE is a virtual many core SoC platform with a corresponding OpenVX runtime based on OpenCL [53]. It achieves a speedup of 2 to 5 for OpenVX graphs as compared to equivalent OpenCL implementations executed on the same platform. OpTIflow [27] shows how the individual nodes of an OpenVX graph can be mapped to various cores available in the TDA4V-Mid SoC (which includes ARM, C66x, and C71x DSP cores) while [11] presents a parallelized software architecture for OpenVX in TI Jacinto 7 SoC (TDA4VM). The AFFIX framework performs a heterogeneous implementation of OpenVX kernels in a CPU-FPGA setup using OpenCL and High-Level Synthesis with high and low level optimizations [55].

3.2 Dataflow Optimization via Kernel Fusing

The general approach of identifying nodes to merge in a dataflow graph (or OpenVX graph in particular) offers various benefits depending on what type of platform is targeted. These efforts may be broadly categorized into the granularity of the hardware resources targeted, such as a pool of processor cores or individual function units. In the case of mapping processor cores, node merging can provide a mechanism to constrain crossbar switch sizes in the case of FPGA-based multiprocessor system-on-chip platforms [12], improve cache performance on traditional shared memory multicores [14], and minimize data movement for FPGA-based soft-core DSPs [54] [40]. In the case of individual functional units such as the case when using FPGA-based high-level synthesis, node merging allows for constraining total hardware cost by optimizing the dataflow graph before lowering the graph to a C-language description immediately prior to its synthesis to gates [39].

3.3 Performance Modeling

Machine learning is emerging as an alternative to analytical-based [50] and simulation-based performance modeling [33] [30]. Particularly machine learning models have been shown to be effective in predicting performance of a given set of image processing kernels on various heterogeneous processors [30]. Plethora of research done in analytical modeling to find effective tile sizes for data flow graphs have not been successful across a range of processor architectures. Auto tuning, which involves experimentally evaluating a range of tile sizes is another effective and widely-used approach for tile size selection [56].

3.4 DSLs (Domain Specific Languages)

OpenVX's graph-based method for describing dataflows is similar to other efforts to develop performance-portable domain-specific languages. Quio et al extended Hipacc, an open source source-to-source compiler to automatically fuse kernels by analyzing the AST (Abstract Syntax Tree) of a compiled image processing application written in Hipacc DSL [43]. PolyMage [38] and Halide [45] are two popular domain-specific languages, both embedded in C++, for auto-tuning image processing applications. Both use a "position independent" representation, in which the programmer provides an element-level description of a computation with only relative indexing and without the surrounding loop constructs or any notion of the order in which the elements are processed. PolyMage relies on polyhedral analysis to generate tiled loop nests. Halide requires its programs to include both a functional representation as well as a meta-program that specifies a method that Halide should follow to optimize the program. An example for this is shown in [60] where Halide and MLIR were used to increasing the speed of OpenVX data flow for CPUs.

3.5 Loop Fusion

Quio et al proposed a kernel fusion technique that transformed the loop fusion problem into a graph partitioning problem and using the minimum cut technique to recursively search fusible kernels. The fusible kernels are selected based on the weights of the edges in the graph and these weights are assigned based on a benefit estimation model. They implemented their solution within Hipacc which is an image processing DSL and a source-to-source compiler and it is targeted for GPUs [42], [44]. We do have a similar approach but our models evaluating the fusibility are very different and we do not use Hipacc because it doesn't have support for Texas Instrument specific intrinsics. Kennedy and McKinley's [29] approach reorders the loop or change the

order in which loop iterations are executed by preserving output dependencies and then fuses multiple distinct loops into a single loop. They also convert the problem into a graph partition problem similar to [42] and use a novel algorithm based on maximum-flow/minimum cut to select kernels for fusing.

Meng et al developed a method to fuse parallel kernels–as opposed to loops–onto the same GPUs to improve data locality [36]. This avoids the DMA accesses to external memory and improves data locality in L1 cache. This approach is different from ours as it's not directly merging kernel loops into an one big single loop. The paper published by Intel [51] discusses about fusing loops that are nested to maximize locality. They convert the source code to a DAG (Directed Acyclic Graph) and fuse different DAGs before converting the DAGs back to source code. In contrast we fuse different image processing kernels while letting our heuristic model decide what kernels to merge.

Fraboulet et al developed a loop fusing approach designed to improve data reuse [20]. Kernel fusion could be potentially advantageous in data-intensive applications like data warehousing. Wu et al evaluated the benefits of kernel fusion on relational algebra operations implemented using CUDA for an NVIDIA Fermi GPU [59].

In our work, we fuse different kernels which have completely different functionality. But the work suggested by [19] breaks down large functions into smaller kernels and then evaluate which kernels combinations would best perform merged. They have their predictor model which primarily focuses on computation against memory access time. Xue et al proposes a technique to merge 'for' loops while removing fusion preventing dependencies by applying loop tiling [62]. In our approach we do not handle image tiling process, the framework around our loop merged kernels are suppose to handle the tiling process and Texas Instruments Vision SDK takes care of that.

Mehta et al. developed a method to perform loop transformations using polyhe-

dral framework which is a mathematical framework that includes parametric linear algebra and linear programming [35]. So they're able to merge loops that are not obvious for the compiler to automatically merge. In our approach, we don't do change code in individual kernels because these kernels are highly optimized for the TI C66 compiler.

3.6 Instruction Scheduling

Instruction scheduling is an NP Complete problem even when the basic building block of code is only several independent DAGs [41]. Wilken et al presents an approach to instruction scheduling based on integer programming and graph transformation to simplify the data dependency graph [58]. Lee and Ha came up with a cost effective approach for scheduling instructions for multi core DSPs [31]. Timmer et al proposed a method to model resource and instruction set conflicts before scheduling an instruction DAG[57]. Deng et al implemented a two-dimensional constrained dynamic programming approach and a quantitative model for instruction scheduling which achieved near optimal performance for VLIW DSP at a cost of time overhead [15]. Bahtat et al developed a enumeration based resource constrained heuristic for modulo scheduling in VLIW architectures which help them improve the performance of a set of signal processing algorithms [7]. Leupers implemented a partitioning and instruction scheduling technique for cluster based VLIW architectures to reduce the interdependence among clusters [32]. Desoli proposed a cluster based instruction scheduling approach to speedup the convergence of a deterministic descent algorithm which was able to beat a common heuristic known as BUG (Bottom Up Greedy) by 5 to 50% [16].

Optimal assembly code generation is an integral part of a successful instruction schedule. Faraboschi et al used region selection and region enlargement techniques such as loop unrolling and branch target expansion to optimizing code generation

for instruction level parallel processors like VLIW and Superscalar [18]. Arslan and Kuchcinski used a state-of-the-art constraint solver to find solutions to large scale code generation problems by redefining instruction selection and scheduling as a constraint satisfaction problem [6]. Gibbons and Muchnick developed a code reorganizing algorithm to significantly reduce the number of runtime pipeline interlocks for pipelined architectures [21].

JOKer framework provides an automatic end-to-end multi-level code generator for kernel optimization through three optimization primitives: Loop Transform, Vectorization and Instruction-level Optimization [64].

A novel reinforcement learning based approach using graph neural networks was attempted for scheduling instructions for a DSP architecture but the results could not achieve any significant improvements [4].

3.7 Register Allocation

In our heuristic scheduler used to predict which loops to fuse from a given set of VXLIB kernels, it was not necessary to allocate physical registers according to the DSP architecture. But we do make sure that register pressure will not go above the register limit constraints on hardware. Register allocation is an NP Hard problem [34]. We did not want to increase the computation time for our Heuristic Scheduler as we have to evaluate many groupings for a given VXLIB graph to find the best set of sub-graph groupings. Furthermore the accuracy of the predictions using our heuristic scheduler was good enough to find the optimal or near optimal graph groupings. Hence we didn't need to implement a register allocator. Register allocation is a classic graph coloring problem [9].

Chapter 4

Kernel Buffer Fusion

Kernel buffer fusing allows run-time environment to pass smaller tiles of an image between kernels through L2 scratchpad buffer rather than using DMA engine to read whole images back and forth from external memory. But buffer fusion has to be done methodically using right kernels and select the most appropriate tile size to achieve the best performance. In our work we use performance models to predict the best characteristics for buffer fusion.

4.1 Performance Considerations

The OpenVX programming model grants the runtime environment two degrees of freedom for optimization.

1. The first is how to decompose the input, output, and intermediate images into tiles. DMA performance depends on characteristics of the DMA transfers, such as tile size and shape. Tile size is denoted by W (Width) x H (Height). The tile size and shape affects DMA performance because the width and height determine the number of consecutive and nonconsecutive addresses accessed from DRAM. The width and height also affect compute performance because of the impact of L1 conflict misses, the startup overhead of software pipelining loops, and because some tile sizes cause the tile to extend further beyond the right- and bottom- boundary of the image when processing the last column and last row of tiles, causing differing levels of redundant calculation and/or

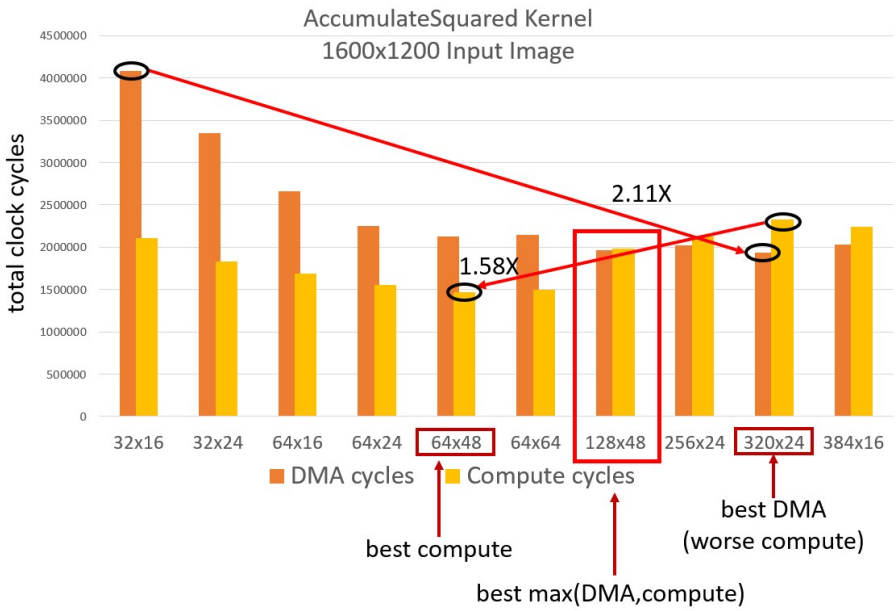

Figure 4.1 DMA and compute cycles for AccumulateSquared Kernel for a Variety of Tile Sizes.

inability to use full SIMD instructions for those tiles. Since tile transfers and computation are perform concurrently, the total execution time for an OpenVX kernel is $max(time_{DMA}, time_{compute})$.

2. The second is how to schedule the kernels onto the processor. The processor can execute the kernels in any order so long as it obeys the graph's data dependencies. The data dependencies are defined using the graph edges, but their granularity can be in terms of whole images or individual tiles. This way, the runtime environment can execute each kernel for a whole image or only a single tile. When the execution order is tile by tile, the processor must switch between kernels at a higher rate than in the former case, which potentially affects L1 instruction and data cache performances.

We propose the use of DMA and compute performance models to facilitate achieving automated exploration of optimization decisions.

4.1.1 Performance Bounds

Fig. 4.1 shows a comparison of the total number of DMA and compute cycles required to process a 1600x1200 pixel image for the AccumulateSquared OpenVX kernel for tile sizes ranging from 32x16 to 384x16. The best DMA performance is achieved at tile size 320x24, which is a 2.11 speedup as compared to the worst-performing tile size of 32x16. The best compute performance is achieved at 64x48, which has a speedup of 1.58 as compared to the worst-performing tile size of 320x24. Note that 320x24 is the worst-performing tile size for compute but the best for DMA. However, since the execution time is $max(time_{DMA}, time_{compute})$, the best performing tile size is 128x48.

The current release of the Texas Instruments OpenVX implementation in VISION SDK sets all tile sizes to 64x48. As per the performance measurements done in a TDA2x evaluation board, results show that for a variety of OpenVX "individual kernels", the 64x48 tile sizes achieves within 21% of the best performing tile size for DMA time, and within 8% of the best performing tile size for compute time, and within 12% of the best since the execution time is $max(time_{DMA}, time_{compute})$.

4.1.2 Kernel Fusing

Under default behavior, the images transferred along the edges of an OpenVX graph are buffered in external DRAM. Specifically, when processing each image as a grid of tiles, a kernel will store each output tile in scratchpad memory while it is being computed, and after completion will transfer it into its position within the output image in its external DRAM buffer. When a successor kernel is executed that consumes the same image as input, each of its tiles are subsequently read back into scratchpad for use as an input to the kernel.

Kernel fusing is a technique where the tiles sent between a predecessor and successor kernel are stored only in on-chip scratchpad, avoiding the round-trip transfer to external DRAM that would otherwise be performed in the baseline software. This

is possible when the connection between the kernels is not connected to a top-level output or read by any other kernels, unless those kernels are also a member of the fused set.

Kernel fusing has two effects on execution behavior. First, it changes the frequency in which kernels are executed. Without kernel fusing, each kernel continuously executes until it completes the processing of the entire output image(s). On the other hand, when a set of kernels are fused, each kernel in the fused set processes only one output tile, at which point its downstream kernels process a single tile and the process repeats until the whole image is processed.

This is depicted in Fig. 2.2, where kernel A and C each execute all the tiles associated with their inputs without interruption, after which fused kernels B and D must alternate execution after processing each of their tiles.

The advantage of kernel fusing is that it eliminates all of the transfers of intermediate images to external DRAM. This can shift the performance bottleneck from memory bandwidth to compute throughput and also reduce the DRAM utilization when executing the fused tiles and lowering the average DRAM utilization for the whole graph.

There are several constraints when choosing which kernels to fuse. First, in order to achieve the intended effect of reducing transfers to external DRAM, the fused kernels should be chosen such that there is at least one intermediate image that is only connected to kernels in the fused set. In other words, the subgraph consisting of the fused kernels should be weakly connected, meaning that there is a path between every pair of kernels if the edges were bidirectional.

Second, as the number of fused kernels is increased, there is a corresponding increase in the scratchpad buffer requirement. Specifically, there must be sufficient space in the scratchpad to store all the input, output, and intermediate tiles needed by the fused set of kernels. Also, since the input and output tiles are double buffered,

there must be allocation of two of each of these . Thus the scratchpad capacity limits the number of kernels that can be fused. Additionally, since the scratchpad memory is cached by the L1 data cache, if the set of fused kernels covers a total size that exceeds the L1 data cache, then as the fused kernels access all the tiles, they will experience increased dynamic stalls due to increased frequency of L1 capacity and conflict misses.

Third, only certain combinations of kernels can be fused. Due the double-buffering used to explicitly transfer input tiles into scratch memory, Kernels must use the same tile size for all inputs. This becomes an important restriction when fusing kernels, since a kernel with two inputs cannot accept tiles from two other kernels in the same fuse set with different output tile sizes. Note that the whole fuse set is associated with an assigned tile size, but internal connections between the kernels in the fused set may have their size reduced. This occurs when a fused set includes a kernel that generates a smaller output tile size than its input tile size because it performs neighborood-based or stencil operations such as 2D filtering.

Finally, the tile size and shape affects the achieved DRAM bandwidth and the compute throughput for each kernel differently as described in Section 2. Likewise, it affects each set of fused kernels differently.

As a result of these constraints and impacts on performance and DRAM utilization, deciding which kernels to fuse must be informed by a performance model or through exhaustive testing on hardware.

4.1.3 Enumerating OpenVX Graphs

In general, a graph of n nodes may be decomposed into B_n distinct groups, where B_n is the recursively-defined "Bell number", as defined in Eq. 1.1. The Bell number scales exponentially, e.g. $B_6 = 203$, $B_7 = 877$, ..., $B_{12} = 4,213,597$. For an OpenVX graph, the possible node groupings (fuse sets) must be weakly connected with respect

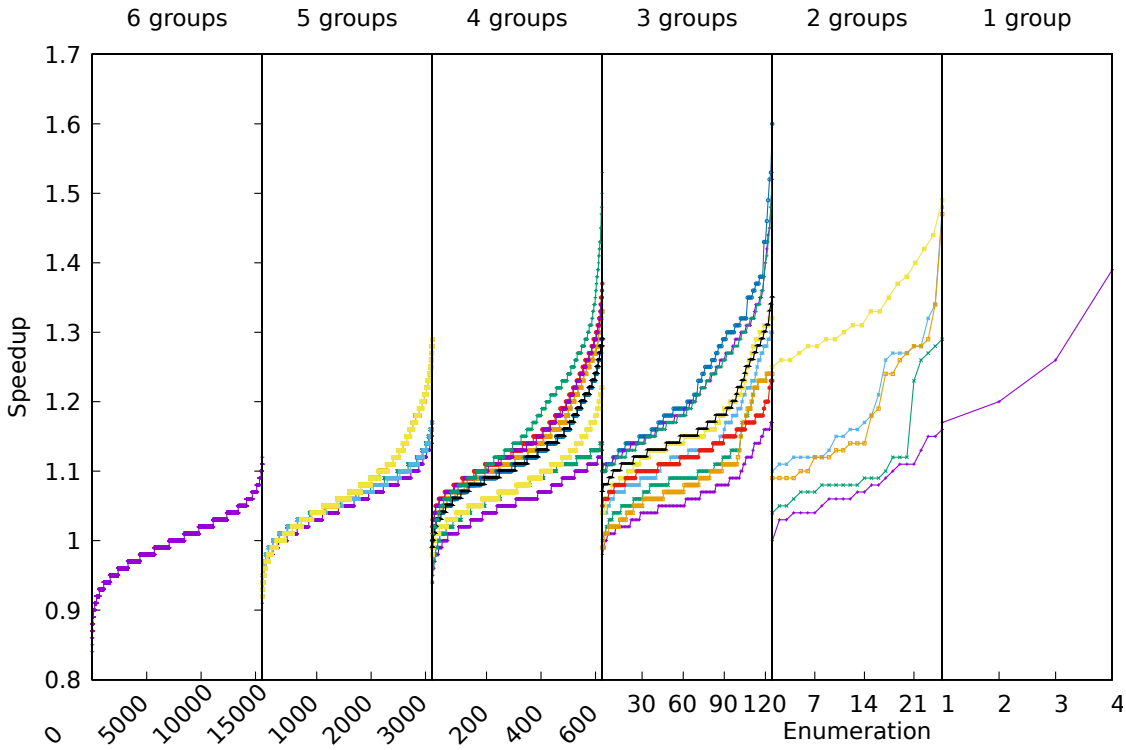

Figure 4.2 Distribution of observed speedups of grouping and tile size enumerations for a single 6-node graph, split into six distributions with each having an equal number of groups.

to the edges that carry image data. This means there must be a path between any pair of nodes in a group if all edges are treated as bidirectional.

For each input graph we enumerate all valid node groupings. For each grouping, we enumerate all possible tile size to groups chosen from 48x36, 64x48, 64x64, 80x60, 128x48. Enumerations that exceed the scratchpad capacity of 128 KB or that produce runtime errors (e.g. fusing not possible due to internal tile size mismatches within the group) are disregarded. Note that a grouping could be a single kernel.

Figure 4.2 depicts the size of the enumeration space and the impact on performance of various enumerations for a 6-node graph. The figure is split into six sections, with each section showing a distribution of speedups relative to the baseline case of no grouping (equivalent to six groups) and using a 64x48 tile size for all kernels.

Each section corresponds to a different number of groups, from 6 groups in the leftmost section to one group in the rightmost section. Each of the data points

represents an assignment of tile sizes to each of the groups and each line represents a single assignment of kernels to groups.

For example, a graph comprised of kernels A, B, C, D, E, F with the kernels assigned to three groups and corresponding group assignments $1, 1, 2, 2, 3, 3$ has $5^3 = 125$ ways to assign the six possible tile sizes to the three groups and the speedups for each of these is shown along one of the lines under the section containing three groups. Likewise, the 125 enumerations for the group assignments $1, 1, 1, 2, 2, 3$ are shown on another line.

As shown in the plot, the highest achieved speedup is 1.6 for one of the enumerations in the section containing three groups but any speedup greater than 1.4 is extremely rare among 4-, 3-, and 2-group enumerations and non-existent in the others.

4.1.4 Optimizing both Performance and DRAM Utilization

Digital Signal Processors (DSPs) are always part of a larger System-on-Chip architecture, generally sharing one DRAM interface among a large diverse set of processors and/or co-processors. In addition to improving performance, fusing kernels and optimizing tile sizes allows for minimizing DRAM utilization when executing the OpenVX graph. In other words, in some cases it may be desirable to choose an enumeration that causes the computation time to far exceed the DMA time in order to free the DRAM interface for other processors sharing the System-on-Chip.

Table 4.1 DRAM utilization of a random 6 node OpenVX graph sorted based on number of groups

Number of Groups	Number of Enumerations	Max Speedup	Corr. Execution time	Corr. DMA time	Corr. DRAM Utilization	Minimum DRAM Utilization	Corr. Execution time	Corr. DMA Time	Corr. Speedup
6	15625	1.24	11.06 ms	10.46 ms	95%	84%	12.21 ms	10.25 ms	1.16
5	15625	1.30	11.44 ms	10.26 ms	90%	72%	12.32 ms	8.89 ms	1.17
4	6125	1.41	11.49 ms	8.76 ms	76%	63%	14.66 ms	9.24 ms	1.14
3	1175	1.42	11.72 ms	7.27 ms	62%	52%	14.96 ms	7.79 ms	1.13
2	110	1.39	12.27 ms	6.80 ms	55%	44%	14.82 ms	6.48 ms	1.16
1	4	1.21	14.56 ms	5.10 ms	35%	35%	14.56 ms	5.10 ms	1.21

Table 4.2 DRAM utilization of another random 6 node OpenVX graph sorted based on number of groups

Number of Groups	Number of Enumerations	Max Speedup	Corr. Execution time	Corr. DMA time	Corr. DRAM Utilization	Minimum DRAM Utilization	Corr. Execution time	Corr. DMA Time	Corr. Speedup
6	15625	1.07	11.97 ms	16.14 ms	100%	100%	11.97 ms	16.14 ms	1.07
5	15625	1.30	11.37 ms	12.60 ms	100%	78%	15.02 ms	11.71 ms	1.08
4	6125	1.45	11.39 ms	11.32 ms	99%	57%	14.89 ms	8.54 ms	1.15
3	1200	1.50	11.68 ms	10.27 ms	88%	33%	15.24 ms	5.03 ms	1.14
2	120	1.48	11.82 ms	8.52 ms	72%	28%	15.60 ms	3.72 ms	1.20
1	5	1.46	12.86 ms	3.51 ms	27%	15%	16.46 ms	2.55 ms	1.20

Table 4.1 shows the maximum speedup and minimum DRAM utilization for all the enumerations of a different six node graph. These results show how either performance or DRAM utilization may be chosen as a primary optimization objective while still achieving a reasonably-good value for the other.

For example, the best speedup of 1.42 is achieved with 3 groups and has a corresponding DRAM utilization of 62%, while the minimal DRAM utilization of 35% is achieved with 1 group while still achieving a 1.21 speedup. Note that optimizing for either maximum performance or minimal DRAM utilization is possible with the approach described below.

The stats for the graph shown in Table. 4.2 indicates that 88% dram utilization is needed to achieve the best speedup possible which is 1.5. To find this value our models described bellow evaluated 38,700 enumerations. This table also indicates that a DRAM utilization as low as 15% could be achieved for a speedup of 1.2.

4.1.5 DMA Model

As shown in Fig. 2.1 the DMA engine concurrently transfers the previously-computed output tile(s) from scratchpad to DRAM, and then transfers the next input tile(s) from DRAM to scratchpad. At the same time, the DSP core processes the current tile(s), reading and writing to scratchpad. DMA performance depends on characteristics of the transfer, such as the size of the transfer, the number of consecutive bytes accessed, and the stride length and frequency. Our approach is to develop a DMA performance model that associates features of the transfer to an achieved effective DMA bandwidth. Note that the DMA engine's waiting time will be greater than 0 cycles when the computation is compute bounded, while the DSP's waiting time will be greater than 0 cycles when the computation is memory bounded.

To build a training set for the model, we use performance counters to measure effective DMA bandwidth for individual OpenVX kernels as well as groups of up to

10 fused OpenVX kernels over a variety of tile sizes and pixel depths, total number of inputs and outputs, and the differences in input and output tile size caused by the halo region of stencil-based kernels. The data set covered is comprised of 23,000 kernel invocations over 27 tilable OpenVX 1.1 [22] kernels for all compatible pixel depths and tile sizes of 48x36, 64x48, 64x64, 80x60, 128x48 for a 1600x1200 image size. From these tests we found that DMA throughput varies from 190 MB/s to 4.01 GB/s. Each configuration is averaged over 100 runs. In general, tile width plays an important part of DMA bandwidth because wider tiles have more consecutively accessed pixels, which reduces the frequency of row activations in the DRAM controller. Also, total tile size determines the total payload transferred, reducing the impact of time needed to start the DMA transfer.

We developed methods to increase the number of features that comprise the independent variables for which we hope to predict bandwidth. In the original dataset, we labeled each observation as a feature vector using a combination of elements chosen from: (1) input tile width in pixels (**TIW**), (2) input tile height in pixels (**TIH**), (3) output tile width in pixels (**TOW**), (4) output tile height in pixels (**TOH**), (5) total input width (tile width x pixel size x number of inputs) (**TTIW**), (6) total output width (tile width x pixel size x number of outputs) (**TTOW**), (7) total input size, (TTIW x TIH) (**TIS**), (8) total output size, (TOW x TOH) (**TOS**), (9) number of inputs (**NI**), (10) number of outputs (**NO**), (11) pixel depth (**PD**), (12) stencil neighborhood width (**SNW**), and (13) stencil neighborhood height (**SNH**). Each combination of feature vectors caused multiple observations to have the same feature vector value but with different DMA bandwidths. We refer to each of these sets of observations as "classes".

In the dataset, the chosen input parameters are: (1) number of inputs (**NI**), (2) number of outputs (**NO**), (3) total input bytes ($\sum_{i=1}^{NI} \textit{pixel depth per each input}$) (**TIB**), (4) total output bytes ($\sum_{i=1}^{NO} \textit{pixel depth per each output}$) (**TOB**), (5) total

tile input width ($\sum_{i=1}^{NI} tile\ width \times pixel\ depth\ per\ each\ input$) (**TTIW**), (6) total tile output width ($\sum_{i=1}^{NO} tile\ output\ width\ \times\ pixel\ depth\ per\ each\ output$) (**TTOW**), (7) tile input height (**TIH**), (8) tile output height (tile height for each output averaged) (**TOH**).

Note that stencil-based kernels (neighborhood-based kernels) generate a smaller output tile than the corresponding input tile. For example, if the input tile size for a Box3x3 kernel is 64x48, the output tile size will be 62x46 because the Box3x3 uses a 3x3 stencil. The effect of these kernels are reflected on the **TTOW** and **TOH** parameters. Some kernel and sets of fused kernels exhibit the same combination of feature values but yield slightly different DMA bandwidths. We refer to each of these sets of observations as "classes". For classes with greater than one member, the class feature vector is associated with the mean of DMA bandwidths observed for the member of the class.

For the input feature set, we evaluated two machine learning techniques to predict the output bandwidth. The dataset was divided into 80% training data and 20% testing data. A neural network with two fully connected layers achieved RMS training and testing error of 129.9 MB/s and 131.5 MB/s, respectively.

We also evaluated an interpolation-based method, which achieved an RMS training and testing error of 53 MB/s and 104.2 MB/s, respectively. The interpolation model is trained by collecting the feature vectors and the corresponding observed DMA bandwidth for each. Each set of observations sharing the same feature vector are consolidated into a single datapoint and associated with a bandwidth that is the arithmetic mean of the bandwidths observed for the set.

During deployment, the model will predict the effective DMA bandwidth for a single kernel or fused set of kernels by using its feature vector to perform a lookup in the table and return the associated bandwidth. If there is no entry in the table for the input features, a prediction is made by interpolating the bandwidth among a

Algorithm 1 Pseudocode for Interpolation

Input: F_{in}, $classes$, $bandwidth$
Output: $bandwidth_{predicted}$
for each $class$ in $classes$ **do**
  $dist\_to\_class_i = |F_{in} - class_i|$
end for
$sort(dist\_to\_class)$
sum = $\sum_{i=1}^{20} dist\_to\_class_i$
for i = 1..20 **do**
  $weight_i = dist\_to\_class_i / sum$
  $weight_i = 1 / weight_i$
end for
sum = $\sum_{i=1}^{20} weights_i$
$bandwidth_{predicted} = 0$
for i = 1..20 **do**
  $weight_i = weight_i / sum$
  $bandwidth_{predicted}$ += $weight_i \times bandwidth_i$
end for

cohort of the 20 nearest feature vectors in the table.

This is shown in Algorithm. 1, which accepts the input feature vector F_{in}, the set of feature vectors associated with each class $classes$, and the associated observed or average of observed bandwidth for each class $bandwidth$.

For each class $class_i$, the model calculates the distance between the input features F_{in} and the features of each of the classes $classes_i$. The model sorts the classes according to their distances to the input features, then the model calculates a weighted average with respect to the inverse normalized distances.

4.1.6 Compute Model

The C66x DSP relies on the compiler to statically schedule its instructions. All stalls resulting from data, control, and structural hazards are explicitly defined in the object code and the number of cycles per loop iteration is reported by the compiler. Since instructions are statically scheduled, the only source of performance uncertainty (aside from DMA bandwidth) are the number of stalls caused by L1 cache data misses.

As shown in Fig. 4.1, the number of raw compute cycles needed by an OpenVX kernel (without including the effects of the DMA controller) is determined by the kernel and its associated tile size. The kernel's L1 access pattern and its tile size determine the number of stalls from L1 data cache misses. L1 data cache performance is maximized at different tile sizes for different kernels.

Similar to the DMA model, our compute model is built around a table that associates the kernel name, its input count, its output count, the number of input bytes per pixel, the number of output bytes per pixel, its stencil neighborhood width, its stencil neighborhood height, and its tile width and tile height with the observed compute time.

The number of input bytes per pixel is calculated inclusive of all the kernel's inputs and the number of output bytes per pixel is calculated inclusive of all the kernel's outputs. For example, a kernel with two inputs each carrying an image with four byte pixels is assumed to have eight bytes per input pixel.

The L1 instruction cache can potentially affect performance when the processor alternates between multiple OpenVX kernels after processing each tile. For the results shown below, we have remapped the kernel code into a contiguous memory block to minimize L1 instruction cache misses.

The L1 data cache performance, on the other hand, depends on the access pattern to the input, output, and intermediate tiles for a given fused set of kernels, making it impractical to optimize. Larger fused sets, having a correspondingly larger footprint of active tiles, are generally more adversely affected by L1 data cache misses than smaller fused sets.

We refer to this effect as "λ scaling", where λ defines the observed performance of a fused set of kernels relative to the sum of execution times of its constituent kernels when executed alone and using the baseline tile size of 64×48.

Texas Instruments Vision SDK's TIOVX allows the programmer to manually

define sets of fused kernels in a given OpenVX graph. In order to construct a training set for predicting λ, we generated thousands of random OpenVX graphs and executed each with every valid enumeration of weakly-connected fused subgraphs. Using this method we have collected 10,381 data points.

From this data we determined that the optimal set of features for predicting λ are:

1. the number of fused kernels,
2. the total tile footprint of the fused set = tile size in bytes for all input edges $\times$ 2 + tile size in bytes for all internal edges, and
3. the total memory intensity = sum of execution cycles per pixel of the individual kernels in the fuse set.

Using these parameters, the data set is divided into 7,880 classes, which is an average of 1.32 observations per class.

We evaluated three techniques using the above parameters to accurately predict the λ value (using 8,335 training samples, 2,106 testing samples):

1. interpolation-based model: training RMS error 0.003, testing RMS error 0.026,
2. linear regression model: training RMS error 0.37, testing RMS error 0.28, and
3. multi-level perception (MLP): we tested various MLPs with one to 50 hidden layers, with the best training RMS error 0.08 and best testing RMS error 0.08.

As with the DMA model, we use the interpolation-based method for the λ model.

4.2 Automated Node Merging and Tile Size Selection

There are no publicly-available benchmarks for OpenVX. In the literature, previous work on optimizing OpenVX graphs uses a set of relatively small graphs to evaluate

the proposed techniques [54] [14], making it difficult to generalize results. In order to evaluate our approach over a large set of OpenVX graphs, we developed a tool that to randomly generate synthetic graphs having a given number of kernels. Of these, only graphs that are unique, valid, and schedulable (as determined by the vendor runtime) are used. For each randomly-generated graph, we enumerate all possible weakly-connected subgraphs and tile sizes for each subgraph and consider each as a potential optimal configuration.

4.2.1 Model Accuracy

As shown in Equation 1.1, n nodes may be decomposed into B_n groups. For an OpenVX graph, the possible groupings must be weakly connected with respect to the edges that carry image data (the OpenVX vx_image type). This means there must be a path between any pair of nodes in a group if all edges are treated as bidirectional (otherwise there would be no reason to group the nodes).

For each graph, we enumerate all valid node groupings. For each grouping, we enumerate all possible tile size to group assignments, chosen from 48x36, 64x48, 64x64, 80x60, 128x48. Assignments that exceed the scratchpad capacity of 128 KB are disregarded. Note that a grouping could be a single kernel on itself too. For each grouping and assigned tile size, we use the DMA and Compute models to predict its execution time, which is $max(DMA\ time, Compute\ time)$. The graph execution time is computed as the sum of group execution times.

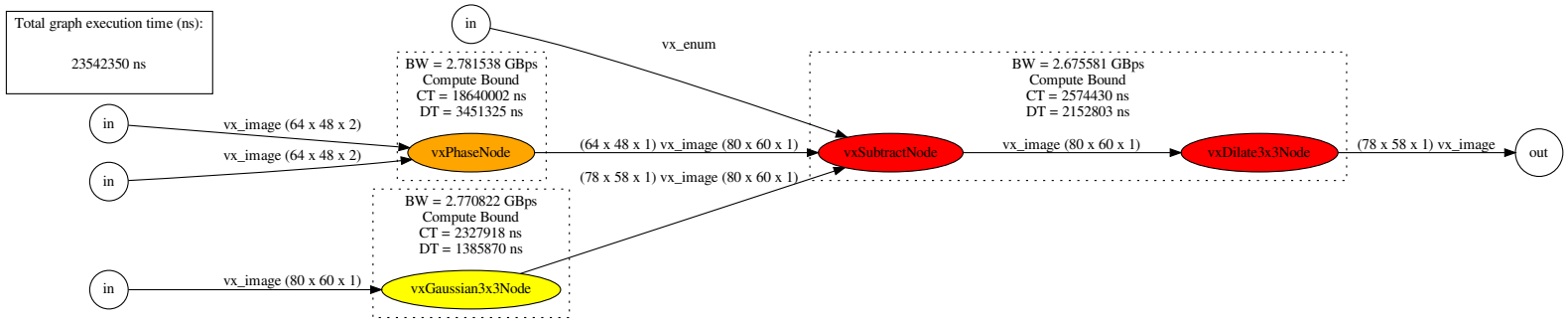

Figure 4.3 Predicted performance of a 4 node graph generated by our random graph generator.

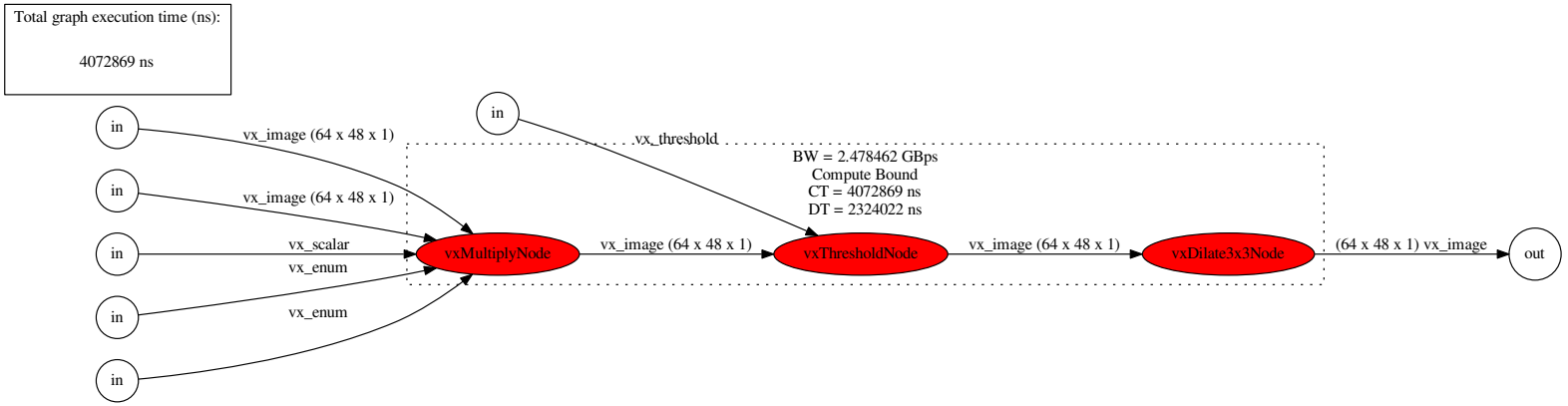

Figure 4.4 Predicted performance of a fused Multiply-Threshold-Dilate graph generated by our random graph generator.

Figure 4.3, shows an example of a 4 node graph generated by our random graph generator tool. In this instance our model has decided that the best performance is achieved if the vxSubtractNode and vxDiliate3x3Node kernels are fused with tile size 80x60, while the vxPhaseNode executes individually with a tile size of 64x48, and the vxGaussian3x3node kernel executes individually with tile size 80x60. The predicted compute time, DMA time, and DMA bandwidth for each kernel group are shown next to each group. vxSubtractNode and vxDiliate3x3Node kernels are both colored red, indicating that they are fused. Here all three groups are predicted to be compute bound and the graph execution time is the sum of execution times of all groupings.

For another example, Figure 4.4, is a 3 node graph. In this instance our model has decided that the best performance could be achieved if all 3 nodes are fused together and the tile size was set to 64 x 48. Notice that all 3 nodes have the same color, which means they're a group. The compute time and the DMA times are shown on top of the group demarcation. Here the compute time is higher than the DMA time so the fused kernel is compute bound and the DMA bandwidth for the grouping is calculated to be 2.48 GBps.

To validate the performance predictions, we deploy the same OpenVX graph with the same groupings on the Texas Instruments TDA2xx EVM (Evaluation Platform). The EVM is designed around a System-on-Chip that includes two C66x DSP cores and ARM Cortex-A15 CPUs. The CPUs execute the boilerplate code that creates and validates the OpenVX graph, creates fused kernel groupings, assigns tile sizes to each kernel/group, and offloads the graph to the DSP to gather performance data.

Table 4.3 Comparison of average speedup values for over 500 graphs per each node count

Node Count	Median EVM Speedup	Average Best EVM Speedup	Average Prognosticated Speedup	Average Model Predicted Speedup	Highest EVM Speedup	EVM vs Model Error %	EVM vs Prognosticated Error %
2	1.31	1.40	1.38	1.39	2.67	4.81%	1.23%
3	1.32	1.38	1.37	1.38	2.88	4.64%	1.32%
4	1.30	1.35	1.33	1.34	2.71	4.51%	1.73%
5	1.26	1.31	1.29	1.32	2.34	3.85%	1.65%
6	1.24	1.29	1.26	1.29	2.03	4.08%	2.03%
7	1.16	1.24	1.22	1.26	2.07	3.35%	1.25%
8	1.19	1.24	1.22	1.27	2.22	3.69%	0.91%
9	1.13	1.21	1.20	1.26	2.19	4.60%	0.67%
10	1.13	1.21	1.20	1.27	2.07	4.78%	0.49%

Graph performance is measured as speedup relative to the baseline case of no kernel fusing and 64x48 tiles for all kernels, as shown in Eq. 4.1

$$speedup = \frac{\begin{array}{c}Number\ of\ clock\ cycles\ to\ execute\ the \\ baseline\ graph\ with\ NO\ fused\ groupings\end{array}}{\begin{array}{c}Number\ of\ clock\ cycles\ to\ execute \\ the\ graph\ with\ fused\ groupings\end{array}} \tag{4.1}$$

Using Eq. 4.1, we associate a predicted speedup with each graph configuration, which defines the kernel groupings given by a graph decomposition and associated tile size for each group.

Table 4.3 shows performance and model accuracy for graphs of size 2 to 10 nodes, for at least 500 random graphs of each size. Note that fully enumerating the graph configurations for graphs with more than 10 nodes in the EVM is not feasible due to the growth of the enumeration space.

The "Average Best EVM speedup" column indicates, for each graph size, the average best speedup achieved for all graphs by selecting a configuration giving the best performance for each graph when run on the hardware. In other words, this represents the speedup a user would expect if it were feasible to evaluate all graph configuration enumerations on the hardware.

Since this is generally not feasible, especially for larger graphs, the "Average Prognosticated Speedup" column shows the expected speedup a user would obtain on the hardware if all graph configuration enumerations are evaluated using the performance model and choosing the one that maximized the expected speedup. When compared to the previous column, this indicates how the model accuracy affects the overall optimization problem of selecting the best graph configuration. In other words, in cases where the model does not exactly identify the best performing graph on the hardware (due to prediction error), it is still able to select a closely-performing graph.

Likewise, the "Average Model Predicted Speedup" is the expected speedup a user would obtain according to the performance model if all graph configuration enumer-

ations are evaluated using the performance model. This column, when compared to the previous column, indicates the overall accuracy of the performance model.

The column labeled "Highest EVM Speedup" shows the speedup of the highest-performing graph among all those tested for each graph size, showing the maximum potential for speedup using kernel fusing. The column labeled "EVM vs Model Error %" shows the average model error when predicting the speedup given by the best configuration of all graphs of each size. The "EVM vs Prognosticated Error %" shows the relative difference in performance between selecting the best-performing configuration based on hardware performance or model predicted performance.

As shown in the table, speedup from kernel fusion and tile size selection as well as the model accuracy, remain relatively consistent across graph sizes, with a slight reduction in performance as the graph size is increased. Generally speaking, a user can expect a speedup of approximately 1.20x for graphs of size nine or larger by automatically configuring a graph using the proposed performance models.

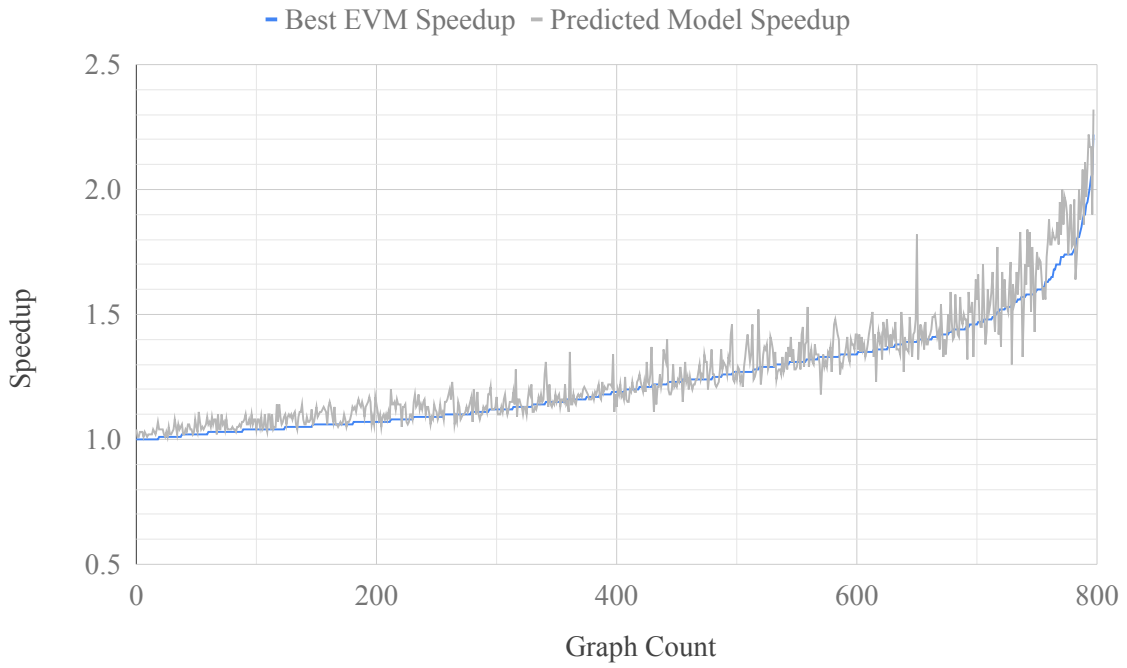

Figure 4.5 Comparison of graph speedup from the EVM vs our model for 800 graphs of 8-nodes. Speedups are sorted based on EVM speedup and it correlates to model speedup.

Figure 4.5 shows, for a set of random 8-node graphs, the distribution of the best speedup achieved by the optimal graph configuration for each graph on the hardware and the corresponding best speedup achieved by the optimal graph configuration for each graph according to the performance model .

The plot is sorted in ascending order of best hardware speedup. The median speedup is 1.19 and the 90th percentile is approximately 1.5. These results show that the speedup attainable for a given graph depends on characteristics of the graph. On average, the model predicts the best attainable speedup for each graph to within 3.69%, although as shown in Table. 4.3 the model accuracy varies with graph size.

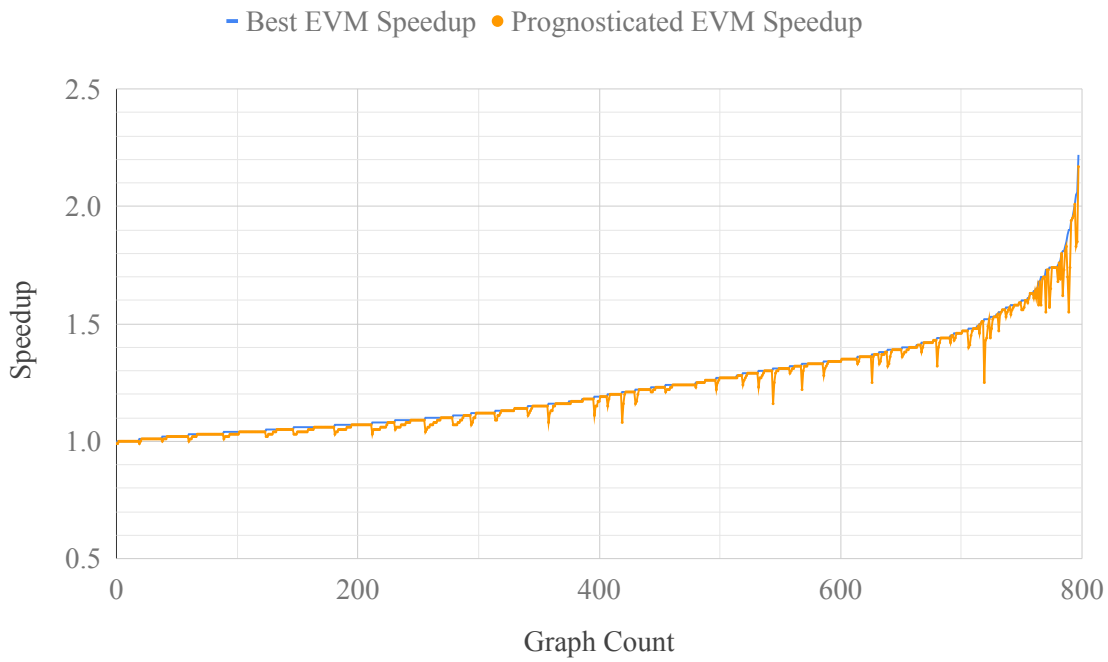

Figure 4.6 Comparison of graph speedup from the EVM vs our prognosticated speedup for 800 graphs of 8-nodes. Speedups are sorted based on EVM speedup and it correlates to the prognosticated speedup.

Figure 4.6 shows a similar distribute to that of Fig. 4.5 but in this case the best attainable hardware speedup is compared to the speedup obtained on the hardware if the graph configuration chosen was that of the configuration with the best speedup as measured by the performance model. In other words, this later speedup represents

the actual speedup when the graph configuration is chosen using only the model for guidance. In this case, there are some graphs where model inaccuracy led to the non-optimal graph configuration not being selected, but this only accounts for 0.91% slowdown on average across all the graphs.

Note that evaluating all graph configurations on the hardware is not feasible for larger graphs due to the time required to generate, compile, deploy, execute, and measure all graph configurations. On the other hand, evaluating the performance model for each graph enumeration requires a trivial amount of time.

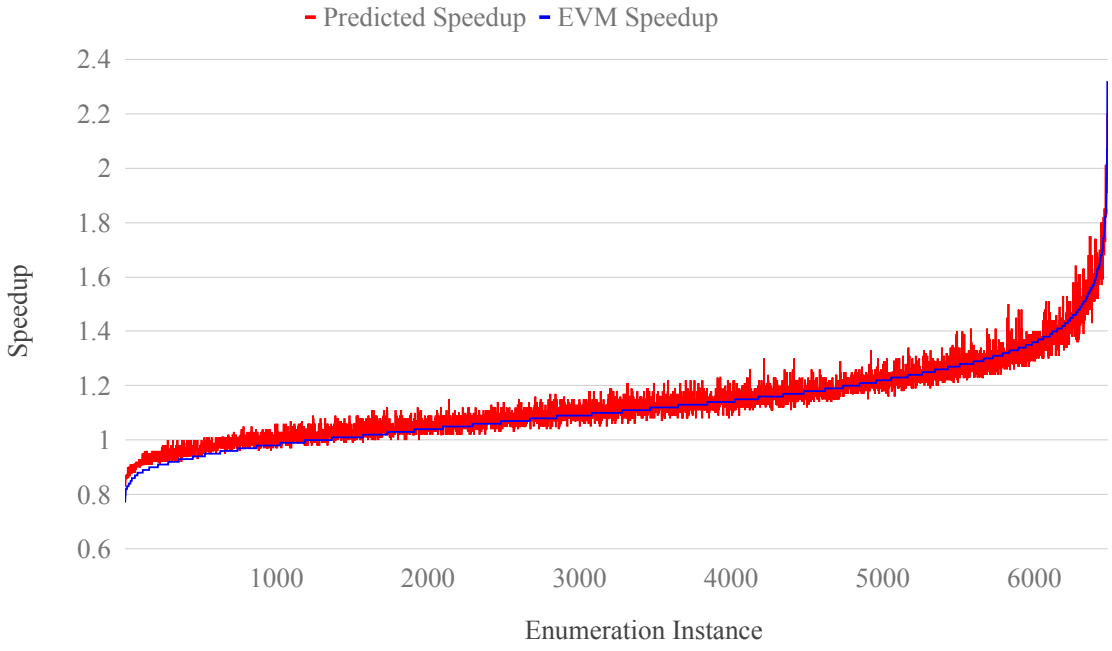

Figure 4.7 Distribution of speedup over enumerated parameterizations of a randomly selected 5-node graph. Speedup data was collected by running the graph in the EVM.

Fig. 4.7 shows the distribution of actual and predicted speedups over all 6,480 graph configurations for a randomly-chosen 5-node graph. The best predicted speedup achieved with this graph is 2.32, but less than 2.7% of the graph configurations achieve a actual speedup of greater than 1.5. This result illustrates the rarity of good graph configurations.

Fig. 4.8 compares the average optimal configuration actual speedup to the actual

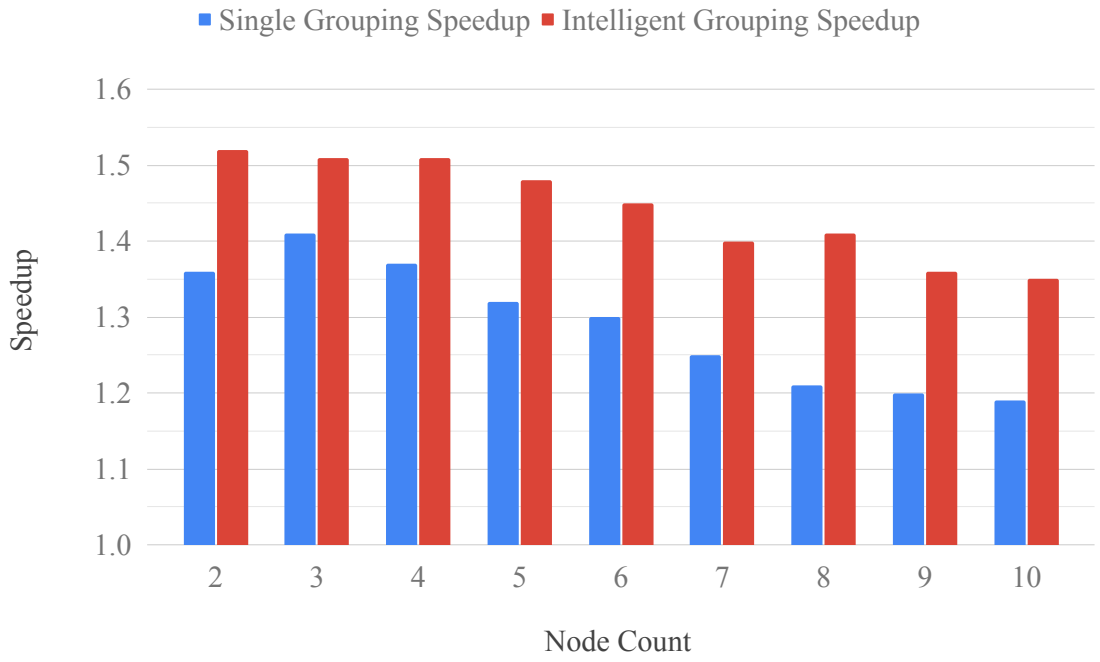

Figure 4.8 Comparison between actual speedup from optimal node fusing vs fusing all nodes. Each speedup shown for graph sizes of up to 10 nodes are computed by averaging speedup results for at least 100 unique graphs.

speedup achieved from fusing all nodes into a single group with a 64x48 speedup. Since it is not possible to fuse all nodes in every graph due to software limitations, the data in this figure only includes those graphs in which all their nodes can be fused. For this reason, this dataset is slightly different from the one presented in Table 4.3).

Speedups for the single grouping case range from 1.36 down to 1.19, while the optimal configuration speedups range from 1.52 to 1.35. This result shows that grouping all the nodes will imposes a 10% - 15% performance penalty relative to the maximum possible speedup.

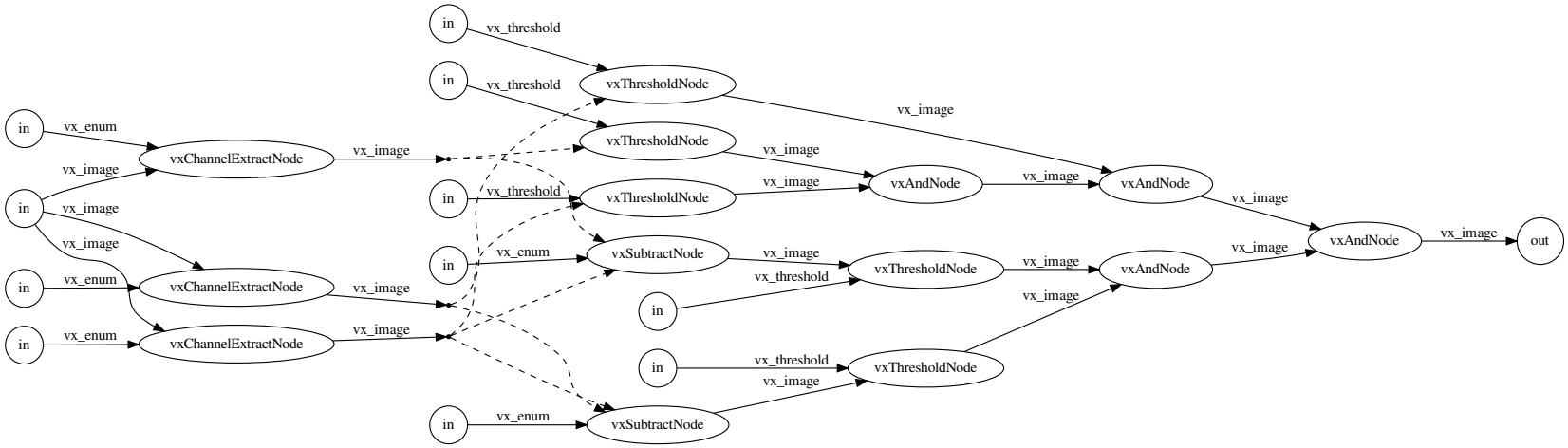

Figure 4.9 OpenVX graph for Skin Tone Detection.

4.2.2 Practical Benchmark Application

Since our prior results are based on synthetic graphs, we also evaluated the proposed fusing method on a practical application, shown in Fig. 4.9. Skin tone detection is a practical application which can identify a person's skin on an image or a video stream. This application can be built using a 14 node OpenVX graph which includes 3 vxChannelExtract nodes, 2 vxSubtract nodes, 5 vxThreshold nodes, and 4 vxAnd nodes.

The optimal configuration for this graph as predicted by our models has 3 groups. First one vxChannelExtractNode, two vxThresholdNodes, one vxSubtractNode and one vxAndNode were fused together and the tile size was set to 64x48. Second, two vxChannelExtractNodes, one vxSubtractNode, one vxThresholdNode and one vxAndNode was fused together with a tile size of 48x36 and finally two vxThreshodNodes and two vxAndNodes were fused together and the tile size was set to 64x48. This configuration achieved a speedup of 2.02 compared to the standard OpenVX graph with no fused kernels. The best grouping enumeration proposed by our models are shown in Fig. 4.10.

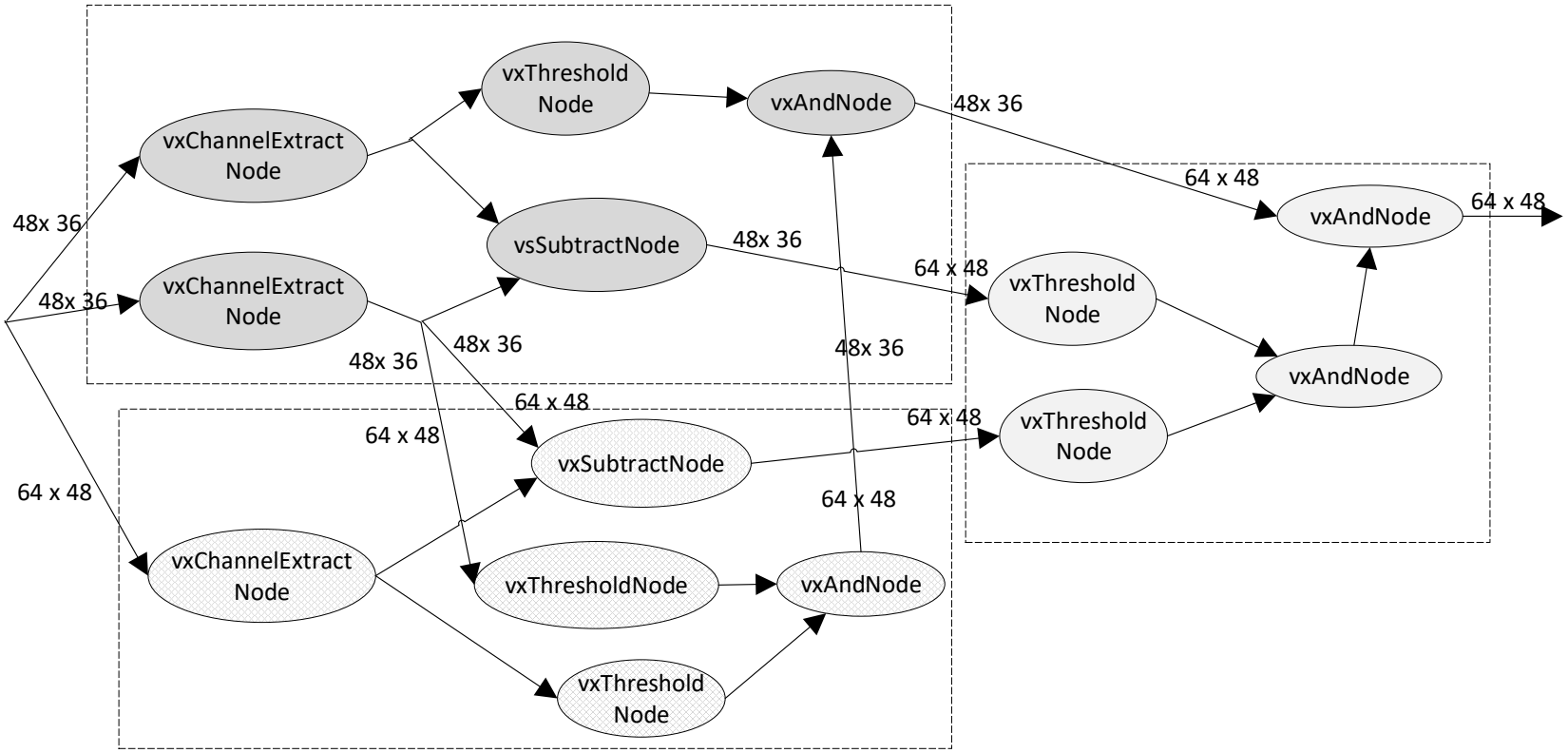

Figure 4.10 Best enumeration for skin tone detection graph chosen by our models

Chapter 5

Kernel Loop Fusion

In image processing applications, input images are often subjected to a progressive sequence of transformations. For example, an incoming image might be converted to another color space, then have color channels extracted, then be added to another image, then filtered with several convolutions such as blurring and edge extraction, and then compute feature vectors for each pixel. These transformations are often available as separate functions in an image processing library. In our case, we target the Texas Instruments VXLIB library [23].

In this scenario, each successive transformation may be performed with a separate loop but with the same number of loop iterations. On a DSP, each loop would read the input data from an memory hierarchy buffer and store the output data in another memory hierarchy buffer that would be read by a downstream loop.

Fusing any pair of loops where one produces data consumed by another allows two loops to be replaced by a single loop and avoids all the memory transactions that would otherwise be needed to convey the intermediate image between the loops. In other words, the intermediate data would be transferred through registers rather than through memory hierarchy buffers.

Fusing loops in this way improves performance through several mechanisms. First, the fused loop, having more workload, can often exploit more on-chip resources, i.e. achieve higher functional unit utilization. Second, the store and load instruction pairs that would otherwise be needed to store intermediate results to memory hierarchy would be eliminated, reducing the total dynamic instruction count. Finally, reducing

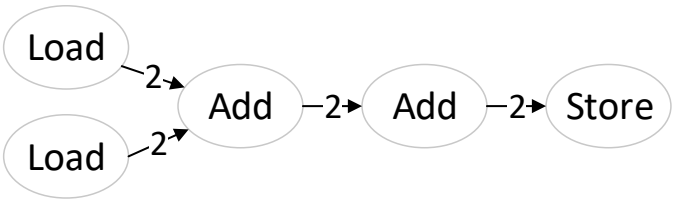

Figure 5.1 Data flow graph for simple loop that loads two values, adds them, increments the sum, and stores the result. The instruction latency is shown on the edges.

memory hierarchy access will correspondingly reduce dynamic stalls resulting from cache misses. Fusing loops also improves energy efficiency by reducing memory system utilization.

Choosing which loops to fuse is a challenge due to several practical considerations. First, maximizing functional unit utilization requires that the loops to be fused have a complementary set of instructions, allowing instructions from one loop to fill empty slots that would otherwise be present in the other loop's schedule.

Second, optimally statically scheduling loops is an NP-complete problem and is thus sensitive to the size of the loop body. However, typical heuristics that are used to software pipeline loops on VLIW architectures have compile times that increase by at least the square or cube of the number of nodes in the scheduling graph. Therefore, to avoid excessive compile time, the TI compiler limits modulo scheduling to loops that have fewer number of instruction nodes than an experimentally determined threshold. Above this threshold, the compiler will simply schedule the loop without modulo scheduling, leading to poor utilization of the available functional units, and thus poor execution performance.

Third, the size of the register file is a constraint for pipeline scheduling, which can lead to reduced performance for certain combinations of loop bodies.

In order to illustrate how loop fusion can lead to speedup after modulo scheduling, consider the following example. Assume a hypothetical processor with one load/store unit and one arithmetic unit. Further, assume the following latencies: 'load' instruc-

Loop Body:
load a
load b
c=a+b
store c

Schedule Details:
II=3
IL=6
IP=2
U=8/12

Piped Loop Cycle	Single Scheduled Iteration Cycle	Load/Store Unit Instruction	Load/Store Unit Iteration	Arithmetic Unit Instruction	Arithmetic Unit Iteration
0	0	load a	0	c=a+b	-1
1	1	load b	0		
2	2	store c	-1		
0	3	load a	1	c=a+b	0
1	4	load b	1		
2	5	store c	0		

Figure 5.2 Modulo schedule for a loop that loads two values, adds them, then stores the result.

tion requires 2 cycles, 'store' instruction requires 1 cycle, and arithmetic operations (such as 'add') requires 2 cycles.

Fig. 5.2 shows a loop body that loads registers 'a' and 'b', computes their sum, then stores the sum back to memory hierarchy.

For this loop and processor, the Minimum II as constrained by resources, or $ResMII$, is computed by counting the number of instructions in the loop body corresponding to each functional unit, dividing each by the corresponding number of functional units, and finding the maximum. In this case, since there are three load/store instructions and only one arithmetic instruction, $ResMII = max(\lceil \frac{3}{1} \rceil, \lceil \frac{1}{1} \rceil) = 3$.

The minimum iteration latency (IL) of the loop is the combined latency of the load, add, and store instructions, which is 5. Thus, to achieve an initiation interval (II) of 3, there must be at least $\lceil \frac{IL}{II} \rceil = \lceil \frac{5}{3} \rceil = 2$ iterations in parallel.

In the schedule shown in Fig. 5.2, the column 'piped loop cycle' refers to the cycle relative to the piped kernel, which is the code that is physically executed on the DSP. This example schedule shows two iterations of the piped kernel. The column labeled single scheduled cycle is the cycle relative to the single scheduled iteration, which is a representation of the original, non-pipelined loop, and is executed over the course of multiple piped kernel iterations.

Note that once the instruction scheduler places 'load a' and 'load b' in 'single

Loop Body:
load a
b=a+1
store b

Schedule Details:
II=2
IL=6
IP=3
U=9/12

Piped Loop Cycle	Single Scheduled Iteration Cycle	Load/Store Unit Instruction	Load/Store Unit Iteration	Arithmetic Unit Instruction	Arithmetic Unit Iteration
0	0	load a	0	b=a+1	-1
1	1	store b	-2		
0	2	load a	1	b=a+1	0
1	3	store b	-1		
0	4	load a	2	b=a+1	1
1	5	store b	0		

Figure 5.3 Modulo schedule for a loop that loads a value, increments it, and stores the result.

scheduled cycle' 0 and 1 respectively, the next instruction 'c = a + b' can be placed in 'single scheduled cycle' 3 (pipelined loop cycle 3 $\mod II = 0$) because of the two cycle delay in loading 'b'. Then 'store c' can be placed in 'single scheduled cycle' 5 because of the two cycle latency required by the add instruction.

The schedule's resultant functional unit utilization is only $\frac{8}{12} = 66\%$, since only 8 of the 12 functional unit slots are occupied in the schedule.

Consider another loop that only loads one value, increments it, and stores the result back to memory. This loop has $ResMII = max(\lceil \frac{2}{1} \rceil, \lceil \frac{1}{1} \rceil) = 2$ and minimum iteration latency (IL) of 5, also requiring $\lceil \frac{IL}{II} \rceil = \lceil \frac{5}{2} \rceil = 3$ iterations in parallel to achieve its minimum II of 2, as shown in Fig. 5.3 and achieving a functional unit utilization of $\frac{9}{12} = 75\%$.

Assuming both loops would normally be executed in sequence, fusing the loops adds an additional 2 cycles to the minimum iteration latency (IL), caused by the addition of the increment operation that depends on the previous sum operation. However, fusing these loops allows the store instruction from the first loop and the load instruction from the second loop to be eliminated, since the intermediate result (variable c) can be allocated in a register. The resultant loop body has three load/store instructions and two arithmetic instructions, with $ResMII = max(\lceil \frac{3}{1} \rceil, \lceil \frac{2}{1} \rceil) = 3$, $IL = 2 + 2 + 2 + 1 = 7$, and minimum iterations in parallel $= \frac{IL}{II} = \frac{7}{3} = 3$.

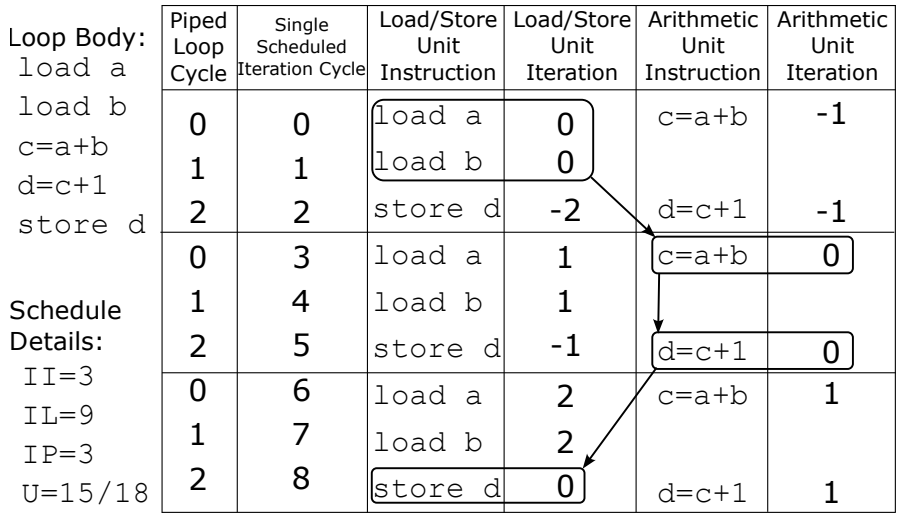

Figure 5.4 Modulo Schedule for the fused loop from Figs. 5.2 and 5.3.

A schedule for the fused loop is shown in Fig. 5.4. It has a resource utilization of 15/18 (18 slots available for 9 cycles with 2 functional units each and 15 occupied slots). The throughput of the fused loop is $\frac{1}{II} = \frac{1}{3}$ and must be compared the combined throughputs of the two constituent loops, computed as $\frac{1}{II_1+II_2} = \frac{1}{5}$. The resultant speedup in terms of throughput achieved from the fusing is this $\frac{1/3}{1/5} = 1.66$. The DAG for this loop is shown in Fig. 5.1.

For the fused loop, as shown in Fig. 5.4 the II is 3, iteration latency (IL) = 9 and iterations in parallel (IP) = 3. It is evident that by fusing the 2 loops, the II did not increase but the functional unit utilization was increased to 15/18 which means the fused loop is doing more work within the same clock cycles as the first loop from Fig. 5.2. So by fusing these two loops we have achieved a speedup of (3 + 2) / 3 = 1.66 in addition to eliminating the memory hierarchy accesses by removing the intermediate load/store instructions.

5.1 Proposed Approach

We have developed a framework that accepts an image processing dataflow graph and generates a corresponding C++ code with explicit loop fusing that, when compiled

with the TI DSP compiler, produces near-optimal code.

This framework is comprised of three main components: (1) a graph generator that enumerations all possible decompositions of an input VXLIB graph to identify which kernels in the dataflow graph to fuse to maximize performance, (2) HeRCide: a C++ code generator and corresponding target library that takes an input graph comprised of VXLIB kernels and generates a single loop encompassing the functionality of multiple fused VXLIB kernels as executable code, and (3) VXOPT: a heuristic modulo scheduler and a library of assembly-level single scheduled iterations corresponding to each VXLIB kernel.

5.1.1 Graph Generator

For a given image processing application described as a graph of VXLIB kernels, our objective is to find the graph decomposition–that is, the assignment of each kernel to a kernel group–that when each group of kernels is synthesized as a single loop and scheduled, achieves the best overall speedup over the baseline case of executing each kernel as a separate loop.

The upper bound for the maximum number of decompositions is defined by the Bell number, defined in Eq. 1.1, which is an exponential function. In practice, since each kernel group must be weakly connected in order for the grouping to allow for the elimination of load and store instructions corresponding to the values conveyed between grouped kernels, the total number of decompositions is potentially small enough to be enumerated and evaluated using a fast performance model, but too large to evaluate on the DSP.

Our graph generator enumerates all weakly connected subgraphs for a given application and estimates the resultant speedup against a performance model that uses a heuristic to compute a modulo schedule for the fused loop. The heuristic scheduler contains an instruction-level dataflow graph for each VXLIB kernel. When the opti-

mal decomposition is found, the resulting fused loops are composed in C++ using a template library called HeRCide.

5.1.2 HeRCide

HeRCide is designed as a DSP-centric analog to Halide [45], is a hierarchical C++ library containing a PixelLib, a template function corresponding to the single iteration workload associated with each VXLIB kernel. As such, a fused loop can be generated by instancing calls to these methods in a single loop body. Since the C66x DSP contains Single Instruction, Multiple Data (SIMD) support, the PixelLib is built on top of a SIMD datatype library called the "core functions" library, which contains assembly-level implementations of primitive operations for SIMD types.

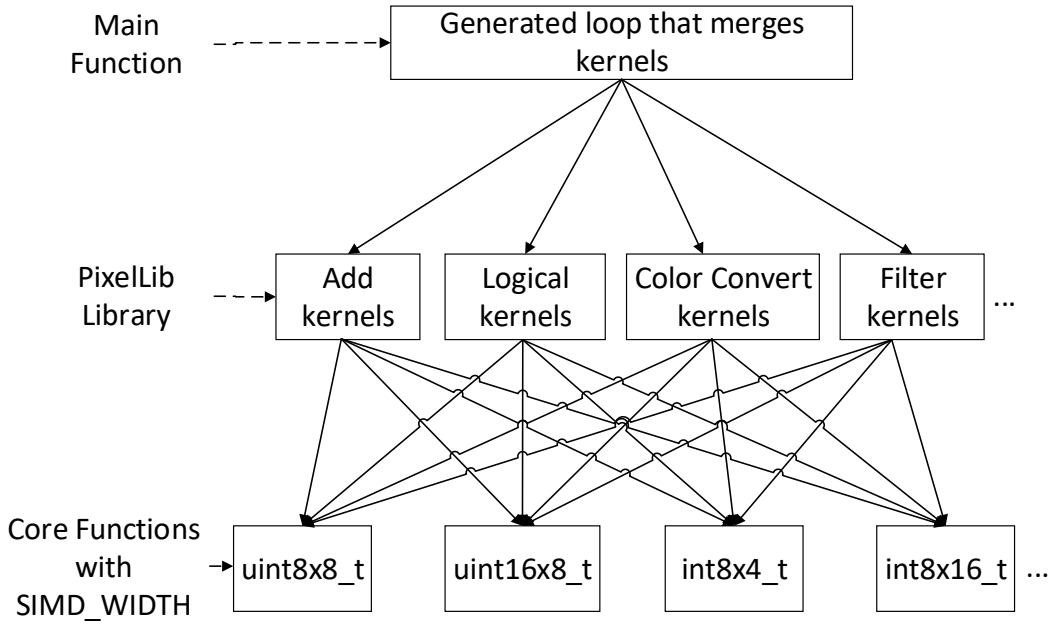

Figure 5.5 HeRCide Architecture

This stack is shown in Fig. 5.5, in which a synthesized fused loop is comprised of one call to the PixelLib library for each kernel being fused, and the PixelLib functions having template parameters corresponding to whatever primitive SIMD type is being processed by each kernel.

This approach requires that the kernels being fused have the same number of loop iterations, which consequently requires that all the included kernels use the same SIMD width.

Code 2 Example type library of "Core Funcions" for uint8x8_t type

```
class uint8x8_t {
public:
  uint64_t data;
  uint8x8_t(uint64_t val=0) : data(val) {}
  uint8x8_t() {}
  inline uint8x8_t operator= (const uint8_t * restrict input) {
    data = _amem8_const(input);
    return uint8x8_t(data);
  }
  inline uint8x8_t operator| (const uint8x8_t op1) {
    uint64_t t0 = data | op1.data;
    return result(t0);
  }
  inline uint8x8_t operator+ (const uint8x8_t op1) {
    uint64_t t0;
    t0=_itoll(_add4(_hill(data), _hill(op1.data)),
              _add4(_loll(data), _loll(op1.data)));
    return uint8x8_t(t0);
  }
  inline void store (const uint8_t * restrict output) {
    _amem8((void *)output) = data;
  }
};
```

An example HeRCide core function class for the 8-element 8-bit unsigned SIMD type is shown in Code. 2. It implements the primitive "bitwise-or" and "add" functions. Note that the operators are overridden and include processor-specific vectorized intrinsics that map to specific SIMD instructions.

An example set of PixelLib routines is shown in Code 3. Code. 4, shows an example loop generated from fusing the "VXLIB_Add" kernel and "VXLIB_Or" kernel. This code, which comprises the top-level loop and several compiler pragmas, is machine-generated based on the kernels and interconnections given by a graph-

Code 3 PixelLib library code for 'add' and 'or' kernels

```
template <typename simd_width>
static inline
void pixellib_add(simd_width src0,
                  simd_width src1,
                  simd_width &dst) {
    dst = src0 + src1;
}

template <typename simd_width>
static inline
void pixellib_or(simd_width src0,
                 simd_width src1,
                 simd_width &dst) {
    dst = src0 | src1;
}
```

based input description.

The VXLIB_Add kernel has two external inputs named src1 and src2 and the output is connected to one of the inputs of the VXLIB_Or kernel, with the other connected to an external input named src3. The output of the VXLIB_Or kernel is an external output.

The "_nasssert" statement indicates to the compiler that the arrays are aligned and can be accessed with load and store instructions that require aligned addresses. The "UNROLL" pragma tells the compiler exactly how many times to unroll the loop and the "MUST_ITERATE" pragma indicates minimum and maximum trip counts. Within the kernel "for loop" customized SIMD load instructions are generated from the overloaded assignment (=) operators.

Calls to the PixelLib library functions and their corresponding inputs and outputs are connected according to the graph structure shown in Fig. 5.6. The output is stored using the store function call which includes a SIMD store intrinsic.

The set of VXLIB kernels currently supported by HeRCide is presented in Table. 5.1.

Code 4 Generated code for fused 'add' and 'or' kernels

```
void vxAddOrKernel(
  const uint8_t * restrict src1,
  const uint8_t * restrict src2,
  const uint8_t * restrict src3,
  uint8_t * restrict dst,
  uint32_t width) {

  uint8x8_t src1a, src2a, src3a, dsta, t0;
  _nassert(((uint32_t)src1 % 8U) == 0);
  _nassert(((uint32_t)src2 % 8U) == 0);
  _nassert(((uint32_t)src3 % 8U) == 0);
  _nassert(((uint32_t)dst % 8U) == 0);

  #pragma UNROLL(1)
  #pragma MUST_ITERATE(0,,2)
  for (int x=0; x<width; x++) {
    #pragma FORCEINLINE_RECURSIVE
    src1a = &src1[x*8];
    #pragma FORCEINLINE_RECURSIVE
    src2a = &src2[x*8];
    #pragma FORCEINLINE_RECURSIVE
    src3a = &src3[x*8];

    #pragma FORCEINLINE_RECURSIVE
    pixellib_add <uint8x8_t> (src1a, src2a, t0);
    #pragma FORCEINLINE_RECURSIVE
    pixellib_or <uint8x8_t> (t0, src3a, dsta);

    #pragma FORCEINLINE_RECURSIVE
    dsta.store(&dst[x*8]);
  }
}
```

5.1.3 VXOPT

Fusing kernels has the potential to achieve substantial speedup for VXLIB graphs, but identifying sub-graphs to fuse requires a model to predict both the feasibility and performance impact of a proposed fusing strategy.

Certain combinations of kernels cannot be fused because the vendor's DSP compiler will fail to modulo schedule loops in which the hardware constraints make it

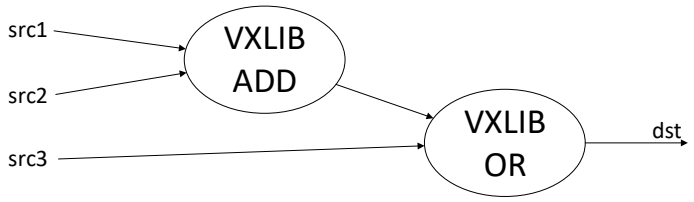

Figure 5.6 Graph structure for the merged add-or kernels

Table 5.1 VXLIB Kernels supported in HeRCide

Logical Kernels	Arithmetic Kernels	Misc Kernels
or_i8u_i8u_o8u	absDiff_i8u_i8u_o8u	colConvert_RGBtoYUV4
and_i8u_i8u_o8u	addWeight_i8u_i8u_o8u	channelExtract_1of[2|3|4]
xor_i8u_i8u_o8u	addSquare_i8u_i16s_o16s	channelCombine_[2|3|4]*to*1
not_i8u_o8u	add_i[8|16][*u*|*s*]_*i*[8|16][*u*|*s*]_*o*[8|16][*u*|*s*]	convertDepth_i[8|16][*u*|*s*]_*o*[8|16][*u*|*s*]
	subtract_i[8|16][*u*|*s*]_*i*[8|16][*u*|*s*]_*o*[8|16][*u*|*s*]	threshold_i[8|16][*u*|*s*]_*o*[8|16][*u*|*s*]
	multiply_i[8|16][*u*|*s*]_*i*[8|16][*u*|*s*]_*o*[8|16][*u*|*s*]	tableLookup_i[8|16][*u*|*s*]_*o*[8|16][*u*|*s*]

difficult or impossible to find a valid schedule having more than one iteration in parallel. The most common cause of this is when there is an insufficient number of registers to hold all the needed live values or when the iteration latency must be increased to accommodate the schedule to the point where the II becomes equal to the latency of the single scheduled iteration. In other words, it is not feasible to fuse a large set of kernels.

Even for kernel sets that can be fused, kernel combinations that have complementary functional unit requirements or ones that are tightly interconnected will achieve a greater speedup than combinations that lack these qualities.

Thus, in order to maximize speedup through kernel fusing, it is necessary to have knowledge of which kernel combinations can be fused and which kernels combinations when fused give the best speedup.

For this reason, we propose a performance model that will take, as input, a set of kernels and predict the initiation interval that the compiler would achieve for a loop that contains the combined workload of the fused kernels, minus the load and store instructions that would have been required to convey intermediate results if the kernels were not fused. Ideally, this model would be substantially faster than

compiling the fused loop. For this reason, we have developed a heuristic scheduler that approximates the modulo scheduling algorithm.

5.2 VXLIB Graph Optimization

For a given set of VXLIB kernels and associated graph, our performance model will merge the corresponding instruction-level dataflow graphs of each kernel, compute the As Soon as Possible (ASAP) and As Late as Possible (ALAP) cycles for each instruction, construct a modulo schedule for a loop containing the kernels, and report the resulting initiation interval, schedule, and register usage table. The reported II is used to estimate the achieved throughput of the fused VXLIB kernels and is compared against other candidate merging strategies.

Our scheduler is substantially faster (30 to 60x) than the TI compiler because it does not require C/C++ parsing, it does not perform register allocation or code generation, and uses a heuristic scheduler. For these reasons, it is not guaranteed to produce results that are consistent with the TI compiler, but is intended to deliver sufficient accuracy to significantly outperform a random fusing strategy.

5.2.1 Merging DAGs

The single scheduled iteration of each VXLIB kernel is stored in a database and, when used, converted to a DAG in which each vertex corresponds to each assembly instruction and each edge represents a dependency between two instructions. To fuse two or more VXLIB kernels that are weakly connected in a VXLIB graph, their corresponding DAGs are combined by matching the edges in the VXLIB graph to the corresponding load and store instructions in the assembly-level DAGs for each kernel.

In other words, each external input and output from each VXLIB kernel is associated with a load or store instruction, respectively. When fusing two kernels in which kernel A's output is connected to the kernel B's input, the corresponding store in-

struction in kernel A and the corresponding load instruction in kernel B are removed and replaced with a single DAG edge.

Fig. 5.7c shows an example VXLIB graph in which the VXLIB absDiff kernel fans out to two VXLIB OR kernels. Figs. 5.7a and 5.7b shows the instruction-level DAGs for the absDiff and OR kernels, respectively.

When fusing these 3 kernels, the store instructions corresponding to the output of the absDiff are removed and the load instructions corresponding to the inputs of the OR kernels are removed. Fig. 5.7d shows the merged DAG for the resultant "absDiff-OR-OR kernel".

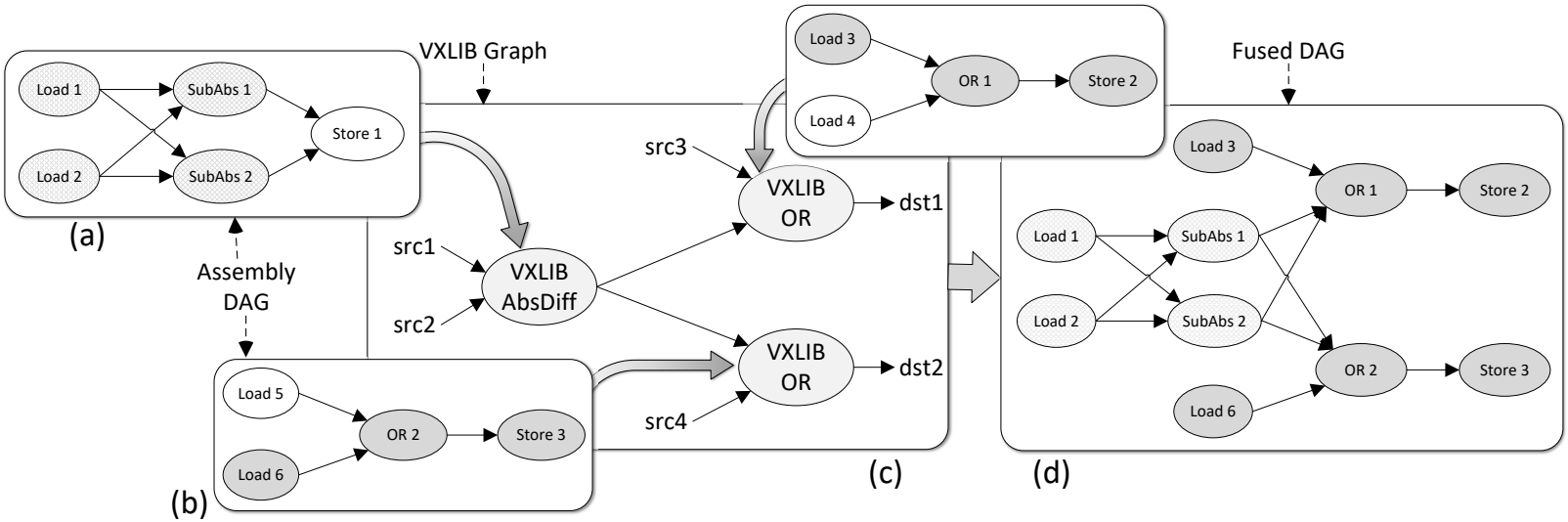

Figure 5.7 VXLIB graph for fused AbsDiff-OR-OR kernel

5.2.2 Performance Model Design

To identify the most efficient set of subgraphs in which to decompose a VXLIB graph, our proposed method performs a search over the decomposition space, evaluating each decomposition with our performance model. Our performance model uses a heuristic to schedule the fused DAG to determine the feasibility of scheduling and the resultant iteration interval (II).

A modulo scheduler must make several decisions when constructing a schedule, each of which represents a branching point in the search space. For example, many instructions can be scheduled on more than one functional unit type, for a given function unit, instructions can be scheduled on either the A-register path or the B-register path, and there may be a difference between the earliest and latest cycle in which an instruction may be scheduled. For this reason, finding an optimal schedule requires a combinatorial search. After the schedule is constructed, the compiler must allocate registers, the decisions of which comprise its own combinatorial space.

In our proposed performance model, we use a scheduling algorithm without a register allocation phase, which in practice produces sufficiently high quality of results to prune away virtually all unpromising VXLIB graph decompositions.

Schedule Initialization

The modulo schedule is made up of functional unit slots, the number of which is $II \times n_{units}$, where II = the initiation interval, or the inverse of the loop throughput. The number of units available in the C66x DSP is eight, namely L1, L2, D1, D2, S1, S2, M1, M2. Each unit can be issued one instruction per cycle. Each issued instruction performs workload on behalf of one of IP loop iterations, in which IP is the number of iterations processed in parallel. Recall that IP is a function of II and IL, where IL is the loop iteration latency as shown in Eq. 2.1.

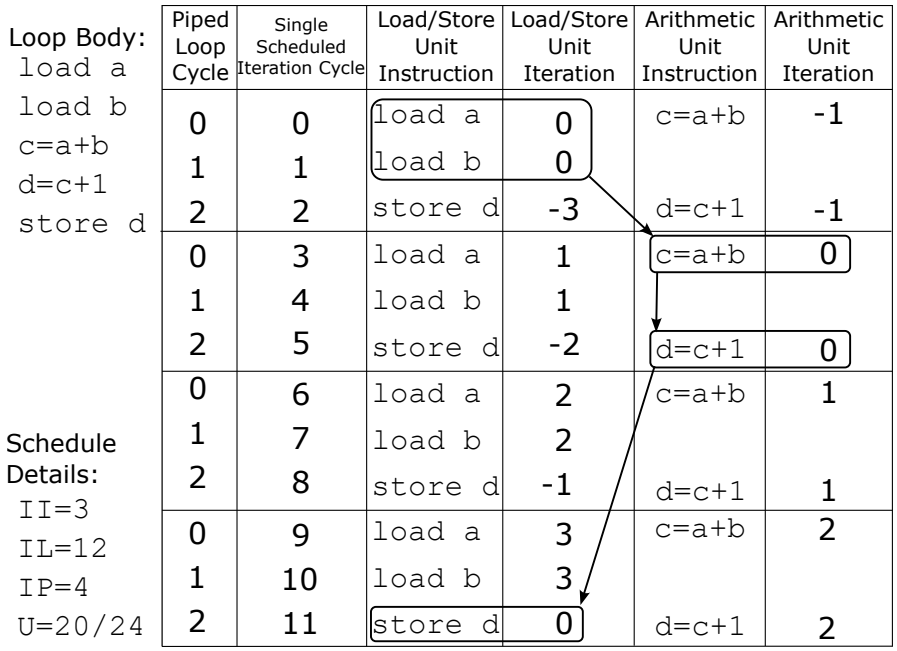

Figure 5.8 Modulo Schedule for the fused loop with an inferred register alive too long.

Register Allocation

The number of live registers in each of the II modulo cycles is an important constraint when scheduling, as the number of live registers cannot exceed 64. For each edge in the DAG connecting instructions a and b, when a is scheduled in modulo cycle a_{cycle} and relative iteration a_i for and instruction b_i is scheduled in modulo cycle b_{cycle} and relative iteration j, and given the latency of instruction a is $a_{latency}$, the number of live registers in cycles $((a_{cycle}+a_{latency})+a_i \times II) \mod II$ to $(b_{cycle}+b_i \times II) \mod II$ is incremented by the number of 32-bit values conveyed on that edge.

The number of live registers in cycle c is incremented for each instruction scheduled n cycles prior to cycle c, where n is the latency of the instruction, and for which at least one of the instruction's successors is scheduled on or after cycle c.

Register Live-to-Longs

Additionally, a live register cannot persist for more than II cycles without being transferred via a move instruction to another register, since each live register can only

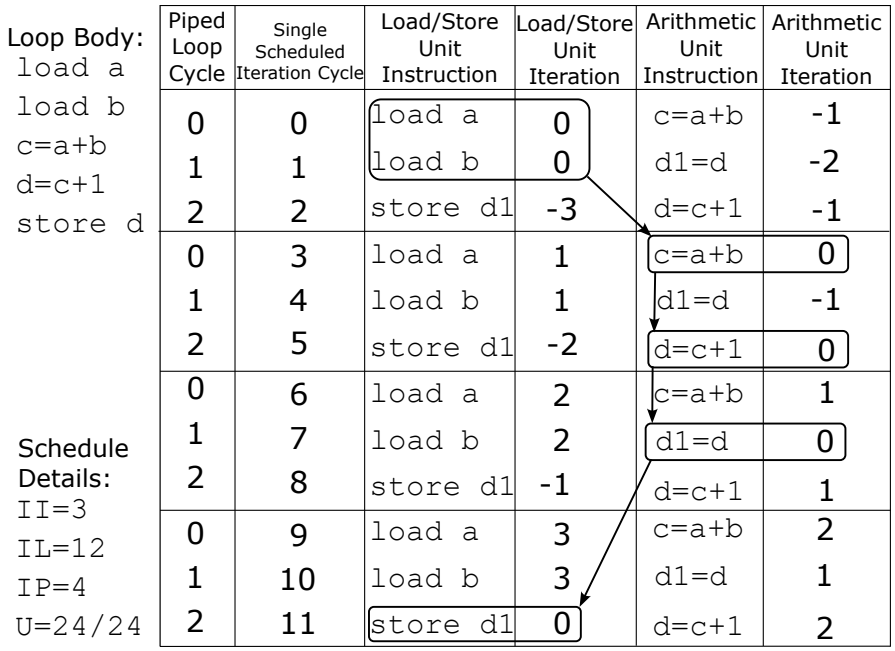

Figure 5.9 Modulo Schedule for the fused loop where the register alive too long has been resolved.

be associated with one loop iteration at any moment. In other words, if an instruction consuming a value is scheduled more than II cycles away from an instruction which produces that value (also taking into account the latency of the producing instruction to account for when the register is written), a live-too-long situation results. Without intervention, the register written by the dependency will be invalidated before it can be used. Inserting an intervening move instruction will remedy this problem but the move instruction will occupy a scheduling slot.

Figs. 5.8 shows how the schedule from Fig. 5.4 could be alternatively generated in a way that includes a register live-too-long. In this case, the number of iterations in parallel has increased from 3 to 4. A register must be allocated to hold the value of the variable d when it becomes available two cycles after the "d=c+1" instruction that generates it. This register is read by the "store d" instruction four cycles later. This requires the register holding the value of d to be alive for 4 cycles, which is greater than the II of 3. This is a "register live-too-long" problem, because that register would be invalidated by the "d=c+1" instruction corresponding to the next

loop iteration 3 cycles later. Our proposed performance model detects registers that are alive too long, and attempts to repair by inserting move instructions to split the lifetime of the register that is live too long.

This is shown in Fig. 5.9, in which the "d1=d" instruction is inserted into an empty slot two cycles after the "d=c+1" instruction and four cycles prior to the "store d1" instruction that reads it. Assuming that latency of the move instruction is one cycle or more, the lifetime of the register allocated to the d1 variable is guaranteed to be less than or equal to the II of 3.

5.2.3 Heuristic Scheduler

Our heuristic algorithm is shown in Alg. 5. The algorithm takes as input the fused assembly instruction DAG. It begins by computing the resource minimum II ($ResMII$) as described in Section 2.2.2 and computes the "As Soon As Possible (ASAP)" and "As Late As Possible (ALAP)" cycle for each instruction in the DAG.

The ASAP cycle of each instruction is computed as the sum of instruction latencies corresponding to its longest chain of dependencies. The ALAP cycle for all store instructions is set to the maximum ASAP cycle among all the instructions that have no output (i.e. the terminating instructions). Finally, the DAG is traversed in reverse order, setting the remaining ALAP values to the minimum ALAP value of each instruction's successors, minus its instruction latency. The critical path consists of all instructions that have the same ASAP and ALAP values.

An example of this is shown for the DAG in Fig. 5.10a, under the simplifying assumption that all instructions have a latency of two cycles. In this case, all instructions are part of the critical path except for store2.

After computing $ResMII$ and the ASAP and ALAP cycles for each instruction, our heuristic first sorts the critical path instructions using $-ALAP$ as the sort key. Second, any pair of instructions A and B in which instruction A is a critical path

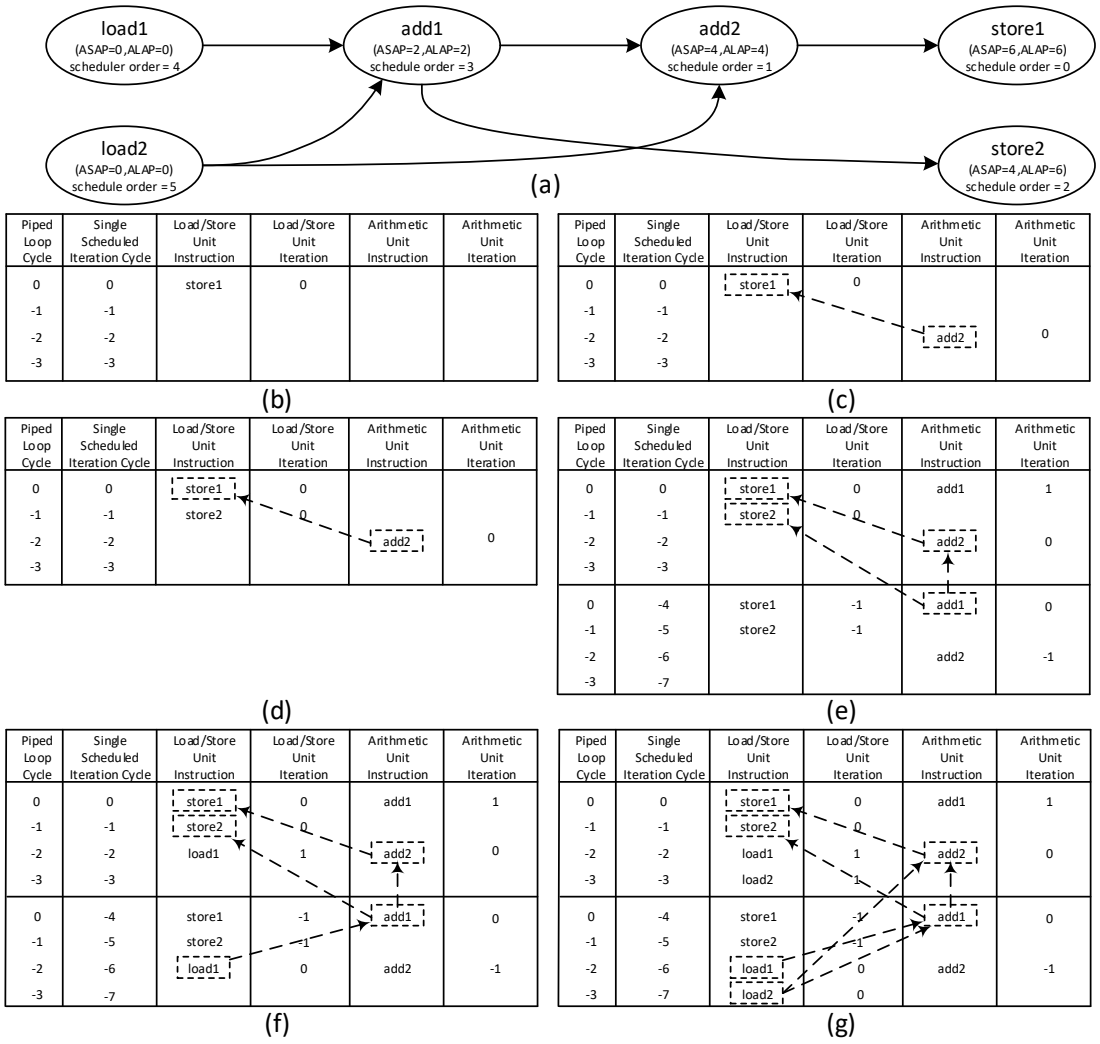

Figure 5.10 Example of scheduling heuristic.

instruction and instruction B is a non-critcal path instruction but a successor of A, the algorithm will insert instruction B before A in the sorted list.

Next, the non-critical path instructions are added to the tail of the sorted list using the edge distance to its nearest critical path instruction as the primary sort key and the ALAP value as the secondary sort key. In the example shown in Fig. 5.10, the resulting sorted order is store1, add2, store2, add1, load1, and load2.

The algorithm then allocates one slot for each functional unit for each of the $ResMII$ cycles. The schedule is constructed in reverse order by choosing and schedul-

ing instructions in sorted order. The cycle numbers are labeled in descending order starting from 0.

Scheduling Heuristic Example

For the example assembly instruction DAG shown in Fig. 5.10a, each instruction is labeled with its corresponding ASAP and ALAP cycles and the instructions are sorted in scheduling order as described above. The $ResMII$ for this DAG is 4 cycles, based on the four load/store instructions and the availability of only one load/store unit in the simplified architecture used for this example.

store1 is the first instruction to be scheduled. The scheduler can place this instruction in any cycle since it has no successor instructions. As shown in Fig. 5.10b, our heuristic allocates the instruction to cycle 0, the latest cycle in the schedule. This instruction is associated with loop iteration 0, which is meant to indicate a loop iteration number that is relative to the iteration number associated with the other instructions in the schedule.

Next, the heuristic schedules the add2 instruction–a dependency of the store1 instruction–which must be placed at least two cycles prior to the store1 instruction. Our heuristic places it in the latest possible position, cycle -2 as shown in Fig. 5.10c.

Next, as shown in Fig. 5.10d, the heuristic schedules the store2 instruction in the latest possible cycle in which the corresponding functional unit is available, -1, since it has no successors.

Next, as shown in Fig. 5.10e, the heuristic schedules the add1 instruction. The add1 instruction must be scheduled no less than two cycles earlier than its successors, add2 and store2, which are already scheduled in cycles -2 and -1, respectively. The latest cycle would therefore be -4. However, since this would require more than II cycles, the heuristic must place the add1 instruction in a cycle of another iteration of the piped kernel, pushing it into cycle 0 of the previous iteration. In the figure, both

iterations are shown in the schedule table in order to depict the flow of data across multiple iterations of the loop. Note that each iteration of the piped kernel must execute the same instructions, but intermediate results flow across loop boundaries. These are indicated by the dotted lines.

Next, as shown in Fig. 5.10f, the heuristic schedules the load1 instruction, which must be scheduled no later than two cycles earlier than its successor, add1, so the heuristic schedules it in the latest available cycle -2. Note that this instruction is placed in cycle 2 of each iteration, as shown in the schedule table.

Finally, as shown in Fig. 5.10g, the heuristic schedules the load2 instruction, which must be scheduled no later than two cycles earlier than its immediate successor, add1, so the heuristic schedules it in the latest available cycle -3.

5.2.4 Making Scheduling Decisions and Pruning the Search Space

As described in Sec. 5.2.2, the C66x DSP has eight functional units, two of each type, and many instructions can be scheduled on multiple function unit types. Additionally, each instruction can potentially be scheduled in II possible cycles, since the iteration number attached to the instruction can be used to meet dependency constraints.

Our scheduling heuristic uses a depth-first branch-and-bound search when searching for a schedule. For each instruction to scheduled, the heuristic recurses on every available cycle and every available and compatible functional unit.

For each candidate instruction placement, the algorithm performs a feasibility test to determine if there is a sufficient number of remaining functional unit slots to accommodate the unscheduled instructions. If not, the partial schedule is rejected and the search space under it is pruned, at which point the heuristic backtracks to the most recent decision point and selects a different unit and/or cycle for the last-scheduled instruction.

Additionally, each time an instruction is placed on the schedule, the heuristic

computes the number of live registers at each of the II cycles and prunes the partial schedule if any cycle exceeds a given register number threshold.

When expanding scheduling choices, the heuristic uses a cycle-priority, then unit-priorty order, meaning that it first attempts to schedule an instruction in the latest possible cycle for each unit type in a given order and then repeats this procedure for progressively earlier cycles.

If the search space is exhausted without finding a valid schedule for the $ResMII$, then the scheduler abandons the schedule, increases the II by one, and repeats the scheduling process. If the II reaches the latency of the single scheduled iteration, the algorithm reports a failure and exists.

After a schedule is found, the heuristic identifies any register live-to-longs and attempts to split the register lifetime by inserting a move instruction into the schedule. Move instructions can be scheduled on the L, S, or D units. If none of these units are available in the range of cycles required to split the register lifetime, the scheduler abandons the schedule increases the II and repeats the schedule process.

A simplified pseudo-code for this algorithm is shown in Alg. 5.

5.3 Results

Table 5.2 shows the potential speedup gained from fusing VXLIB graphs as well as the accuracy of our proposed performance model. Each row of the table summarizes the results from up to 500 random VXLIB graphs of the size given in column 1. These results were generated by fusing all the valid sub-graph decompositions for each of the randomly generated graphs using our "HeRCide" automated loop fusing library.

The column labeled "Average Number of Fusing Configurations" indicates the average number of ways the randomly generated graphs can be decomposed. Note that the number of decompositions depends on the graph structure since all sub-graphs in a decomposition must be weakly connected. Thus, graphs that are more

densely connected will generally have a greater number of decompositions.

The column labeled "Median Potential Speedup" indicates the median best speedup that can potentially be obtained from any fusing configuration. In other words, it is the speedup one may expect if it were known for each graph which decomposition achieves the best performance. Speedup is measured relative to the baseline performance given the graph without kernel fusing, as shown in Eq. 5.1.

Likewise, the column labeled "Average Potential Speedup" indicates, the average best speedup achieved for all graphs by selecting the set of sub-graphs achieving the best performance for each graph when evaluated for II for by the vendor compiler. In other words, this represents the speedup a user would expect if it were feasible to evaluate all sub-graph enumerations using the compiler.

$$speedup = \frac{\begin{array}{c} Sum\ of\ Initiation\ Intervals\ (II)\ for\ the \\ baseline\ graph\ with\ NO\ fused\ groupings \end{array}}{\begin{array}{c} Sum\ of\ Initiation\ Intervals\ (II)\ for \\ the\ graph\ with\ fused\ groupings \end{array}} \tag{5.1}$$

The column labeled "Average Prognosticated Speedup" shows the speedup achieved on average if a user selects the fusing configuration suggested by the proposed heuristic scheduler, as opposed to the best overall fusing configuration. Compared to the previous column, this indicates how the accuracy of the heuristic scheduler affects the overall optimization problem of selecting the best fusing configuration. In other words, in cases where the heuristic scheduler does not exactly identify the best performing configuration (due to prediction error), it is still able to select a closely performing configuration.

Likewise, the "Average Heuristic Predicted Best Speedup" is the average speedup reported by the heuristic scheduler. This column, compared to the "Average Potential speedup" column, indicates the overall accuracy of the heuristic scheduler.

The column labeled "Max Potential Speedup" shows the speedup of the highest-performing fused graph among all those tested for each graph size, showing the max-

imum potential for speedup using kernel loop fusing. The column labeled "Actual vs Heuristic Error %" shows the average heuristic scheduler error when predicting the speedup given by the best sub-graph of all graphs of each size. The "Actual vs Prognosticated Error %" shows the relative difference in performance between selecting the best-performing configuration based on performance given by the compiler or the heuristic scheduler.

Table 5.2 Comparison of average actual speedup values vs heuristic predicted speedup values for up to 500 graphs per each node count

Node Count	Average Number of Fusing Configurations	Median Potential Speedup	Average Potential Speedup	Average Prognosticated Speedup	Average Heuristic Predicted Best Speedup	Max Potential Speedup	Actual vs Heuristic Error %	Actual vs Prognosticated Error %
2	2	1.33	1.48	1.48	1.55	4.00	7.64%	0.50%
3	4	1.60	1.64	1.62	1.72	3.00	6.70%	1.13%
4	8	1.67	1.71	1.69	1.81	3.00	6.56%	2.07%
5	18	1.71	1.80	1.76	1.91	3.33	7.89%	3.25%
6	41	1.70	1.81	1.74	1.92	3.50	9.14%	4.63%
7	85	1.73	1.84	1.79	1.98	4.00	8.65%	5.97%
8	208	1.69	1.84	1.72	1.96	5.00	10.40%	9.55%
9	483	1.74	1.84	1.67	1.98	6.33	10.46%	9.24%
10	1216	1.71	1.80	1.64	1.97	4.25	9.92%	9.99%

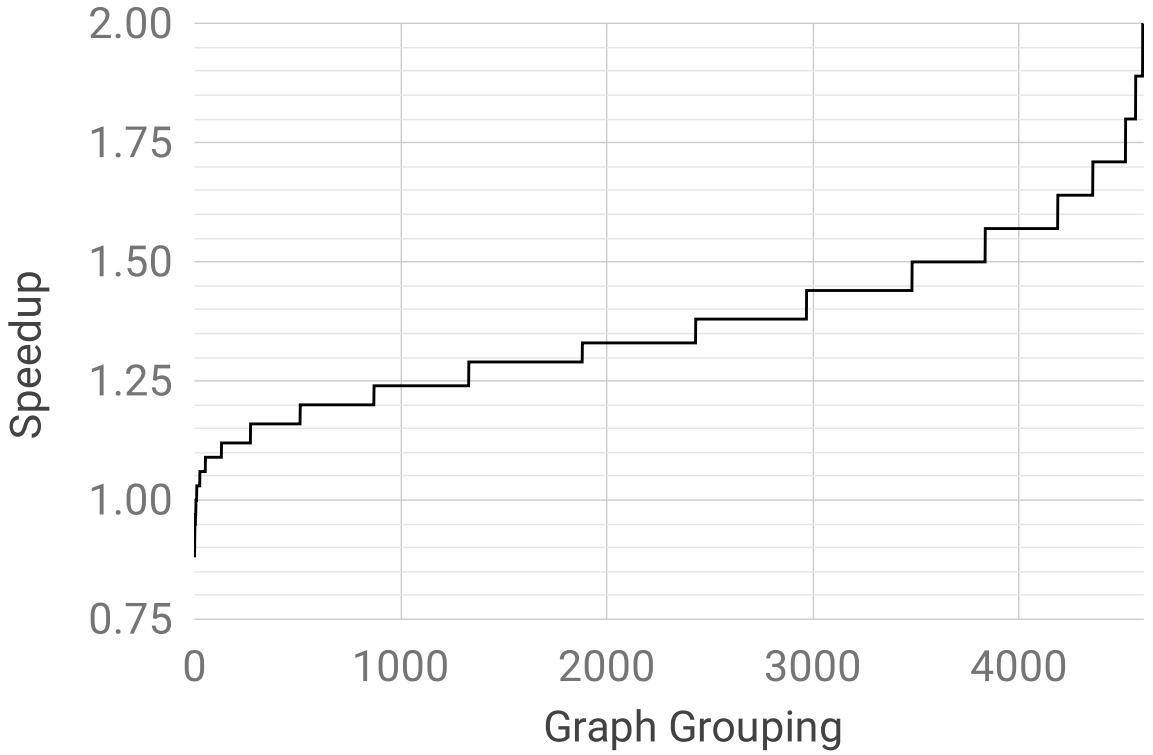

Figure 5.11 Distribution of speedups over sub-graph groupings of a randomly selected 10 node VXLIB graph. Speedup data was collected by running all possible groupings in the compiler for the selected graph.

Fig. 5.11 shows the distribution of speedups for all the 4,606 fusing configurations for a randomly-chosen 10-node VXLIB graph, computed from the output of the DSP compiler. The highest potential speedup achieved for this graph is 2.00, but only 4 out of 4,606 fusing configurations achieve a speedup of 2.00. This result illustrates the rarity of optimal graph fusing configurations.

Algorithm 5 Heuristic Scheduler

```
1: Input: inst, dag
2: Output: sched
3: function SCHED_INST(inst, dag)
4:     for each i ∈ [0, ii − 1], unit ∈ {L|S|D|M} do
5:         place inst into sched(i, unit)
6:         if avail_units > calc_units_needed(dag) then
7:             if is_reg_press_high(sched) then
8:                 exit recursion
9:             else
10:                SCHED_INST(inst_next, dag)
11:            end if
12:        else
13:            return NULL
14:        end if
15:    end for
16:    if are_reg_live_too_long(sched) then
17:        mv = insert_mv_instructions(dag)
18:        is_sucess = sched_mv_inst(mv, sched)
19:        if is_sucess then
20:            return sched
21:        else
22:            exit recursion
23:        end if
24:    end if
25:    return sched
26: end function
27: Input: dag
28: Output: sched
29: function BUILD_SCHED(dag)
30:    sort dag_inst per Sec. 5.2.3
31:    compute ii_min using ResMII
32:    compute ii_max as latency of single scheduled iteration
33:    for each ii from ii_min to ii_max do
34:        sched = SCHED_INST(inst_first, dag)
35:        if sched != NULL then
36:            return sched
37:        end if
38:    end for
39:    return NULL
40: end function
```

Chapter 6

Conclusion and Future Work

Image processing workloads that are represented by a graph structure inherently allow further optimization. This is especially true for a DSP-based architecture where the workload is statically scheduled. In a graph where the nodes are represented by image processing kernels, Kernel buffer fusing allows memory access optimizations. Grouping the most optimal kernels and selecting an optimal tile size for images allows for a reduction in DMA accesses and improved compute efficiency. This also reduces power consumption and achieves greater performances. Performance of a graph can be further increased if the kernels are fused together at the loop level, allowing for increased memory locality and functional unit utilization.

All the above mentioned optimizations are possible, only if the correct set of kernels are fused. In this dissertation, a model-based approach is presented to efficiently select the correct group of kernels to be fused through buffers or at loop level.

In the first part of this dissertation, machine learning techniques were applied to train the models using data collected from performance observed on the DSP hardware having various graph configurations. The models, when used to search for an OpenVX graph configuration that gives maximum performance, are able to select one that achieves within 2%, on average that of the best possible graph configuration. Our results show the potential to achieve 1.3 speedup, on average, for synthetic graphs of up to 10 nodes.

In the second part of this dissertation, a method for quickly estimating the performance given by fusing two or more image processing loops taken from a graph

composed of VXLIB image processing kernels (part of the Texas Instruments Vision SDK) was proposed. The performance model allows for the estimation of end-to-end performance of the graph when decomposed into multiple weakly-connected fused subgraphs. Using our code synthesis tool HeRCide, we can generate fused loops as C++ code for graph deployment.

Our performance model is based on modulo scheduling the instructions comprising the body of the fused loop. The scheduler is a heuristic that trades off less than 10% accuracy, notably by searching only a portion of the schedule space and skipping the register allocation phase, allowing for the rapid exploration of a large graph decomposition space. Our results indicate that the accuracy of the performance model is sufficient for selecting the optimal fusing configuration in 80% of graphs and selecting a configuration that achieves within 10% of the performance of the optimal configuration for 90% of graphs. This technique enabled us to achieve up to 1.9x speedup on average for sufficiently large synthetic graphs.

In a given graph of image processing kernels, performance improvements can be achieved from both kernel buffer fusing and kernel loop fusing. In our current work, these two approaches are done independently. As per future work, we can develop a unified model or set of models to apply both kernel buffer and loop fusing together to achieve the best performance possible for a given generic graph. This can be tested on the EVM board for at least a few graphs. Currently is it not possible to run this test for a large number of graphs because some part of the test has to be hard coded due to the static nature of the VXLIB kernel library in Texas Instruments Vision SDK (Software Development Kit).

Bibliography

[1] Madushan Abeysinghe, Jesse Villarreal, and Jason D. Bakos. "Optimizing OpenVX Graphs for Data Movement". In: (Feb. 2024).

[2] Madushan Abeysinghe, Jesse Villarreal, Todd T Hahn, and Jason D. Bakos. "Automated Loop Fusion for Image Processing". In: (Mar. 2024).

[3] Madushan Abeysinghe, Jesse Villarreal, Lucas Weaver, and Jason Bakos. "OpenVX Graph Optimization for Visual Processor Units". In: *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. Vol. 2160-052X. 2019, pp. 123–130.

[4] Lucas Alava Peña. "Machine Learning-Based Instruction Scheduling for a DSP Architecture Compiler : Instruction Scheduling using Deep Reinforcement Learning and Graph Convolutional Networks". MA thesis. KTH, School of Electrical Engineering and Computer Science (EECS), 2023, p. 76.

[5] AMD. *AMD OpenVX*. https://github.com/10imaging/openvx. 2019.

[6] Mehmet Ali Arslan and Krzysztof Kuchcinski. "Instruction Selection and Scheduling for DSP Kernels on Custom Architectures". In: *2013 Euromicro Conference on Digital System Design*. 2013, pp. 821–828.

[7] Mounir Bahtat, Said Belkouch, Phillipe Elleaume, and Phillipe Le Gall. "Fast enumeration-based modulo scheduling heuristic for VLIW architectures". In: *2014 26th International Conference on Microelectronics (ICM)*. 2014, pp. 116–119.

[8] F. Brill and E. Albuz. "NVIDIA VisionWorks Toolkit". In: Presented at the GPU Technology Conf, 2014.

[9] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. "Register allocation via coloring". In: *Computer Languages* 6.1 (1981), pp. 47–57. ISSN: 0096-0551.

[10] K. Chitnis, J. Villarreal, B. Jadav, M. Mody, L. Weaver, V. Cheng, K. Desappan, A. Jain, and P. Swami. "Novel OpenVX implementation for heteroge-

neous multi-core systems". In: *2017 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*. Oct. 2017, pp. 77–80.

[11] Kedar Chitnis, Jesse Villarreal, Lucas Weaver, Brijesh Jadav, Shyam Jaganathan, Mihir Mody, Taehun Kim, and Sujith Shivalingappa. "System Data Flow Pipelining for Embedded Heterogenous SoCs using OpenVX". In: *2020 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia)*. 2020, pp. 1–4.

[12] A. Cilardo, E. Fusella, L. Gallo, and A. Mazzeo. "Joint communication scheduling and interconnect synthesis for FPGA-based many-core systems". In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2014, pp. 1–4.

[13] L. Codrescu. "Architecture of the Hexagon™ 680 DSP for mobile imaging and computer vision". In: *2015 IEEE Hot Chips 27 Symposium (HCS)*. Aug. 2015, pp. 1–26.

[14] D. Dekkiche, B. Vincke, and A. Merigot. "Investigation and performance analysis of OpenVX optimizations on computer vision applications". In: *2016 14th International Conference on Control, Automation, Robotics and Vision (ICARCV)*. Nov. 2016, pp. 1–6.

[15] Can Deng, Zhaoyun Chen, Yang Shi, Yimin Ma, Mei Wen, and Lei Luo. "Optimizing VLIW Instruction Scheduling via a Two-Dimensional Constrained Dynamic Programming". In: *ACM Trans. Des. Autom. Electron. Syst.* (Jan. 2024). Just Accepted. ISSN: 1084-4309.

[16] Giuseppe S. Desoli. "Instruction Assignment for Clustered VLIW DSP Compilers: A New Approach". In: 1998.

[17] M. Ditty, A. Karandikar, and D. Reed. "Nvidia's Xavier SoC". In: Presented at Hot Chips: A Symposium on High Performance Chips, 2018.

[18] P. Faraboschi, J.A. Fisher, and C. Young. "Instruction scheduling for instruction level parallel processors". In: *Proceedings of the IEEE* 89.11 (2001), pp. 1638–1659.

[19] Jan Fousek, Jiři Filipovič, and Matuš Madzin. "Automatic Fusions of CUDA-GPU Kernels for Parallel Map". In: *SIGARCH Comput. Archit. News* 39.4 (Dec. 2011), pp. 98–99. ISSN: 0163-5964.

[20] A. Fraboulet, K. Kodary, and A. Mignotte. "Loop fusion for memory space optimization". In: *International Symposium on System Synthesis (IEEE Cat. No.01EX526)*. 2001, pp. 95–100.

[21] Philip B. Gibbons and Steven S. Muchnick. “Efficient instruction scheduling for a pipelined architecture”. In: *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*. SIGPLAN ’86. Palo Alto, California, USA: Association for Computing Machinery, 1986, pp. 11–16. ISBN: 0897911970.

[22] Kronos Group. *The OpenVX 1.1 Specification*. http://www.khronos.org/registry/OpenVX/specs/1.1/OpenVX_Specification_1_1.pdf. 2017.

[23] Texas Instruments Inc. *VXLIB User’s Manual (c66x)*. https://software-dl.ti.com/jacinto7/esd/processor-sdk-rtos-jacinto7/06_01_01_12/exports/docs/vxlib_c66x_1_1_4_0/docs/doxygen/html/index.html. 2019.

[24] Texas Instruments Incorporated. *TMS320C66x DSP CPU and Instruction Set Reference Guide*. https://www.ti.com/lit/ug/sprugh7/sprugh7.pdf. 2010.

[25] Intel. *Intel FPGA SDK for OpenCL*. https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html. 2019.

[26] Intel. *OpenVINO Toolkit*. https://software.intel.com/en-us/openvino-toolkit. 2019.

[27] Shyam Jagannathan, Vijay Pothukuchi, Jesse Villarreal, Kumar Desappan, Manu Mathew, Rahul Ravikumar, Aniket Limaye, Mihir Mody, Pramod Swami, Piyali Goswami, Carlos Rodriguez, Emmanuel Madrigal, and Marco Herrera. “OpTIFlow - An optimized end-to-end dataflow for accelerating deep learning workloads on heterogeneous SoCs”. In: *Electronic Imaging* 35.16 (2023), pp. 113–1–113–1.

[28] R. Staszewski K. Chitnis and G. Agarwal. “TI Vision SDK, Optimized Vision Libraries for ADAS Systems”. In: *White Paper: SPRY260, Texas Instrument Inc.* (Apr. 2014).

[29] Ken Kennedy and Kathryn S. McKinley. “Maximizing loop parallelism and improving data locality via loop fusion and distribution”. In: *Languages and Compilers for Parallel Computing*. Ed. by Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 301–320. ISBN: 978-3-540-48308-3.

[30] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. “Modeling GPU-CPU Workloads and Systems”. In: *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. GPGPU-3. Pittsburgh, Pennsylvania, USA: ACM, 2010, pp. 31–42. ISBN: 978-1-60558-935-0.

[31] E.A. Lee and S. Ha. “Scheduling strategies for multiprocessor real-time DSP”. In: *1989 IEEE Global Telecommunications Conference and Exhibition 'Communications Technology for the 1990s and Beyond'*. 1989, 1279–1283 vol.2.

[32] R. Leupers. “Instruction scheduling for clustered VLIW DSPs”. In: *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00622)*. 2000, pp. 291–300.

[33] U. Lopez-Novoa, A. Mendiburu, and J. Miguel-Alonso. “A Survey of Performance Modeling and Simulation Techniques for Accelerator-Based Computing”. In: *IEEE Transactions on Parallel and Distributed Systems* 26.1 (Jan. 2015), pp. 272–281. ISSN: 1045-9219.

[34] Roberto Castañeda Lozano and Christian Schulte. “Survey on Combinatorial Register Allocation and Instruction Scheduling”. In: *ACM Comput. Surv.* 52.3 (June 2019). ISSN: 0360-0300.

[35] Sanyam Mehta, Pei-Hung Lin, and Pen-Chung Yew. “Revisiting Loop Fusion in the Polyhedral Framework”. In: *SIGPLAN Not.* 49.8 (Feb. 2014), pp. 233–246. ISSN: 0362-1340.

[36] Jiayuan Meng, Vitali A. Morozov, Venkatram Vishwanath, and Kalyan Kumaran. “Dataflow-driven GPU performance projection for multi-kernel transformations”. In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–11.

[37] Mihir Mody, Kedar Chitnis, Hemant Hariyani, Shyam Jagannathan, Jason Jones, Gregory Shurtz, Abhishek Shankar, Ankur, Mayank Mangla, Sriramakrishnan Govindarajan, Aish Dubey, and Kai Chirca. “Single Chip Auto-Valet Parking System with TDA4VMID SoC”. In: *Electronic Imaging* 33.17 (2021), pp. 113-1–113-1.

[38] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. “PolyMage: Automatic Optimization for Image Processing Pipelines”. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. Istanbul, Turkey: ACM, 2015, pp. 429–443. ISBN: 978-1-4503-2835-7.

[39] H. Omidian and G. G. F. Lemieux. “Exploring automated space/time tradeoffs for OpenVX compute graphs”. In: *2017 International Conference on Field Programmable Technology (ICFPT)*. Dec. 2017, pp. 152–159.

[40] Hossein Omidian and Guy G. F. Lemieux. “JANUS: A Compilation System for Balancing Parallelism and Performance in OpenVX”. In: *Journal of Physics: Conference Series* 1004 (Apr. 2018), p. 012011.

[41] Krishna Palem and Barbara Simons. “Scheduling time-critical instructions on RISC machines”. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’90. San Francisco, California, USA: Association for Computing Machinery, 1989, pp. 270–280. ISBN: 0897913434.

[42] Bo Qiao, Oliver Reiche, Frank Hannig, and Jïrgen Teich. “From Loop Fusion to Kernel Fusion: A Domain-Specific Approach to Locality Optimization”. In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2019, pp. 242–253.

[43] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. “Automatic Kernel Fusion for Image Processing DSLs”. In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*. SCOPES ’18. Sankt Goar, Germany: ACM, 2018, pp. 76–85. ISBN: 978-1-4503-5780-7.

[44] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. “Automatic Kernel Fusion for Image Processing DSLs”. In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*. SCOPES ’18. Sankt Goar, Germany: Association for Computing Machinery, 2018, pp. 76–85. ISBN: 9781450357807.

[45] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. “Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’13. Seattle, Washington, USA: ACM, 2013, pp. 519–530. ISBN: 978-1-4503-2014-6.

[46] Erik Rainey, Jesse Villarreal, Goksel Dedeoglu, Kari Pulli, Thierry Lepley, and Frank Brill. “Addressing System-Level Optimization with OpenVX Graphs”. In: *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*. CVPRW ’14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 658–663. ISBN: 978-1-4799-4308-1.

[47] B. Ramakrishna Rau. “Iterative modulo scheduling: an algorithm for software pipelining loops”. In: *Proceedings of the 27th Annual International Symposium on Microarchitecture*. MICRO 27. San Jose, California, USA: Association for Computing Machinery, 1994, pp. 63–74. ISBN: 0897917073.

[48] J. Redgrave, A. Meixner, N. Goulding-Hotta, A. Vasilyev, and O. Shacham. “Pixel Visual Core:Google’s Fully ProgrammableImage, Vision, and AI Processor For Mobile Devices”. In: Presented at Hot Chips: A Symposium on High Performance Chips, 2018.

[49] J. Sankaran and N. Zoran. “TDA2X, a SoC optimized for advanced driver assistance systems”. In: *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. May 2014, pp. 2204–2208.

[50] R. Saussard, B. Bouzid, M. Vasiliu, and R. Reynaud. “Optimal Performance Prediction of ADAS Algorithms on Embedded Parallel Architectures”. In: *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. Aug. 2015, pp. 213–218.

[51] Jason Sewall and Simon J. Pennycook. *High-Performance Code Generation though Fusion and Vectorization*. 2017. arXiv: 1710.08774 [cs.PF].

[52] Eric J. Stotzer. “Complementary compiler and architecture features for embedded vliw processors”. AAI3470430. PhD thesis. USA, 2010. ISBN: 9781124127828.

[53] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini. “ADRENALINE: An OpenVX Environment to Optimize Embedded Vision Applications on Many-core Accelerators”. In: *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. Sept. 2015, pp. 289–296.

[54] Giuseppe Tagliavini, Germain Haugou, Andrea Marongiu, and Luca Benini. “Optimizing Memory Bandwidth Exploitation for OpenVX Applications on Embedded Many-core Accelerators”. In: *J. Real-Time Image Process.* 15.1 (June 2018), pp. 73–92. ISSN: 1861-8200.

[55] Sajjad Taheri, Payman Behnam, Eli Bozorgzadeh, Alexander Veidenbaum, and Alexandru Nicolau. “AFFIX: Automatic Acceleration Framework for FPGA Implementation of OpenVX Vision Algorithms”. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’19. Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 252–261. ISBN: 9781450361378.

[56] S. Tavarageri, L. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. “Dynamic selection of tile sizes”. In: *2011 18th International Conference on High Performance Computing*. Dec. 2011, pp. 1–10.

[57] Adwin H. Timmer. “Conflict Modelling and Instruction Scheduling in Code Generation for In-House DSP Cores”. In: *32nd Design Automation Conference*. 1995, pp. 593–598.

[58] Kent Wilken, Jack Liu, and Mark Heffernan. “Optimal instruction scheduling using integer programming”. In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. PLDI ’00.

Vancouver, British Columbia, Canada: Association for Computing Machinery, 2000, pp. 121–133. ISBN: 1581131992.

[59] Haicheng Wu, Gregory Frederick Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili, and Srimat T. Chakradhar. "Optimizing Data Warehousing Applications for GPUs Using Kernel Fusion/Fission". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum* (2012), pp. 2433–2442.

[60] Song-Lin Wu, Xiang-Yu Wang, Ming-Yi Peng, and Shih-Wei Liao. "Accelerating OpenVX through Halide and MLIR". In: *J. Signal Process. Syst.* 95.5 (Feb. 2023), pp. 571–584. ISSN: 1939-8018.

[61] Xilinx. *Xilinx SDAcel*. https://www.xilinx.com/products/design-tools/legacy-tools/sdaccel.html. 2019.

[62] J. Xue, Q. Huang, and M. Guo. "Enabling loop fusion and tiling for cache performance by fixing fusion-preventing data dependences". In: *2005 International Conference on Parallel Processing (ICPP'05)*. 2005, pp. 107–115.

[63] Kecheng Yang, Glenn A. Elliott, and James H. Anderson. "Analysis for Supporting Real-time Computer Vision Workloads Using OpenVX on Multicore+GPU Platforms". In: *Proceedings of the 23rd International Conference on Real Time and Networks Systems*. RTNS '15. Lille, France: ACM, 2015, pp. 77–86. ISBN: 978-1-4503-3591-1.

[64] Xiaolei Zhao, Zhaoyun Chen, Yang Shi, Mei Wen, and Chunyun Zhang. "Automatic End-to-End Joint Optimization for Kernel Compilation on DSPs". In: *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 2023, pp. 1–6.