12-15-2014

Automatic Loop Tuning and Memory Management for Stencil Computations

Fan Zhang
University of South Carolina - Columbia

Automatic Loop Tuning and Memory Management For Stencil Computations

by

Fan Zhang

Bachelor of Science
Northeastern University 2002
Master of Science
Northeastern University 2008

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

2014

Accepted by:

Jason Bakos, Major Professor

Manton Matthews, Committee Member

Jijun Tang, Committee Member

Song Wang, Committee Member

Enrico Santi, Committee Member

Lacy Ford, Vice Provost and Dean of Graduate Studies

Acknowledgments

First of all, I would like to express my deep gratitude to my advisor Dr. Jason Bakos, for his valuable suggestions and encouragement during my research. Without his guidance and support, I could not complete my dissertation.

I would also like to thank my labmates in Heterogeneous and Reconfigurable Computing Labs. I am particularly grateful for the assistance provided by Yang Gao in the TI C66x DSP platform building and programming. I would like to offer my special thanks to Yan Zhang for his help in providing background knowledges. My grateful thanks are also extended to Mr.Zheming Jin, who helped me during design troubleshooting.

Finally, I wish to thank my family, for their understanding and unwavering support throughout my study.

Abstract

The Texas Instruments C66x Digital Signal Processor (DSP) is an embedded processor technology that is targeted at real time signal processing. It is also developed with a high potential to become the new generation of coprocessor technology for high performance embedded computing. Of particular interest is its performance for stencil computations, such as those found in signal processing and computer vision tasks. A stencil is a loop in which the output value is updated at each position of an array by taking a weighted function of its neighbors. Efficiently mapping stencil-based kernels to the C66x device presents two challenges. The first one is how to efficiently optimize loops in order to facilitate the usage of Single Instruction Multiple Data (SIMD) instructions. On this architecture, like most others, SIMD instructions are not directly generated by the compiler. The second problem is how to manage on-chip memory in a way that minimizes off-chip memory access. Although this could theoretically be achieved by using a highly associative cache, the high rate of data reuse in stencil loops causes a high conflict miss rate. One way to solve this problem is to configure the on-chip memory as a program controlled scratchpad. It allows user to buffer a 2D block of data and minimizes the off-chip data access.

For this dissertation, we have accomplished two goals: (1) Develop a methodology for optimization of arbitrary 2D stencils that fully utilize SIMD instructions through microachitecture-aware loop unrolling. (2) Deliver an easy-to-use scratchpad buffer management system and use it to improve the memory efficiency for 2D stencils. We show in the results and analysis section that our stencil compiler is able to achieve up to 2x speed up compared with the code generated by the industrial standard compiler developed by Texas Instruments, and our memory management system is able to achieve up to 10x speed up compared with cache.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Stencil loops are implemented using iterative finite-difference techniques that sweep over a spatial grid and perform nearest neighbor computations. In a stencil loop, each point in a regular grid is updated with weighted contributions from a subset of its spatial neighbors. Stencil loops are widely used in computer vision routines such as spatial filters, feature extraction, and movement detection. They are useful not only in computer vision but also in other numerical methods such as partial derivative equation solvers and spatial data clustering.

Optimizing stencil loops is a time-consuming task and requires a large amount of engineering effort. For example, a naively implemented Gaussian filter contains less than 10 lines of C code. However its performance could be 3-5x slower than an optimized implementation, which is over 100 lines of C code and partially implemented by assembly intrinsic. Difficulties of implementing high performance stencil loops come from the dense floating point arithmetic and data reuse. In this dissertation, we study the techniques that optimize the memory and computation efficiency for stencil loops and develop methodologies for automatic code optimization. In addition, we select the TI TMS320C6678 embedded DSP as our design platform and targeted at accelerating a set of widely used stencil kernels in computer vision and signal processing.

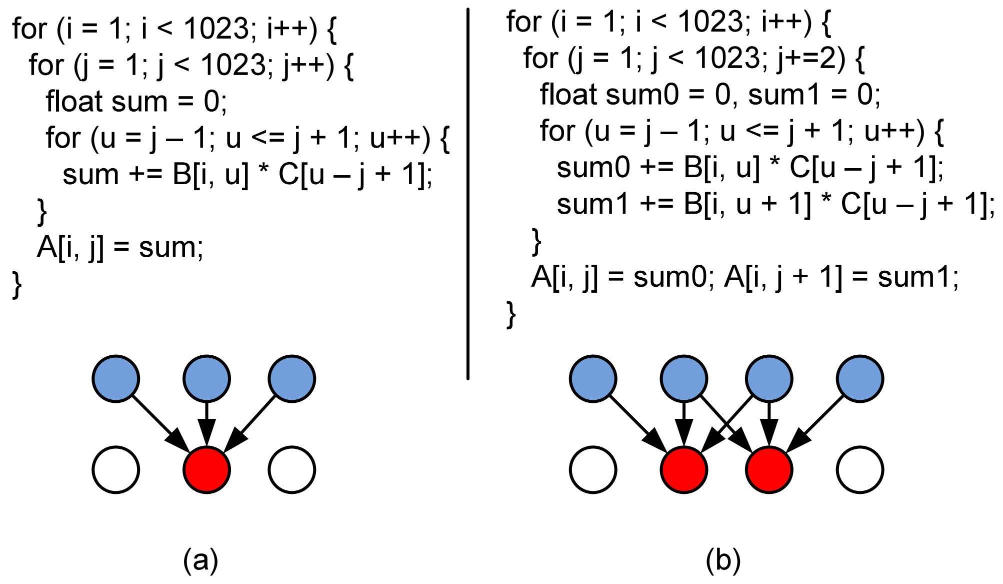
1.1 Optimization of Stencils by Loop Unrolling and SIMD Instructions

Two of the most powerful techniques for stencil optimization are SIMD instruction and loop unrolling. However, the compiler do not have the ability to perform these optimization mechanisms unless the code is written in a specific way. This is because that the compiler assumes a loop is general structuralized. This limits the ability of the general-purpose language compiler such as C and C++ to perform these code optimizations on specialized loops.

Unrolling loops allows the compiler to pack up statically scheduled instructions into multiple registers and functional units and to achieve higher instruction throughput from parallel execution. However, optimizing unroll parameter requires a comprehensive analysis on both the arithmetic structure of the loop and the architecture of the microprocessor. A small unrolling factor does not fully utilize the hardware parallelism potential while large ones may exhaust register resources and create overhead from data traffic. One of the goals of this dissertation is to develop a compiler technology that is able to select the best unrolling strategy.

The trade off of loop unrolling is given in Figure 1.1. This figure shows a horizontal 1x3 filter (a) and a 2x manually unrolled version (b). We use the same strategy to manually generate the code with unrolling up to 20x and test the performance of these unrolled versions in Figure 1.2. It shows how loop unrolling affects the performance as well as the other factors that are related to the code performance, such as the functional unit utilization, register usage and instruction parallelism. Although the search space of loop unrolling is small, consider that it could be combined with the other optimization techniques, It could become a hard problem to solve.

SIMD instruction is an important aspect of modern CPU technology that performs instruction-level parallelization. It is performed during the basic block optimization



```
for (i = 1; i < 1023; i++) {
  for (j = 1; j < 1023; j++) {
    float sum = 0;
    for (u = j – 1; u <= j + 1; u++) {
      sum += B[i, u] * C[u – j + 1];
    }
   A[i, j] = sum;
}
```

```
for (i = 1; i < 1023; i++) {
  for (j = 1; j < 1023; j+=2) {
    float sum0 = 0, sum1 = 0;
    for (u = j – 1; u <= j + 1; u++) {
      sum0 += B[i, u] * C[u – j + 1];
      sum1 += B[i, u + 1] * C[u – j + 1];
   }
   A[i, j] = sum0; A[i, j + 1] = sum1;
}
```

Figure 1.1: A Typical Stencil Loop and Unrolling Manually by 2x

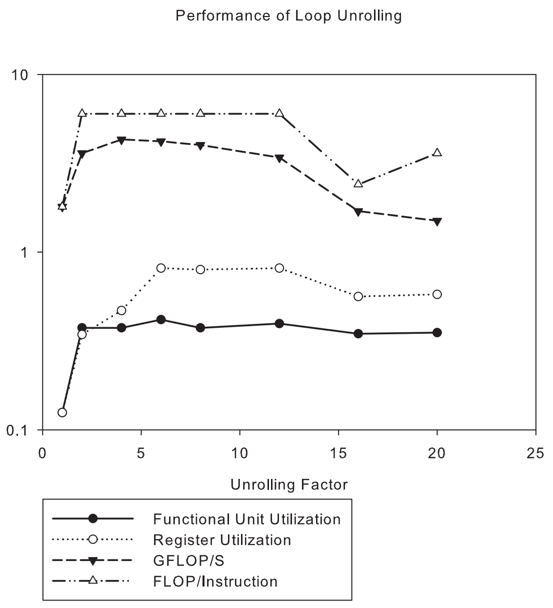
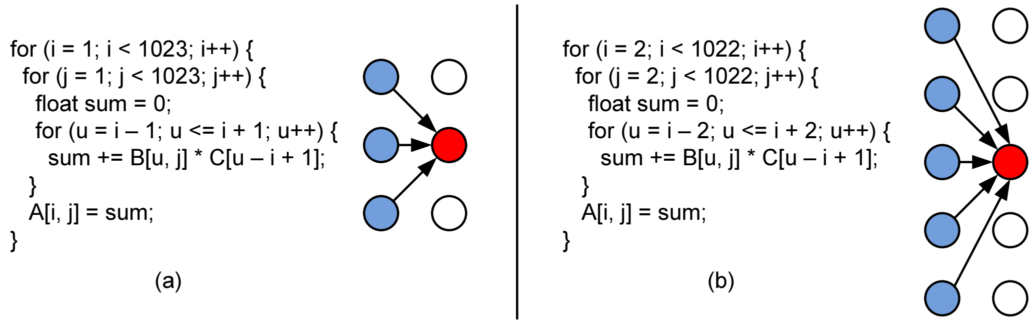


Figure 1.2: Performance Comparison of Loop Unrolling



```
for (i = 1; i < 1023; i++) {
  for (j = 1; j < 1023; j++) {
    float sum = 0;
    for (u = i – 1; u <= i + 1; u++) {
      sum += B[u, j] * C[u – i + 1];
    }
   A[i, j] = sum;
}
```

```
for (i = 2; i < 1022; i++) {
  for (j = 2; j < 1022; j++) {
    float sum = 0;
    for (u = i – 2; u <= i + 2; u++) {
      sum += B[u, j] * C[u – i + 1];
    }
   A[i, j] = sum;
}
```

Figure 1.3: 3 Points and 5 Points Vertical Stencil Loop Example

phase of compilation in which the data-independent scalar operations can be grouped up into vectorized instructions. To automatically generate SIMD optimization for stencil loops, the compiler need to make two decisions, how to assign the operands of the stencil into vectorized registers so that the number of inter-register transfer is minimized and how to assign functional units to the computation pipeline in order to minimize CPU stall rate. In this dissertation, we develop a technique to perform SIMD instruction binding based on compiler-level code analysis, and use it together with loop unrolling to explore the best strategy for stencil loop optimization.

1.2 Improving the Memory Performance by Loop Tiling and Buffering

Memory efficiency influences the performance of stencil loop. In modern processor design, data access patterns that exhibit locality benefit from cache. However, TI C66x DSP is designed with a limited associative cache and it does not perform efficiently on multi-dimensional data. Reading data using a stride pattern such as accessing row major data in column order potentially causes cache conflict misses.

One of the example code that cause cache misses is vertical stencils. Figure 1.3 shows two examples of 3 points and 5 points vertical stencil.

One way to improve the memory performance of stencil loops is to tile the loop in blocks and use the high speed on-chip memory as program controlled scratchpad. By

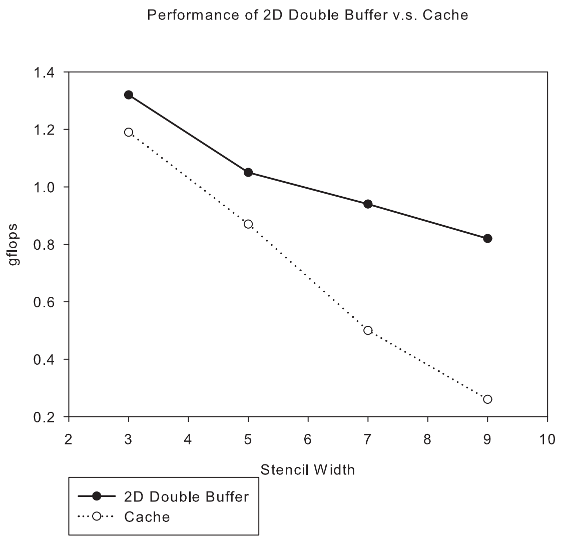


Figure 1.4: Scratchpad Buffer v.s. Cache

applying this method, the blocks of the source matrices are copied into the scratchpad memory before they are processed by the computation kernel. However, customizing buffer for a particular stencil loop requires the programmer to manage on-chip memory explicitly and adds considerable work load into the code design phase. In order to provide a generic and easy-to-use memory management system, we develop an automatic data transfer system on TI C66x DSP. It is able to parallelize data transfer and computation by using the hardware memory transfer DMA so that the memory performance could be maximized.

Figure 1.4 shows the performance of using 2D buffer v.s cache on vertical stencil of different width. We can observe from the figure that as the stencil width increases, cache performance drops fast while the performance of using strachpad memory stays constant.

1.3 Specific Aims

Figure 1.5 shows an overview of our stencil loop optimization system. User composed stencil code will be parsed into LLVM IR and processed by our optimization tools

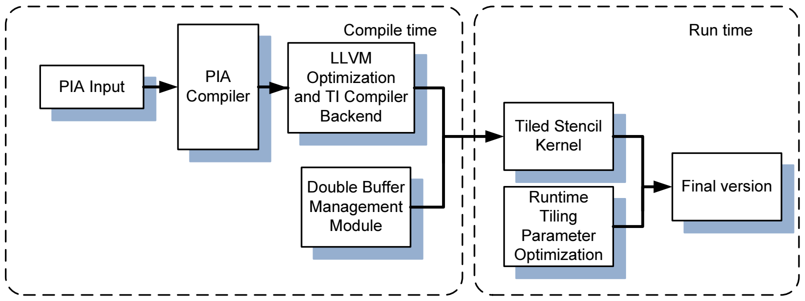

Compile time
Run time
PIA Input
PIA Compiler
LLVM Optimization and TI Compiler Backend
Double Buffer Management Module
Tiled Stencil Kernel
Runtime Tiling Parameter Optimization
Final version

Figure 1.5: A System Overview

and linked with the buffer management module. A cost evaluation model will be performed to optimize the tiling parameters and converge the system to the best performance.

Our contributions can be summerized as follows:

- Research on optimization methodologies that target at microarchitecture-aware stencil loop optimization.
- Develop an automatic double buffer scratchpad that improves the memory performance on stencil loop tiling.

The structure of this dissertation is organized as follows, Chaper 1 gives the introduction, Chaper 2 introduces the background of Stencil Loops and the DSP's micro-architecture, Chapter 3 lists and compares the related work. Chapter 4 provides the methodology and implementation. Chapter 5 gives the experimental results and analysis. Chapter 6 gives the conclusion.

Chapter 2

Background

In this chapter we give the background knowledge of stencil loops, introduction to the typical stencil loops that plays an important role in signal processing and computer vision, and an overview of the TI C66x embedded processor architecture and code optimization.

2.1 1D and 2D Stencil Loops

In a stencil operation, each point in a regular grid is updated with weighted contributions from a subset of its neighbors in both time and space called coefficients. These coefficients may be the same at every grid point (a constant coefficient stencil) or not (a variable coefficient stencil). Figure 2.1 shows a typical 3 points 1D stencil and a 5 points 2D stencil.

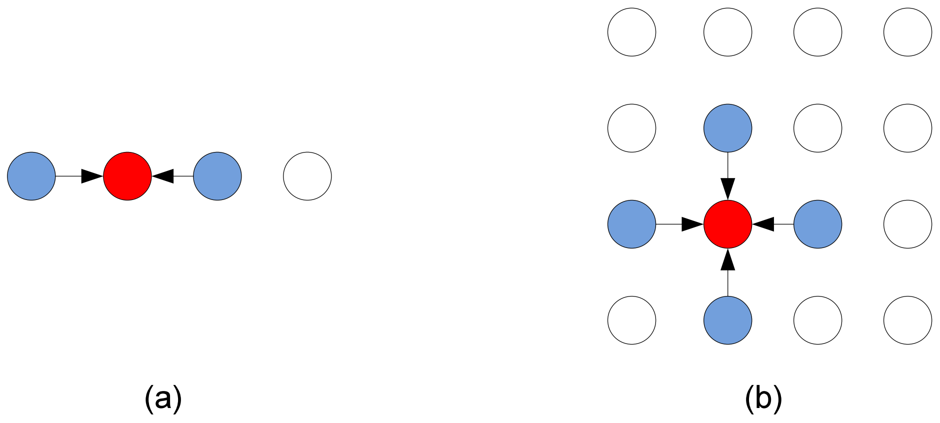


Figure 2.1: Example of 1D and 2D Stencil Loops

Stencil performs sweeps through data structures that are usually much larger than the capacity of the available data caches. However, it contains exploitable temporal locality that allows for data reuse. Normally, a 1D stencil can reuse all but one point per iteration. The efficiency of utilizing stencil locality is limited by two factors: cache size and cache conflict rates. To avoid performance loss from cache conflict miss, optimization technique such as tiling are widely used in such cases that strided data access is often, but it requires user to explicitly manage the data transfer between off-chip and on-chip memory [44].

The stencil loop can be single input, single output or multiple inputs, multiple outputs. Single input and single output stencil loops read value from a source matrix and output the results to a destination matrix. It is commonly used for spatial filters (Gaussian, Sobel, Laplacian). Multiple input and multiple output stencils read and write values from multiple matrices (for example, matrix addition) and has more complicated data locality pattern. It is used widely as feature extraction and motion detection (Derivatives, Haar Feature, DOG, HOG, Lucas-Kanade method).

2.2 Stencil Loops in Signal Processing and Computer Vision

1D and 2D Mean Filter

Mean filtering is a simple, intuitive and easy to implement method of smoothing signals and images, i.e. reducing the amount of intensity variation between one pixel and the next. In computer vision, it is commonly used for noise removal.

The idea of mean filtering is simply to replace each pixel value in an image with the average value of its neighbors, including itself. This has the effect of eliminating pixel values which are unrepresentative of their surroundings. Mean filtering is a typical example of a convolutional filter. It is based around a kernel, and represents the shape and size of the neighborhood to be sampled when calculating the mean.

Often a $N \times N$ square kernel is used.

Gaussian Filter

The Gaussian filter is a 1D or 2D convolutional operator that is used to blur images or remove noise from signals. In this sense it is similar to the mean filter, but it uses a different kernel that represents the shape of a Gaussian hump. Computationally, Gaussian filter is a weighted mean filter. It requires more arithmetic operations to compute each output value. Gaussian filter could be be generalized to high dimensional space, however, 1D and 2D Gaussian are used more frequently than higher dimensional Gaussians.

Equation 2.1 shows an 1D 7 points Gaussian filter. Equation 2.2 shows a 2D 5×5 Gaussian filter.

$$Gaussian1D = \begin{pmatrix} 0.006 & 0.061 & 0.242 & 0.383 & 0.242 & 0.061 & 0.006 \end{pmatrix} \tag{2.1}$$

$$Gaussian2D = \frac{1}{273} \begin{pmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{pmatrix} \tag{2.2}$$

Jacobi Stencil

The Jacobi stencil is a type of stencil loops that are used in numerical solver of partial differential equations. To be more specific, the Jacobi stencil is the iterative format for finding the solutions to the class of boundary value problems of the form $\Delta_u = 0$. In two dimensional space, Jacobi stencil can be formulated as Equation 2.3.

$$O_{i,j} = 0.25 \times (I_{i-1,j} + I_{i+1,j} + I_{i,j-1} + I_{i,j+1}) \tag{2.3}$$

Sobel Filter

The Sobel filter uses two 3x3 kernel matrices which are convolved with the original image to calculate approximations of the derivatives of the pixel intensive in horizontal and vertical direction. If we define A as the source image, and G_x and G_y are two images which at each point contain the horizontal and vertical derivative approximations. The horizontal and vertical derivatives of the image can be calculated by $G_x * A$ and $G_y * A$, in which $*$ is the convolution operator.

$$G_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \tag{2.4}$$

$$G_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \tag{2.5}$$

Harris Corner Method

The Harris Corner Detector is used widely in image processing for discovering the feature points, which are marked as corners. The basic idea of Harris Corner Detector is to find the shift score of a given patch from all directions. This method is introduced by Harris and Stephen in 1988.

The Harris Corner score is computed within a sliding window. Let the sliding window be W, the scoring function involves firstly computing the horizontal and vertical partial derivatives of the image I_x and I_y, and then computing the three summation values ΣI_x^2, ΣI_y^2 and $\Sigma I_x I_y$ over the sliding window. The corner score can be computed by $\Sigma I_x^2 + \Sigma I_y^2 - \alpha \Sigma I_x I_y$.

Equation 2.6 and 2.7 shows the computation performed at each pixel of Harris Corner Method.

$$A = \sum_w \begin{pmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{pmatrix} \tag{2.6}$$

$$S = det(A) - trace^2(A) \tag{2.7}$$

Lucas Kanade Method

Lucas Kanade method is a typical computer vision algorithm that can be organized as a series of 2D stencil loops. It is widely used for computing the pixel displacement between two consecutive frames in a video stream. Lucas Kanade method assumes the pixel movement can be approximated by its first order partial derivatives. Assuming that an image sequence is a three dimensional function $f(x, y, t)$ in which the value of the function is the pixel intensity. Lucas-Kanade method evaluates optical flow by solving the Equation 2.8.

$$f(x, y, t) = f(x + \Delta x, y + \Delta y, t + \Delta t) \tag{2.8}$$

The right side of the equation can be approximated by its first-order Tylor expansion. (Equation 2.9)

$$f(x, y, t) = f(x, y, t) + \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y + \frac{\partial f}{\partial t}\Delta t \tag{2.9}$$

And it provides Equation 2.10

$$\frac{\partial f}{\partial x}V_x + \frac{\partial f}{\partial y}V_y = -\frac{\partial f}{\partial t} \tag{2.10}$$

There are multiple ways to evaluate partial derivatives from an image and most of them are based on 2D stencil computation (convolution). The simplest way to compute the partial derivative is to use convolution matrix $kv_x = [-0.5, 0, 0.5]$ and $kv_y = [-0.5, 0, 0.5]^T$. Other kernels that are performed with local smoothing such as Prewitt and Sobel Filter are introduced by Scharr [43].

However, Equation 2.10 still has two unknown variables that can cannot be solved if no additional condition is provided. This is known as the aperture problem. Lucas Kanade method assumes that the optical flow is spatially perservative. The pixels close to each other should have identical velocity. Thus the velocity value can be solved by performing least square method on a neighbor window. Assume the neighbor window around the current point includes n piexels $q_1, q_2, ...q_n$.

$$\begin{gathered}
\frac{\partial f}{\partial x}(q_1)V_x + \frac{\partial f}{\partial y}(q_1)V_y = -\frac{\partial f}{\partial t}(q_1) \\
\frac{\partial f}{\partial x}(q_2)V_x + \frac{\partial f}{\partial y}(q_2)V_y = -\frac{\partial f}{\partial t}(q_2) \\
... \\
\frac{\partial f}{\partial x}(q_n)V_x + \frac{\partial f}{\partial y}(q_n)V_y = -\frac{\partial f}{\partial t}(q_n)
\end{gathered} \tag{2.11}$$

Because the number of equations is larger than the number of the unknowns, Equation 2.11 need to be solved by least square method. Let

$$A = \begin{bmatrix} f_x(q_1) & f_y(q_1) \\ f_x(q_2) & f_y(q_2) \\ ... & \\ f_x(q_n) & f_y(q_n) \end{bmatrix} \tag{2.12}$$

$$v = \begin{pmatrix} V_x \\ V_y \end{pmatrix} \tag{2.13}$$

$$b = \begin{pmatrix} -f_t(q_1) \\ -f_t(q_2) \\ ... \\ -f_t(q_n) \end{pmatrix} \tag{2.14}$$

The velocity values V_x and V_y of each pixel can be computed by Equation 2.15.

$$v = (A^T A)^{-1} A^T b \tag{2.15}$$

Algorithm 1 shows the pseudo code of the least square method.

Algorithm 1 Compute Flow Field by Least Square Method

```
Input: Gaussian blurred images $im1, im2$ and derivative matrices $v_x, v_y, v_t$, neighbor window size $l$
Output: Optical flow $f_x, f_y$
for $x = 0 \rightarrow m - 1 - l$ do
    for $y = 0 \rightarrow n - 1 - l$ do
        $a_{11} = 0, a_{12} = 0, a_{22} = 0, ab_1 = 0, ab_2 = 0$
        for $u = x \rightarrow x + l$ do
            for $v = y \rightarrow y + l$ do
                $a_{11} = a_{11} + v_x^2(u, v)$
                $a_{12} = a_{12} + v_x(u, v)v_y(u, v)$
                $a_{22} = a_{22} + v_y^2(u, v)$
                $ab_1 = ab_1 + v_x(u, v)v_t(u, v)$
                $ab_2 = ab_2 + v_x(u, v)v_t(u, v)$
            end for
        end for
        $det_a = a_{11}a_{22} - a_{12}^2$
        $ia_{11} = a_{22}/det_a$
        $ia_{12} = -a_{12}/det_a$
        $ia_{22} = a_{11}/det_a$
        $f_x(x + l/2, y + l/2) = ia_{11}ab_1 + ia_{12}ab_2$
        $f_y(x + l/2, y + l/2) = ia_{12}ab_1 + ia_{22}ab_2$
    end for
end for
```

Lucas Kanade method is a typical algorithm in computer vision that can be mapped to a series of stencil loops. It is a good baseline method for testing our proposed stencil loop code generator. In order to make the implemented optical flow system more practical, we follow the state-of-the-art research and build up the system from the Lucas Kanade method with a number of algorithmic improvements.

2.3 The Benchmark Stencil Loop Kernels

From the introduced stencil loops, we select a set of benchmarks for the evaluation of our stencil code generation and optimization tool. In details, the C code of the benchmark stencil loops are listed in the Appendix A of the dissertation.

Our benchmark kernels include:

- Matrix add: Compute the element wise summation of two matrices.
- 1x3 Mean filter: Compute and store the average value of horizontal consecutive three elements.
- 3x3 Mean filter: Compute and store the average value of a 3 by 3 grid.
- 4 point Jacobi stencil: An instance of 2D Jacobi stencil family.
- 7 point Gaussian filter: Horizontal Gaussian filter.
- Sobel filter: Compute the image gradient on x and y direction.
- Harris Corner detector: The corner scoring kernel of Harris method.
- Lucas Kanade method: The least square method kernel of Lucas Kanade optical flow estimation.

The characteristic of the selected kernel benchmarks are listed in the Table 2.1. In the table, we provide the number of load/store/floating point addition and multiplication in the innermost loop of the stencil. We also list the memory to compute ratio (The number of memory instructions divided the number of arithmetic instructions), the bottleneck of the computation, and the theoretical maximum performance measured by Gflops (Giga Floating Point Operation per Second).

Table 2.1: Characteristic of the Selected Stencil Kernel Benchmark

Kernel	Load	Store	Mul	Add	Mem	Arith	M/C	Bound	Gflops
Matrix Add	2.0	1.0	0.0	1.0	3.0	1.0	3.0	Mem	1.3
1x3 mean	3.0	1.0	1.0	2.0	4.0	3.0	1.3	Mem	1.0
3x3 mean	9.0	1.0	1.0	8.0	10.0	9.0	1.1	Mem	1.0
Jaccobi	4.0	1.0	1.0	3.0	5.0	4.0	1.2	Mem	1.0
Gaussian	7.0	1.0	7.0	6.0	8.0	13.0	0.6	Mem	2.0
Sobel	8.0	2.0	2.0	6.0	10.0	8.0	1.3	Mem	1.5
Harris	18.0	2.0	27.0	27.0	19.0	54.0	0.35	Comp.	3.0
LK	27.0	2.0	45.0	45.0	29.0	90.0	0.32	Comp.	3.3

In the benchmarks, the first 6 stencils are memory bound. The last 2 stencils, Harris Corner and Lucas Kanade, are compute bound and have a more complex computation characteristic.

2.4 TI Keystone High-Performance DSP Architecture

Nowadays the term High Performance Computation is more and more moves off from traditional PC and computer servers, but into the new generation of devices that is able to perform computation faster or with higher degree of parallelism. For example, general purpose GPUs and DSPs are widely applied into the domain of scientific computation [3] [7] [8] [13] [14], simulation [4], [30] [23] data mining and machine learning [51] [12].

The Texas Instruments Keystone architecture have the potential to achieve high power efficiency for scientific computation. They are designed as VLIW microprocessors that allow wide instruction-level parallelism and multicore support. However, unlike GPUs, these DSPs have integrated network interfaces and are capable of running an operating system, thus in theory they could participate in a distributed processing system with little or no involvement from CPU-based hosts. Much like how the GPU's role as a coprocessor was an outgrowth of its initial market in the 3D gaming industry, the TI DSP that originally designed for data processing in cel-

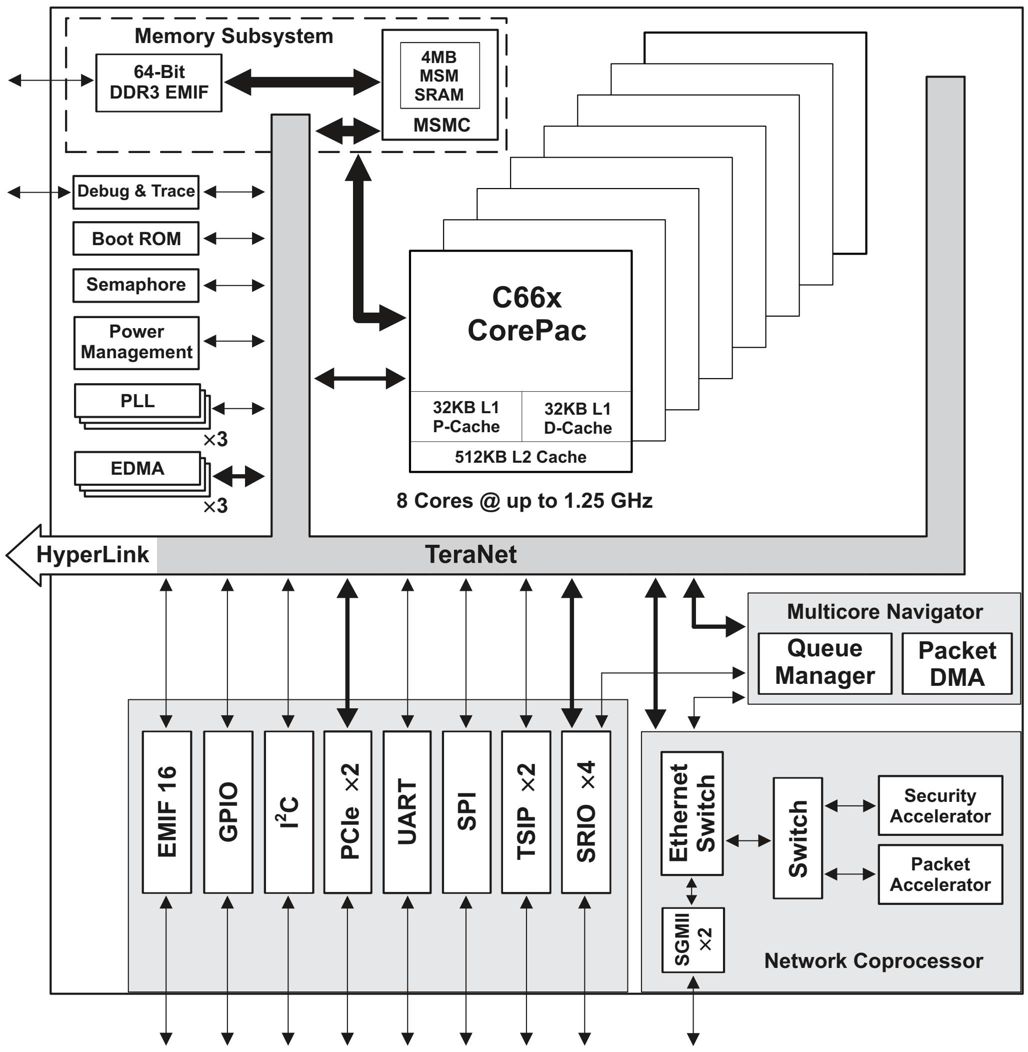


Figure 2.2: Architecture of TMS320C6678 DSP

lular phone base stations will continue to sustain its continued development while it grows in a potential secondary role as a coprocessor for supercomputing. Figure 2.2 shows an overview design architecture of TI TMS320C6678 DSP. We will separately introduce the components of micro-architecture in the following subsections.

Register File

The C66x DSP has two register files, called side A and side B. On each side there are 32 32-bits registers. These registers could be used to store either integer or single precision floating point numbers. The registers on the same side of the processor could be grouped up into 2's group (register pair) or 4's group (register quad) to store a number that requires 64 bits or 128 bits.

Microprocessor Functional Units and Instruction Sets

Each C66x DSP has two sets of functional units on each side of the register file. Each functional unit set includes 2 .M units, 2 .L and .S units, 2 .D units and a cross-path data transfer unit .X, .M units are used to perform multiplications, .L and .D units are used to performs all the other arithmetic instructions besides multiplications, include addition, subtraction, Boolean operations and bit-shift operations. The .D units are used for load and store operations. The data cross-path transfer unit .X is used to move the data between the two register files.

The DSP allows SIMD instructions and VLIW instruction packing. 2-way SIMD (Single Instruction Multiple Data) Instructions allow the processor to perform computation on a vector of 2 in one functional unit. For example, a scalar multiplication instruction "MPYSP A1, A2, A3" takes input operand value in register A1 and A2, multiply and save the result into register A3. The SIMD version of MPYSP is "DMPYSP A1:A2, A3:A4, A5:A6". It takes two register pairs as the input operands, multiplies the value of A1 and A3, A2 and A4 and save the two products into the register pair A5 and A6.

The VLIW (Very Long Instruction Word) instructions allow the processor to group up data independent instructions together so that they could be issued simultaneously. Unlike SIMD instructions, the VLIW instructions can be classified as Multiple Instruction Multiple Data (MIMD). However they require multiple functional units to be executed. For example, Two instructions "ADDSP A1, A2, A3 ($A3 = A1 + A2$)" and "MPYSP A4, A5, A6 ($A6 = A5 \times A4$)" can be grouped up into a VLIW instruction "ADDSP A1, A2, A3 || MPYSP A4, A5, A6". The " || " sign between the two instructions indicates that these two instructions are belongs to the same VLIW pack. The SIMD instructions are able to be further packed into VLIW instructions as long as there are enough functional units to lauch these two SIMD instructions simultaneously.

Memory Subsystem

Each DSP processor has its own on-chip memory. It includes a 32KB L1P instruction cache, 32KB L1D data scratchpad/cache and a 512KB L2 data scratchpad/cache. The fastest on-chip memory is the L1P instruction cache and L1D data scratchpad/-cache. The L1P instruction cache is designed specifically for instruction prefetching and is not usable by user program. The L1D on-chip memory could be either configured as program controlled scratchpad memory or cache. The L1 memory has the fastest speed. The latency of reading data from L1 memory is 4 cycles (1 CPU cycle for issue, and 3 cycles for delay slot). The L2 on-chip memory is similar to L1. However, in exchange of its larger size, Peak speed of L2 is half of the peak speed of L1.

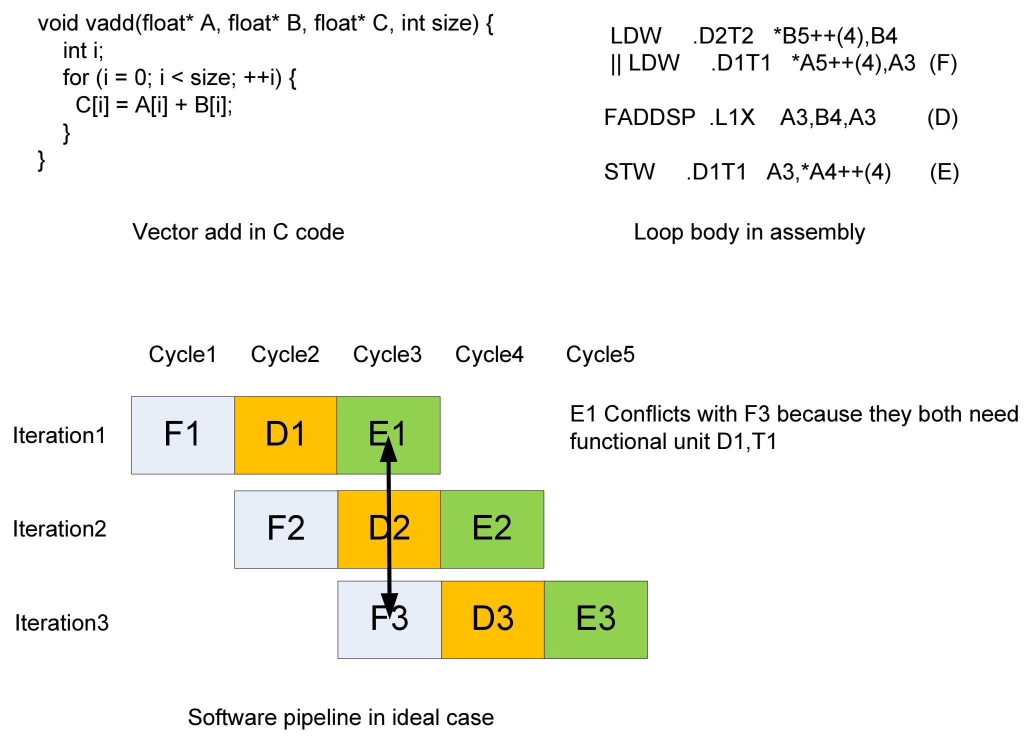
The External Memory Interface (EMIF) allows the DSP to access the off-chip DDR memory. The EMIF is shared between all the DSP cores. It supports instruction level memory fetch or Direct Memory Access (DMA), which is a hardware data movement module that allows the memory transfer from off-chip memory onto on-chip memory without stalling CPU execution.

The maximum computational throughput that the C66x DSP is able to achieve can be calculated in the following methodologies. Assume the maximum bandwidth from global memory to DSP core is a, the memory to compute ratio of the stencil loop is r_{mc}. Because the maximum number of arithmetic instructions of the DSP core can execute per cycle is $2 \times (2+2+2) \times 2 = 24$ (2 sets of functional units on each side times the number of the functional units (.L, .S. and .M unit) times the maximum floating point vector width of each functional unit). The maximum number of floating point operation can be executed per cycle is the minimum between $a \times r_{mc}/4$ and 24, determined by whether the stencil computation is memory bounded or compute bounded.

Software Pipeline

The basic building block of code optimization on TI C6678 DSP is loops. In order to fully utilize the computational resources on the processor. C66x DSP uses a compiler technique called Software Pipeline to achieve higher throughput on the data independent loops [21]. The main idea of software pipeline is to overlap loop iterations as much as possible so that the parallelized instructions from different loop iterations could be executed simultaneously. An example of software pipeline optimization is adding a constant value to a floating point array, $B[i] = A[i] + 1.0$. From this example we can see that the loop body is composed of three parts: load data from $A[i]$, perform $B[i] = A[i] + 1.0$ and store $B[i]$. In the ideal case. those three instructions could be pipelined with a step length equals to one cycle.

In a software pipelined loop, the number of CPU cycles between the launching time of two consecutive loop iterations is called Iteration Interval (II). II describes the average running time of each loop iteration and measures the code optimization efficiency. Ideally, for a data independent loop, the processor should be able to issue a new iteration in every cycle (II = 1). However since the number of functional units on the processor is limited. If the new loop iteration requires more functional units than the chip is able to allocate. It has to be stalled until the required functional units are free. The number of cycles delayed because of the limitation of functional units is called resource bound. To be more specific, the delay caused by memory operation units is called memory operation bound; the delay caused by arithmetic functional units is called arithmetic operation bound. For most of the stencil loops



```
void vadd(float* A, float* B, float* C, int size) {
   int i;
   for (i = 0; i < size; ++i) {
     C[i] = A[i] + B[i];
   }
}
```

```
LDW     .D2T2   *B5++(4),B4
|| LDW     .D1T1   *A5++(4),A3  (F)

FADDSP  .L1X    A3,B4,A3        (D)

STW     .D1T1   A3,*A4++(4)     (E)
```

Figure 2.3: An Example of Software Pipeline on Vector Add Kernel

that contain more instructions than vector add, the II is very likely to be larger than 1 because of resource bound. In Figure 2.3. We also show an example of functional unit contention between parallel iterations. In this case, iteration 3 cannot be issued until iteration 1 finish storing the value and free functional unit D1 and T1.

In the following chapters, we will focus on the methdology and implementation of our stencil optimization framework.

Chapter 3

Related Work

In this chapter, we summarize a variaty of related work on optimizing stencil loops and improving the performance of optical flow methods. Implementation of these methods is available on various of platforms including multi-core CPUs, GPUs and FPGAs.

3.1 Stencil Loop Acceleration on CPUs, GPUs and FPGAs

The code generation and optimization of stencils are studied intensively on GPUs and CPUs that allow large scale parallelization. In this section we summarize the state-of-the-art stecil loop optimization researches.

Holewinski et al. developed an automatic stencil loop generation and tuning tool for both CPUs and GPUs [18]. They developed a domain-specific language to describe stencil and a compiler which is able to translate the domain-specific language to C and CUDA code. However, their methodolody focus on minimizing the memory access rate by loop tiling and cache performance optimization and does not involve SIMD utiization and unrolling optimization. Experimental result shows that their code generation scheme is able to achieve high performance on a range of GPU architectures.

The PATUS framework developed by Christen et al. [6] is a code generation and auto-tuning framework for stencil computations targeted at modern multi- and many-core processors, such as multicore CPUs and graphics processing units. The goal of this work is to provide a means towards productivity and performance on current and future high performance computing platforms. The framework generates the code for a computation kernel from a specification of the stencil operation and a description of optimization strategy. It is leveraged to find the optimal hardware architecture-specific and strategy-specific parameter configuration. The shortcoming of the PATUS framework is that it requires the users to provide the hardware platform specifications and write both the stencil loops and the parameter configuration for auto-tuning, which adds coding complexity. Similar to the Holewinski's work mentioned previously, PATUS also does not involve optimization with SIMD instructions.

Han et al. presented in his work a PAttern-Driven Stencil (PADS) compiler-based tool in 2011 [17]. Extension of this tool applies and depicts high-level data structures in order to facilitate recognition of various stencil computation patterns. The PADS allows programmers to rewrite kernel of stencils or reuse source-to-source translator outputs as optimized stencil template codes with related tuning parameters. In addition, PADS consists of a OpenMP to CUDA translator and code generator using optimized template codes. It is able to achieve in average 2.5x speed up compared with non-optimized C code on a selected benchmark of three stencil algorithms.

Datta et al. developed an optimization and auto-tuning framework for stencil computations [9], targeting at multi-core systems, NVidia GPUs, and Cell SPUs. They proposed autotuning as essential in order to achieve performance levels on GPUs where the benefits outweight the cost of sending data across the PCIe bus. Their method is able to achieve 3x speed up compared with non-optimized version on a 3D heat equation. However, their auto-tuning tools is based on stencil parameterization and does not perform assembly-level code optimization.

The work of Tang et al. [48] proposes the Pochoir stencil compiler which uses a DSL embedded in C++ to produce high performance code for stencil computations using cache-oblivious parallelograms for parallelism. The Pochoir stencil compiler allows a programmer to write a simple specification of a stencil in a domain-specific language embedded in C++ which the Pochoir compiler then translates into high-performing Cilk code that employs an efficient parallel algorithm based on parallel space cut. And then a set of optimization such as code cloning and loop indexing are performed to generate efficient Cilk code. Pochoir supports general d-dimensional stencils and handles both periodic and aperiodic boundary conditions in one unified algorithm.

Overlapped tiling is a technique used in automatic code generation framework. It is early presented by Krishnamoorthy et al. [31] for enhancing tile-level concurrency for multi-core systems. Nguyen et al. [37] proposed a data blocking scheme that optimizes both the memory bandwidth and computation resources on GPU devices. Peng et al. [11] investigated the optimization of selection of tile sizes with an emphasis on stencil computations.

Paulius Micikevicius described a GPU parallelization of the 3D finite difference computation using CUDA [35]. In his work, a hand-tuned 3D finite difference computation stencil achieved an order of magnitude performance increase over existing CPU implementations on GT200-based Tesla GPUs. Multi-GPU parallelization is also described, achieving linear scaling with GPUs by overlapping inter-GPU communication with computation.

Since the memory access pattern of stencil loops are highly regular, hardware acceleation techniques such as FPGAs are often utilized for stencil loop speed up. Those research works many focus on optimizing the memory controller, memory access scheduling system and floating point arithmetic pipeline. Jin et al. presented his work for stencil optimization on Convey HC-1 FPGA platform [28] [27] and later publish an appication specific research on BEAGLE library acceleration [26].

In summary, the state-of-the-art stencil loop compilers are mainly targeted at code generation for multicore systems such as OpenMP and CUDA platforms. Most of those work focus on optimizing loop tile size and increasing data reuse rate (cache performance). None of these work provides a solution with instruction-level parallelism and improve the memory bandwith utilizing on-chip memory such as scratchpad.

3.2 Improve System Performance by Scratchpad Memory

Unlinke CPUs, embedded systems allow users to manipulate on-chip memory explicitly. Those on-chip memory called stratchpad memory, are 3-4 times faster than off-chip memory, but relatively small. In this section, we summarize the literatures that work towards improving scratchpad memory efficiency on embedded system.

Kandemir et al. presented a compiler strategy to optimize data accesses in regular array-intensive applications running on embedded multiprocessor environments [29]. Their optimization algorithm targets at reducing extra off-chip memory accesses caused by inter-processor communication. It is achieved by increasing the application-wide reuse of data that resides in the scratchpad memories of processors. The experimental results obtained on four array-intensive image processing applications indicate that exploiting inter-processor data sharing can reduce the energy cost by as much as 33.8% (and 24.3% on average) on a four-processor embedded system. However, Their memory scheduling method is based on program I/O and did not utilize hardware memory transfer functionalities such as Direct Memory Access (DMA). This causes two problems: (1) Memory bandwidth can not be fully utilized. (2) CPU is stalled when data copy is being processed.

Chattopadhyay et al. works towards reducing scratchpad usage through bus address sharing. In his work, a compile-time scratchpad allocation framework for multiprocessor platforms is developed where the processors share on-chip scratchpad space and external memory. The allocation method considers the waiting time for bus access while deciding which memory blocks to load into the shared scratchpad memory space. Zhang et.al proposed a partitioning heuristics scheme for scratchpad allocation based on High Access Frequency First (HAFF) variable partitioning and Global View Prediction (GVP) variable partitioning [52]. A loop pipeline scheduling was developed to improve the overall memory efficiency. Ozturk et al. proposed a control flow graph based technique to reduce the scratchpad memory usage on multicore embedded system. It tracks the lifetime of instructions at the basic block level. Based on the CFG analysis, if a basic block is known to be not accessible in the rest of the program execution, the instruction memory space allocated to this basic block is reclaimed. All these proposed works focus on reducing memory usage and does not emphasize performance issue.

Scratchpad memory are useful on embedded system or special propose processor such as System-On-Chip. For the application which has a complicated memory structure, How to partition and assign scratchpad memory so that the overall system performance is optimized is still an open problem for research. Yang et al. studies the scratchpad structure of TI Keystone2 and present their application specific study on sparse matrix multiplication [15]. Yan et al. designed and optimized FPGA accelerator with customized scratchpad memory architecture to accelerate frequent itemset mining algorithm [53] [54].

3.3 Accelerating Lucas-Kanade Method on Various of Platforms

One of the most influencial work that optimizes Lucas-Kanade method on multicore CPU is published by Anguita et al. [1]. In the paper, a high-performance implementation of Lucas-Kanade method that takes advantage of the multicore processor's architecture is presented. Their optimized implementation is highly interesting for a number of applications since it delivers real-time motion estimations at high-image resolution on a PC or in an embedded system based on a general-purpose processor. On a 2.83 GHz Core 2 Quad PC, it achieves a speedup of 14x compared to the baseline version and 2052.7 fps for the well-known 252x316 Yosemite sequence, and a speedup of 17.6x and 68.5 fps for a 1016x1280 sequence. Their optimization of Lucas-Kanade method in the paper is separated into two stages. In stage 1, an error-free baseline version is implemented in C language with single floating point matrix type and optimized primitively by compiler options and cache parameter tuning. A complete combination of the compiler options are studied and compared to guarantee the optimal running time. Beside tuning compiler options, author rounds up the matrix width used in the code with L2 cache line size of the CPU to reduce the cache conflict miss rate. The first step optimization is able to achieve 25fps tested on the Yosemite sequence.

The stage2 optimization includes:

- Cloning of the convolution functions. The convolution functions implemented take advantage of the symmetry of the convolving filters (except the row convolution in the smooth stage). Furthermore, the smooth stage is implemented with two nested loops (loop fusion).

- Loop unrolling is applied to the convolution operations.

- OpenMP directives have been explicitly added in order to distribute the computation between different threads and thus take advantage of the multiple cores.

- Block-driven processing is applied to reduce the negative impact of the image sizes on the processing time. The algorithm is changed in order to process the images in blocks. In this way, data locality is improved by taking advantage of the cache.

Marzat et al. implemented CUDA based Pyramidal Lucas-Kanade method [34]. The paper tackles the problem by proposing a parallel implementation of the pyramidal and iterative refinement algorithm of Lucas-Kanade in a Graphics Processing Unit (GPU). It is able to compute a dense velocity field at about 15 Hz with a 640x480 image definition using the Compute Unified Device Architecture interfance. It parallelizes the velocity computation across 512 CUDA threads and utilizes hardware acceleration module provided from CUDA Library to compute Gaussian blur and flow bilinear interpolation. Their CUDA implementation achieves 100x speed up compared with an optimized CPU version and 15 fps on 640x480 RGB images.

NVidia provided many GPU accelerated computer vision kernel functions based on OpenCV library. Pyramidal Lucas-Kanande methods is implemented and integrated into OpenCV library [38].

Devi et al. design an hardware Lucas-Kanade architecture with multi-scale extension on FPGA device simulation [10]. Their method is based on software simulation model and provided with simulation results.

Diaz et al. developed a pipeline architecture for non-pyramidal Lucas-Kanade method. In their work. Lucas-Kanade method is divided into a 5 phases, over 70 pipeline stages (Super Pipelining). Since implementing a full pipelined optical flow uses considerable amount of gate resources on the board. They simplify the design by using fix point arithmetic. Their super pipeling model is developed on the Virtex II XC2V6000-4 Xilinx FPGA and achieves 170 fps on 800x600 video stream [24] [33].

Besides differential method. Other methods for optical flow estimation include phase-based methods [41] [49] [16], tensor based and block matching methods [36] [25] [45] [50].

Although nowadays Digital Signal Processors are growing more and more powerful in numerical data processing and are able to achieve better power efficiency compared with CPUs and GPUs, very few work has been published on accelerating computer vision tasks on those platforms. Basavaiah presented his paper of building object tracking system on embedded processor in 2012, but his work many focuses on system development and does not put much emphasis on to performance study [2]. Hutchings et al. use a different platform called Ambric Massively Parallel Processor Array (MPPA) as the hardware accelerator to implement real-time Lucas-Kanade optical flow method and achieves 4x speed up [19].

Comparison studies of optical flow performance between different platforms are available in the following published work [5] [40].

In summary, current stat-of-the-art research of accelerating Lucas-Kanade method are all based on multicore CPU, GPU and FPGA. Our proposed work is the first one that concerns how to improve the performance of the Lucas-Kanade method on a multicore embedded processor.

Chapter 4

Approach

In this chapter, we provide the detailed design of our stencil code generation, optimization and scratchpad memory management methodologies and implementation.

4.1 Performance Metrics

The performance of a stencil loop can to be characterized using the following metrics.

- Iteration Interval, or II: It is the number of cycles between two consecutive iterations are issued. II is most important metric since it is a direct throughput measurement of the loop. Because the stencil loop iterations are completely data-independent, the II is determined by resource bound such as functional units and register usage. As shown in Figure 4.1, Because both of the memory functional units are occupied in cycle 3, the launch of a new loop iteration must be delayed until one of them is free. The same II delay could also be caused by arithmetic operation unit contention. We separately discuss the two factors that contribute to II, which are memory operation bound and arithmetic operation bound.

- Operations per iteration: It characterizes the effectiveness of the code generation. It is the total number of operations in the loop. Scalar instruction counts as 1 operation and n-way SIMD instruction counts as n operations. Operations per iteration can be retrived by counting from the backend generated assembly code.

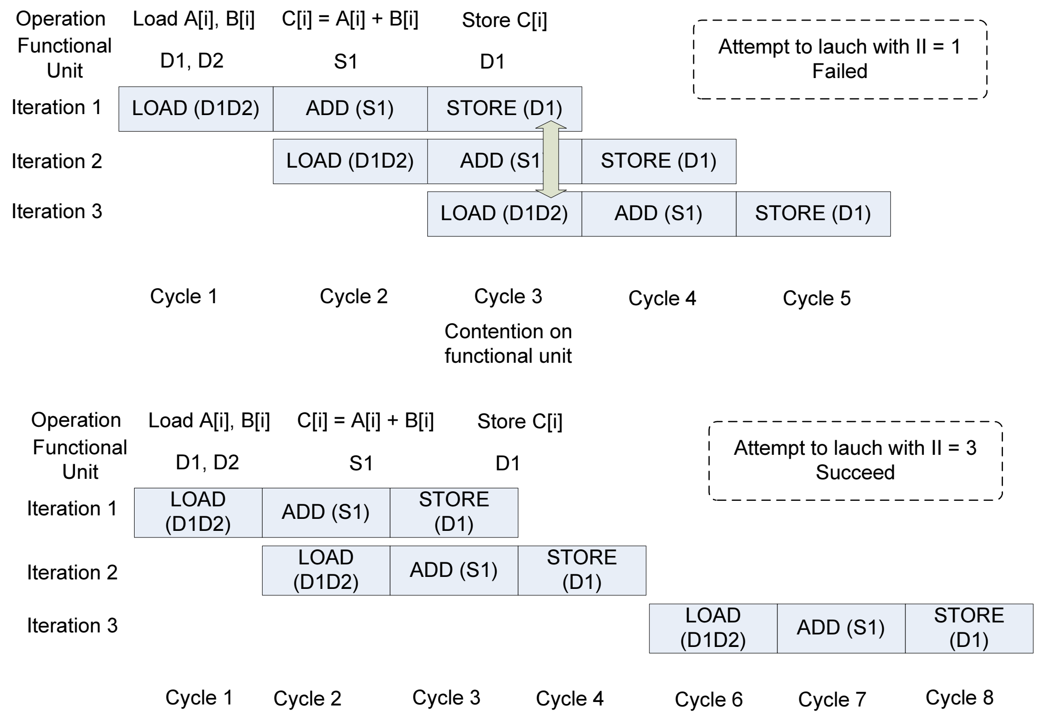


Figure 4.1: Example of II Increasing Caused by Functional Unit Allocation

- Operations per cycle: It is the number of operations executed per cycle. It measures the VLIW and SIMD level parallelism of the generate code. The operations per cycle is computed from the operations per iteration divided by II.

- Allocated register number: It measures the register resource usage of the generated code. This number is computed by parsing the assembly code of the loop body and counting the unique number of registers.

4.2 Stencil Code Generation and Optimization

The stencil code generation and optimization infrastructure we developed is composed of two components.

- A Domain Specific Language called Position Independent Arithmetic (PIA) and a frontend compiler that generates code from PIA syntax to LLVM intermediate representation.
- A code optimization engine that drives the optimization process on the LLVM intermediate representation.

The LLVM is a compiler infrastructure designed as a set of reusable libraries with well-defined interfaces. It is written in C++ and is designed for compile-time, link-time, and run-time optimization of programs written in arbitrary programming languages. We list more details of LLVM in the Appendix B section of the dissertation.

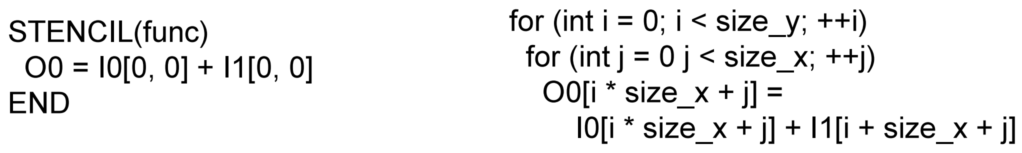
The main motivation of separating the design of the stencil loop code into a Domain Specific Language is to simplify the design of stencil loop body and take the control flow management away from the programmer, given the fact that the identical computation must be done on each entry of the output matrix. An user of PIA is only required to specify the computation performed on each output matrices' entries.

The user-defined code section is called the stencil body. We design a positional index system to refer to the elements from the input matrices. More complex stencils that contain multiple inputs and outputs, or dimesion higher than 2 are also supported.

Syntax Design of the Position Independent Arithmetic

In order to illustrate how the stencil loops are represented in PIA format, we show a hello world program in Figure 4.2. It shows a matrix add kernel in PIA. As a comparison, we also show the equivalent C code implementation in the figure.

To be more specific, a PIA stencil loop function is declared by starting with "STENCIL(func_name)" and ending with "END". The "STENCIL" and "END" are



```
STENCIL(func)
  O0 = I0[0, 0] + I1[0, 0]
END
```

(a) Stencil loop in PIA

```
for (int i = 0; i < size_y; ++i)
  for (int j = 0 j < size_x; ++j)
    O0[i * size_x + j] =
      I0[i * size_x + j] + I1[i + size_x + j]
```

(b) Stencil loop in C

Figure 4.2: Matrix Add in PIA and C

two of the reserved keywords in PIA. The “func_name” specified the name of the stencil function and it is exported as the module function name in the generated intermediate code and LLVM bytecode file. User code calls to PIA generated function will need to used it as the function name.

In the PIA stencil body, standard arithmetic operations such as “+”, “-”, “*” and “/” are supported. Assignment operation is performed by “=” operator.

There are 4 different types of identifiers in PIA syntax. They are “input field”, “output field”, “local variable” and “parameter”. Input fields and output fields are used to refer to the elements in the input and output matrices. They are named by ANSI C standard, starting with letter or underline and followed by letters, underlines and numbers. Each input or output field is parsed into a floating pointer parameter in the output function’s parameter list.

The input fields can be used together with a dimensional offset modifier [_, _, ... _]. For example I0[-1, -1] means the elements to the top left position of the current position in matrix I0. The output fields are not allowed to be used with the offset modifier. The input fields are read-only. The output fields are write-only.

The local variables are identified by a starting character “$”. They are used to store the intermediate results. For example “$a = I0[0, -1] + I0[0, 1];”. This statement takes the left and right elements from the current position and sum them up into variable $a.

Parameters begin with “@”. They are used when the stencil loops need values to

```
translation_unit
    : START_STENCIL '(' IDENTIFIER ')'  stencil_statements  END_STENCIL ;
stencil_statements : stencil_statements stencil_statement ;
stencil_statement : stencil_lval '=' stencil_rval ';'       ;
stencil_lval : LOCAL_VARIABLE  | OUTPUT_FIELD    ;
stencil_rval : stencil_expression ;
input_field  : INPUT_FIELD '[' stencil_offset ',' stencil_offset ']' ;
stencil_offset : '-' CONSTANT_INT_DEC | CONSTANT_INT_DEC ;
stencil_expression
    : stencil_expression '+' stencil_mul_expression
    | stencil_expression '-' stencil_mul_expression
    | stencil_mul_expression
stencil_mul_expression
    : stencil_mul_expression '*' stencil_primary_expression
    | stencil_mul_expression '/' stencil_primary_expression
    | stencil_primary_expression;
stencil_primary_expression
    : '(' stencil_expression ')'
    | CONSTANT
    | input_field
    | LOCAL_VARIABLE
    | INPUT_VARIABLE ;
```

Figure 4.3: The Context Free Grammar of PIA in Bison

be passed in from outside. They are read-only and parsed and listed as parameters of the output function declarations.

Each statement in PIA ends with a semi column.

In order to better demonstrate the design of PIA syntax. Figure 4.3 list the context free grammar of PIA.

The PIA stencil code is parsed by GNU Bison and Flex. The abstract syntax tree is generated after code parsing. In the PIA syntax tree, there are three types of syntax nodes. (1) value nodes, these nodes represent either a variable or a constant, (2) operation nodes: these nodes represent arithmetic operations, (3) assignment node, these nodes represent assignment operations. Each syntax code is provided with a code generation method so that corresponding LLVM IR code can be generated with a post order traversal of the syntax tree.

In the Figure 4.4, we list the major components of the LLVM code generation, which includes syntax tree generation, stencil margin detection, loop structure gen-

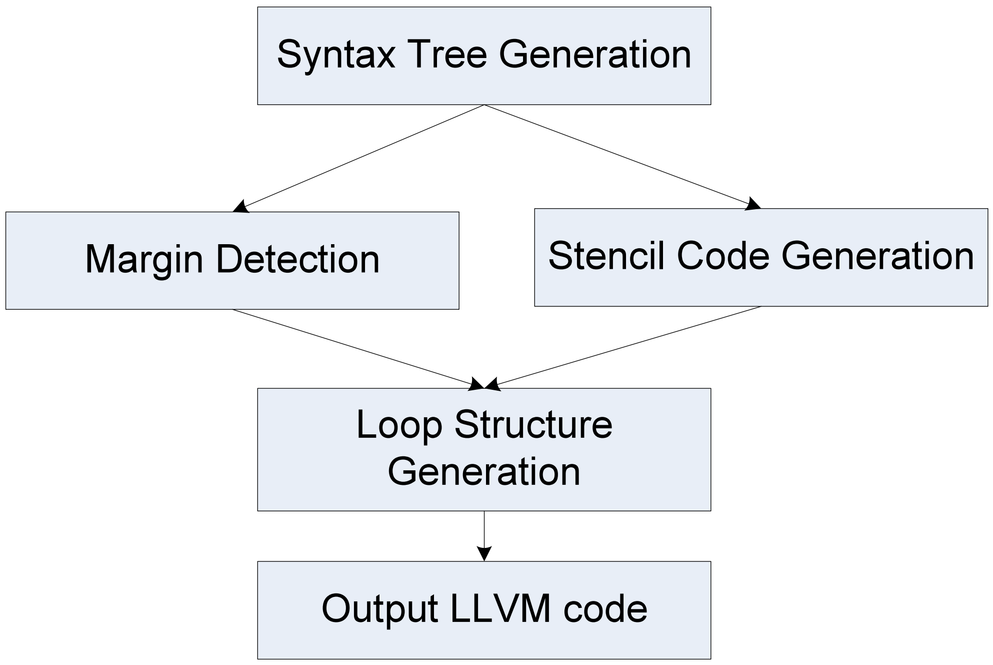


Figure 4.4: The System Overview of the PIA Compiler

eration and stencil body generation.

Stencil Loop Margin Detection

The margin of a stencil is the neighbor window size of the computation on each matrix entry. For example, the computation structure of a 2D Jacobi stencil is shown in Figure 4.5. Each point in the output matrix is computed by averaging adjacent 5 points (self, top, bottom, left and right neighbor) in the input matrix. The margin has to be detected before code generation because the loop starting and ending conditions are dependent to the margin size. The four margin values of the stencil loop can be computed from the largest offset values from the modifiers fields in the input field nodes.

The margin values need to be collected before control flow structure generation because they are needed for construction of the starting and ending condition of the loop.

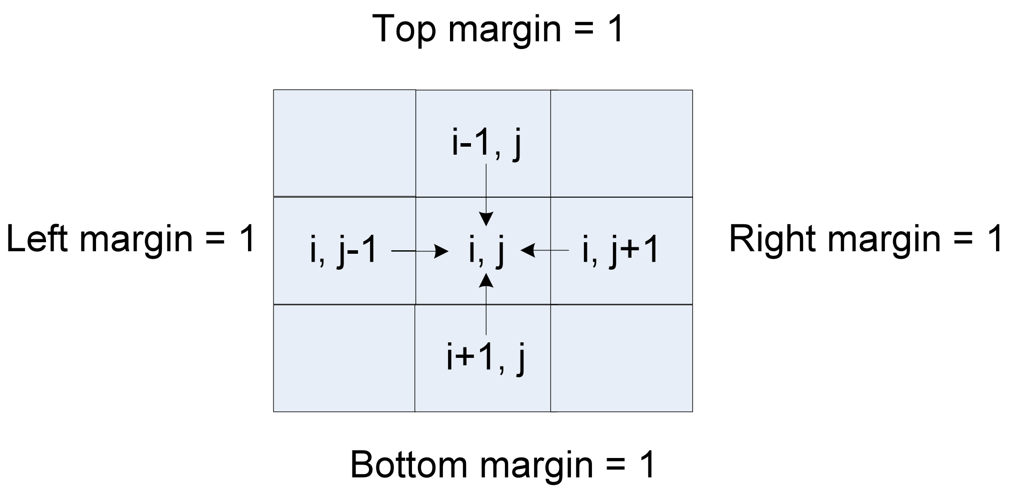


Figure 4.5: A Stencil Loop Example (2D Jacobi) with Margins Sizes Equal to 1

Control Flow Structure Generation

After the stencil margins are detected, those values are used to generate the multi-level nested loop structure. Wach level of the nested loop is consisted of 4 basic blocks labeled Preheader, Loop, LoopCond and AfterLoop. The Preheader block initializes the loop variables and then it passes the control flow into the Loop block. The Loop block is generated from parsing the PIA stencil body into arithmetic operations in LLVM intermediate representation format. The LoopCond block is executed after the Loop block. It updates the value of the loop variables and evaluates the loop condition to decide whether the loop should be started over or exit.

We use a recursive algorithm to generate nested loop structure. The input of the algorithm is the margin sizes of x and y direction from margin detector and the abstract syntax tree. The detailed design of the LLVM code generation is listed in Appendix B of this dissertation.

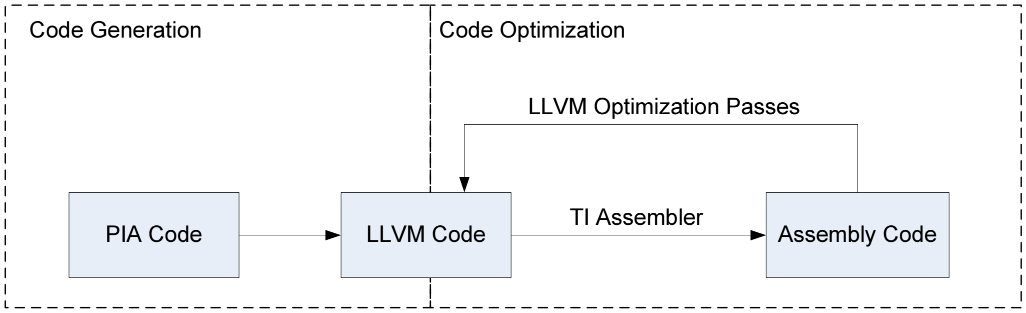


Figure 4.6: An Overview of Code Optimization Engine

An Overview of Stencil Loop Optimization in LLVM

The code generation methodology we have introduced previously is able to generate executable code. In the following sections we focus on discussing how the code optimization is performed and in which way the performance of the code could be optimized. The basic idea of the code optimization is to search the optimized code transformation strategy based on the feedback from the assembly code generator.

The Figure 4.6 shows an overview of the code optimization searching strategy. The code optimization is separated into two parts: static optimizations are always executed, such as constant propagation, instruction reassociation, common sub-expression extraction, and dynamic optimizations are parameterized and executed conditionally based on the feedback information from the backend assembler. In the following section, we focus on introducing the dynamic optimization strategies. The dynamic optimization strategies we developed and implemented are loop unrolling and SIMD binding.

Loop Unrolling

Loop unrolling increase the number of instructions in the loop body and provide more space for the assembler to perform VLIW parallelization. In order to perform loop unrolling, the loops can be rewritten as a repeated sequence of similar independent statements.

Programmatically perform loop unrolling could be more challenging than doing it manually. The unroll procedure need to decide:

- How the loop is structuralized. Generally before a loop can be unrolled. The three sections has to be identified, phi node instructions, loop condition and loop variable updating instructions and loop body.

- How to duplicate the loop body. Unroll the loop body is not as simple as writing the loop body twice. In order to generate valid code, new variables need to be allocated for the unrolled loop. This includes how to keep track of the operands of all the instructions and decide which operand should be referred to.

- We need to perform the loop unrolling on the LLVM IR level. which is an assembly-like formatted code. This means we cannot use most of the structuralized information from the syntax parsing.

An example of how loop unrolling could optimize the code efficiency is shown in Figure 4.7, In the figure, the instructions concatenated by "||" sign means they are parallelized into a VLIW instruction. After the loop that performs vector add is unrolled by 2, The homologous instructions could be grouped into one VLIW pack and issued in parallel and the running time of the loop is reduced.

The unroll methodology we developed is composed of three major steps: loop variable adjustment, instruction duplication, and middle block generation. In order to expand multiple iterations of the loop into one. We must scan over every instruction of the loop body and build an operand registration list to store the mapping relation of each operand in the original loop and its duplications in the unrolled loop.

The loop unrolling algorithm is shown in Algorithm 2. At the beginning of the algorithm, we use the LLVM loop information analysis module to find the loop variable and manually spawn it into duplicates (line 4). For example, if we unroll a loop

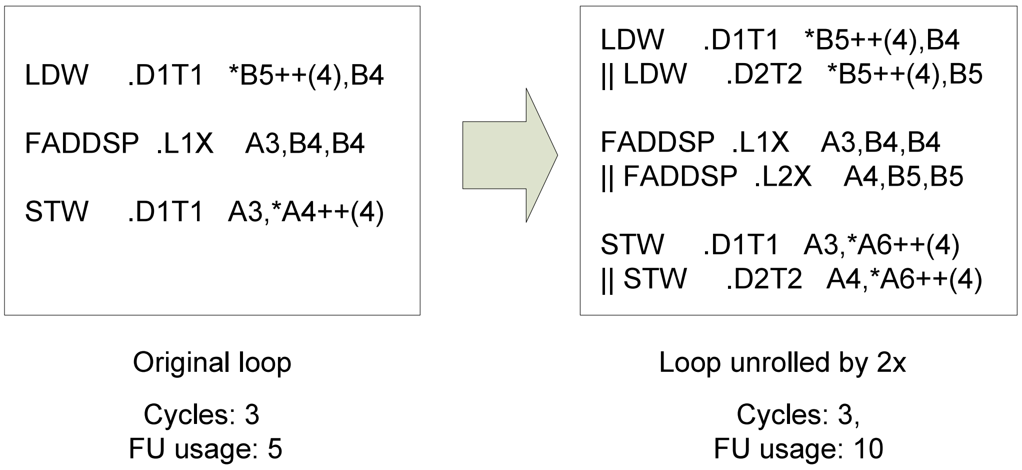


Figure 4.7: Increase Functional Unit Utilization Rate by Loop Unrolling

with loop variable i with a factor of 2, the i will be duplicated to $i1 = i$ and $i2 = i+1$ (line 5 to 8). The next step is to start from the beginning of the loop, duplicate the phi node instructions into the new loop (line 10 to 12), then iterate over all the instructions in the original loop, look up and duplicate them into the new loop, The operands of the duplicated instructions are looked up from the operand registration list (line 13 to 19). At the last, we insert the instructions for the loop variable updating into the new loop (line 21). Instead of increment by 1, the loop variable is now increment by N. The branch instruction is attached at the end of the loop body. It lead the instruction flow into a middle block (line 22, 23).

Because the unrolled loop only process the loop iteration numbered in the multiple of the unroll factor. For example, if a loop with 17 iterations is unrolled in a factor of 4. It will stop at iteration 15 and leaves iteration 16 undone. The middle block compare the value of the loop variable and the loop condition and process the remained iterations. The exit of the middle block is set to be the exit of the unrolled loop.

Algorithm 2 Loop Unrolling Algorithm

```
1: **Input:** Original loop in LLVM format, Unroll factor $N$
2: **Output:** Unrolled loop in LLVM format
3: Initialize operand registration list $m$
4: Identify loop variable $LoopVar$
5: $m[LoopVar] = [LoopVar[0], LoopVar[1], ...LoopVar[N-1]]$
6: for $i = 0$ to $N-1$ do
7:     Generate LLVM code that assigns $LoopVar + i$ into $LoopVar[i]$
8: end for
9: for each instruction $i$ in the original loop do
10:     for each operand $o$ in $i$ do
11:         if $m[o] == NULL$ then
12:             Allocate $N$ new variables $[v0...vN]$ and let $m[o] = [v0...vN]$
13:         end if
14:     end for
15:     Generate duplicated instruction from $i$ and the operand registration list $m$
16: end for
17: Update the loop condition blocks
18: Generate middle block
19: Generate exit block
```

SIMD Binding

The second optimization we perform is SIMD binding (or called loop vectorization). This optimization is often combined together with loop unrolling. Once the loop is unrolled. The homologous instructions are expanded multiple times in the same loop iteration, but with different operands. These homologous instructions can be grouped up into SIMD instructions and reduce the functional unit occupation given that the SIMD instruction is supported on the processor's micro-architecture.

Like loop unrolling, SIMD binding also requires programmatically transforming the LLVM's IR code. The challenges of performing SIMD binding includes:

- Distinguish vectorizable instructions and transform them into the equivalent SIMD instructions. Non-vectorizable instructions such as comparison, boolean and branch insturctions are unrolled and scalarized.

- Group scalar variables into vector varables. As the instructions are SIMD binded, the operands of those instructions need to be transformed into vector form. It is required that the instructions such as “insertelements” are automatically inserted into the code to build vector values from scalar values.

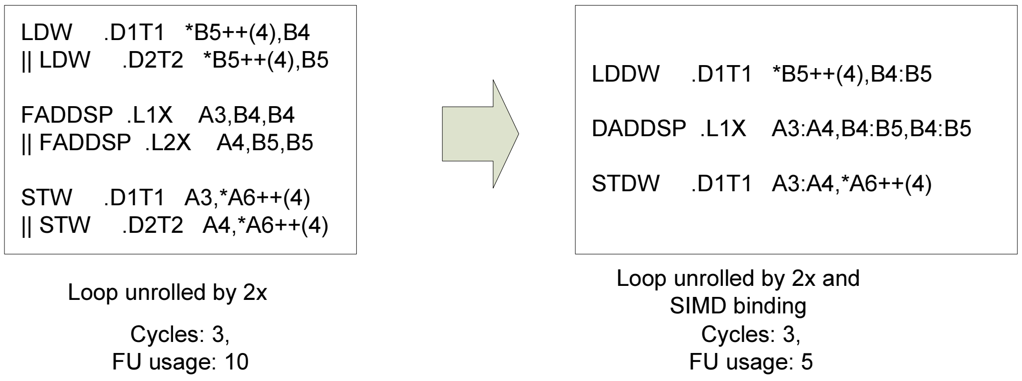
- Generate C66x compatible SIMD instructions. As the C66x assembler does not recognize LLVM vectorized memory instructions, for examples, the 2-way SIMD load/store instructions. We need to explicitly generate C66x intrinsic function call in the LLVM IR level. This requires to add a translation layer on the top of the SIMD binding LLVM code, that distinguishes those instructions into C66x intrinsic function call.

In order to perform the SIMD binding, the loop is firstly unrolled by the vector width of the micro-architecture, for TI C66x DSP this number is 2. We search and combine the homologous instructions together as SIMD instruction in LLVM. The SIMD binding are targeted at two types of instructions: Memory instructions and arithmetic instructions.

Figure 4.8 shows an example of SIMD binding after loop unrolling by 2x. The SIMD binding process is able to convert the scalar load and store instruction LDW and STW into 2-way SIMD load and store instructions LDDW/LDNDW and STDW/STNDW, and convert the scalar addition instruction FADDSP to DADDSP

We design a Function Pass module called PiaVectorize and add it to the LLVM source code repository. The PiaVectorize module always vectorize code in a width of 2, because that is the largest width that TI C66x DSP supported.

The PiaVectorize first calls the LLVM loop analysis module to locate the innermost loop body and the loop variable, then it enumarates all the homologous memory operations and arithmetic operations and binds them into into SIMD operations, In order to make the TI assembler backend to correctly translate LLVM code into SIMD instructions, it generates inline intrinsic calls for the SIMD memory operations


```
LDW    .D1T1   *B5++(4),B4
|| LDW    .D2T2   *B5++(4),B5

FADDSP  .L1X    A3,B4,B4
|| FADDSP  .L2X    A4,B5,B5

STW    .D1T1   A3,*A6++(4)
|| STW    .D2T2   A4,*A6++(4)
```

```
LDDW    .D1T1   *B5++(4),B4:B5

DADDSP  .L1X    A3:A4,B4:B5,B4:B5

STDW    .D1T1   A3:A4,*A6++(4)
```

Figure 4.8: Increase Functional Unit Utilization Rate by SIMD Binding

"_memd8()".

As shown in Algorithm 3, the SIMD binding function reads in the structure of the loop (Line 4), create a new vectorized loop and replace the old one with the new one. First, phi node instructions are identified and scalarized (Line 6). Next, If the consecutive instructions are from the same parent from the loop unrolling, the algorithm first check if they are vectorizable. If so, they will be combined into a SIMD instruction (Line 13). If the instructions are memory operations, an address casting instruction from single data pointer into vectorized data pointer is inserted before the vectorized instruction. In order to correctly invoke C66x backend assembler to generate vector load and store instructions, A call instruction that invokes the intrinsic function _memd8() will be generated instead of the LLVM vector load/store (Line 10). After all of the instructions in the loop body are completed then we adjust the incoming edges to the updated phi nodes (In Algorithm 3, Line 16). At the last, we output the vectorized basic blocks into the new loop body.

Algorithm 3 SIMD Binding Algorithm

```
1: Input: Original loop in LLVM format
2: Output: Vectorized loop in LLVM format
3: Unroll all the innermost loops by a factor of 2
4: for each innermost loop L in the function do
5:     Check if L is vectorizable
6:     Identify and scalarize the phi node and loop condition check code
7:     for each vectorizable instruction pair $i$ in L do
8:         if i is a memory instruction then
9:             Static cast the address to vector format
10:            Generate LLVM code that invokes the _memd8() intrinsic function
11:            Fix address alignment
12:        else if i is an arithmetic instruction then
13:            Merge the instructions into SIMD operation
14:        end if
15:    end for
16:    Update the branch instruction so that the loop will jump to the fixed phi node
17: end for
```

4.3 Improve Memory Throughput by Compile-Time Alignment Detection

There are two types of vectorized load/store instruction in TI C66x instruction set, Aligned 2-way SIMD load/store LDDW and STDW, and unaligned 2-way SIMD load/store LDNDW/STNDW. The aligned load and store must be used for accessing memory data from an 8 byte aligned address and they are faster than the unaligned load/store. Generally, all the memory operations are interpreted as the unaligned operation. However, if we are able to use some of the prior knowledge from the input data, for example, in the matrix add kernel, the programmer of PIA tells the compiler that the starting address of all the rows of the input and output matrices are aligned with 8 bytes. In the vectorized version of the stencil, the PIA compiler can safely interpret all the memory operations as aligned.

We add into PIA syntax an additional field called "row align" declaration, located after the declaration of the stencil header, for example"STENCIL(func) [row_align =

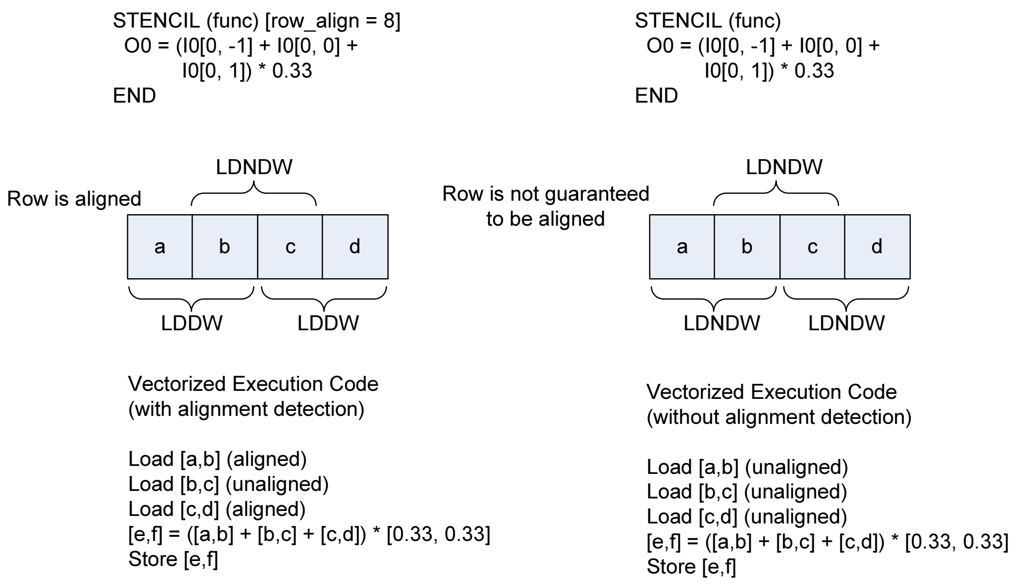

STENCIL (func) [row_align = 8]
O0 = (I0[0, -1] + I0[0, 0] +
I0[0, 1]) * 0.33
END
STENCIL (func)
O0 = (I0[0, -1] + I0[0, 0] +
I0[0, 1]) * 0.33
END
LDNDW
Row is aligned
a
b
c
d
LDDW
LDDW
LDNDW
Row is not guaranteed to be aligned
LDNDW
LDNDW
Vectorized Execution Code (with alignment detection)
Load [a,b] (aligned)
Load [b,c] (unaligned)
Load [c,d] (aligned)
[e,f] = ([a,b] + [b,c] + [c,d]) * [0.33, 0.33]
Store [e,f]
Vectorized Execution Code (without alignment detection)
Load [a,b] (unaligned)
Load [b,c] (unaligned)
Load [c,d] (unaligned)
[e,f] = ([a,b] + [b,c] + [c,d]) * [0.33, 0.33]
Store [e,f]

Figure 4.9: An Example of Applying Alignment Detection on the PIA Code of an1x3 Mean Filter

8]". The row align attribute tells the PIA compiler that the starting address of each row from the input and output matrices is aligned with the number of bytes specified. If the stencil matrices are specified by row_align = 8, when the code generation is performed, The address alignment will be calculated and inserted into each load and store instruction's metadata field. When the SIMD binding procedure is performing the code vectorization. It reads the alignment information from the metadata and generate intrinsic function call to _amem8 (LDDW/STDW) for alignment = 8 or _mem8 (LDNDW/STNDW) for alignment = 4. This optimization methdology is called compile-time alignment detection. An example of compile-time alignment detection is shown in Figure 4.9.

4.4 Finding the Best Optimization Stragegy

The dynamic optimization engine iteratively scan the search space of loop unrolling factor $UF \in \{1...N\}$ and vectorization factor $VF \in \{1, 2\}$. The upperbound of UF theoretically could be as large as possible, but it is oberserved that the performance of the loop unrolling would drop normally after 8x unroll. Our iterative search procedure parses the output information from the generated assembly code. It stops searching forward with increasing loop unrolling when it observes the output assembly code reporting that the register allocation is exceed the hardware limitation. The parameter optimization procedure output the assembly code with the best iteration interval result.

In the iterative optinization method, the unroll factor UF stops to increase based on the three condition listed below.

- The iteration interval converges. In order to avoid to stop iteration at a local minima, firstly we allows the user to manually set up the minimal number of unroll that us need to be tried. Secondly we allow a tolerant factor. The iteration is ended when the iteration iterval stops to decrease in consecutive ϵ unroll factors.
- If the register number of the unrolled loop exceeds the processor limit, the iteration stops.
- If the generated loop disqualifies software pipeline, the iteration stops.

Improve Memory Performance by Loop Tiling and Scratchpad DMA

As shown in Figure 4.10, the memory performance of stencil could be low when the computation is performed within a multi-dimensional block. Accessing data in a combination of row and column order will result in a large amount of cache misses. Because of the structure of cache is linear, the horizontal overlapping read could be effectively caught by cache, but the vertical overlapping read could encounter a high miss rate. Since the vertical overlaps are between the data access of two consecutive rows, as the computation is performed row-wisely, it is very likely when the vertical

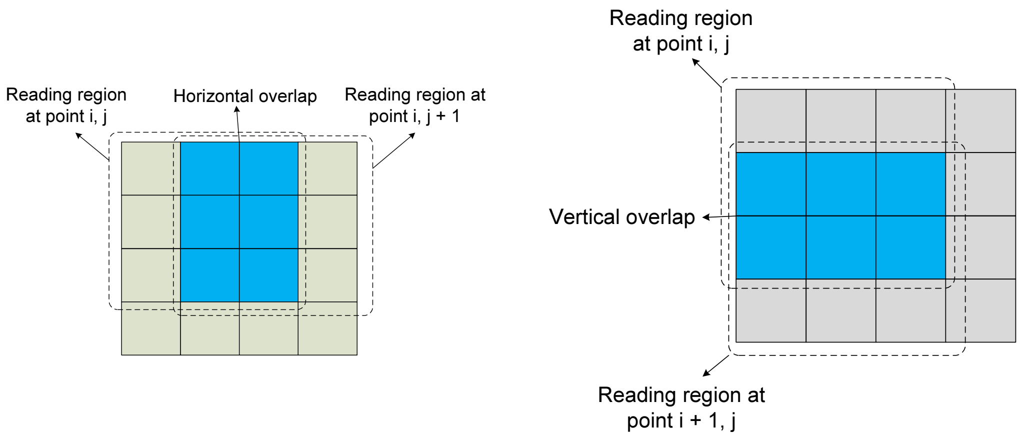

Reading region at point i, j
Horizontal overlap
Reading region at point i, j + 1
Reading region at point i, j
Vertical overlap
Reading region at point i + 1, j

Figure 4.10: Data Overlapping in Stencil Loop Computation

overlapping region is revisited, they have been moved out from the cache and causes capacity misses.

In order to improve the memory performance, we use the on-chip memory as programmer controlled scratchpad memory and build an automatic tiling and buffering infrastructure for the stencil loops. We further generalized this idea to a set of interdependent stencils, and extend our tiling and buffering system in order to solve stencil pipelines.

In order to align the output of each tile output, the tiles of the source matrix are overlapped. The overlapping region between consecutive tiles is decided by the margin size of the stencil. For example, the halo width of a 3x3 mean filter is $w_x = 2, w_y = 2$.

We illustrate how to tile and optimize stencil loops with a 3x3 mean filter example in Figure 4.11. The computation of stencil loop is split into blocks. Each block is copied into the scratchpad memory by the hardware DMA (Direct Memory Access) module. The computation of the stencil is performed on scratchpad memory and the results are copied out to the destination matrix by DMA. In order to make the scratchpad memory easier to use, we implement a memory manager module to manage the allocation and recycling of the memory space.

Buffering the data in program controlled blocks performs better than cache, since

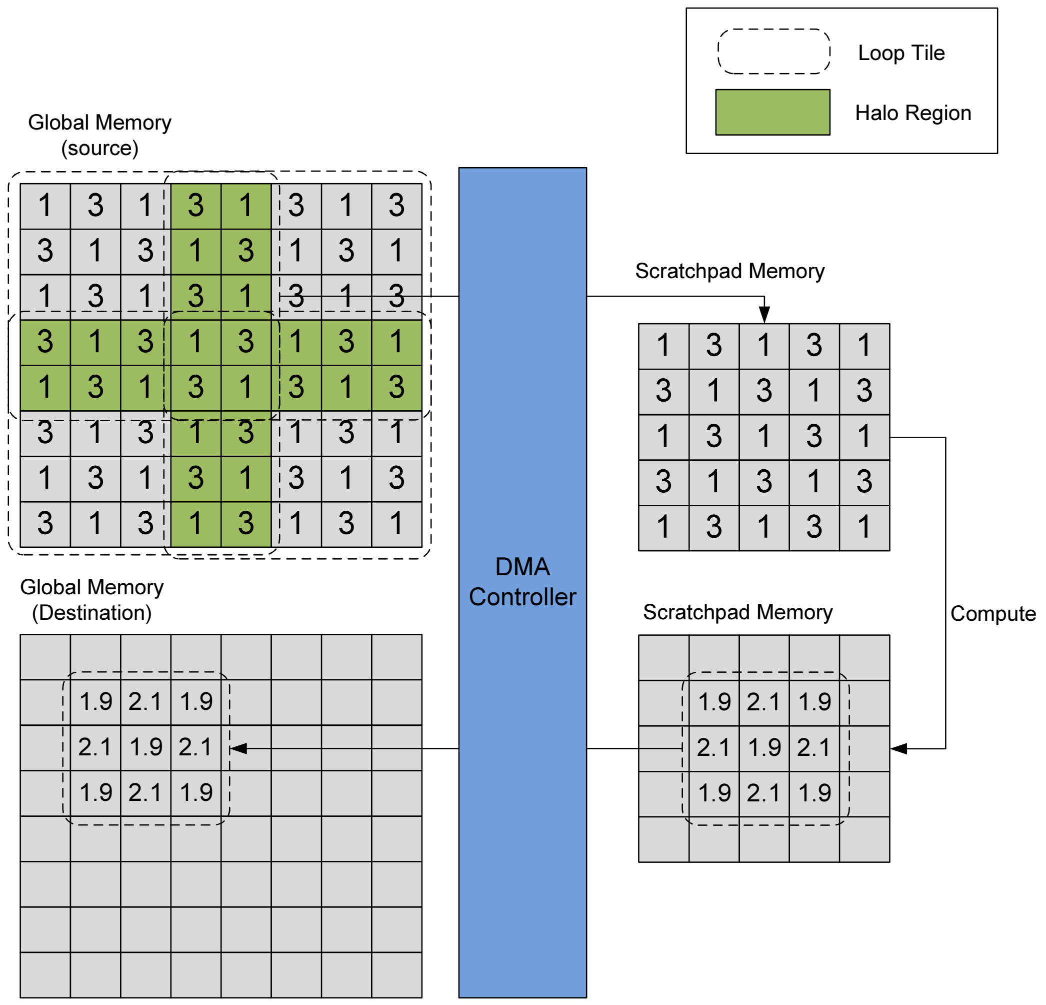


Figure 4.11: Stencil Loop Tiling and Processing on Scratchpad Memory

the vertical overlapping region stays in the on chip memory as long as it is needed, whereas using cache could results in large number of capacity misses.

Hiding Memory Transfer Latency by Double Buffer Rotation

The DMA memory transfer can be performed simultaneously when CPU is processing data. The further improvement of the scratchpad buffering is to allocate two buffers and execute computation on one buffer and data transfer on the other buffer at the same time. This technique is called double buffer rotation. Double buffer rotation is able to hide the memory transfer latency and is effective on optimizing the stencil

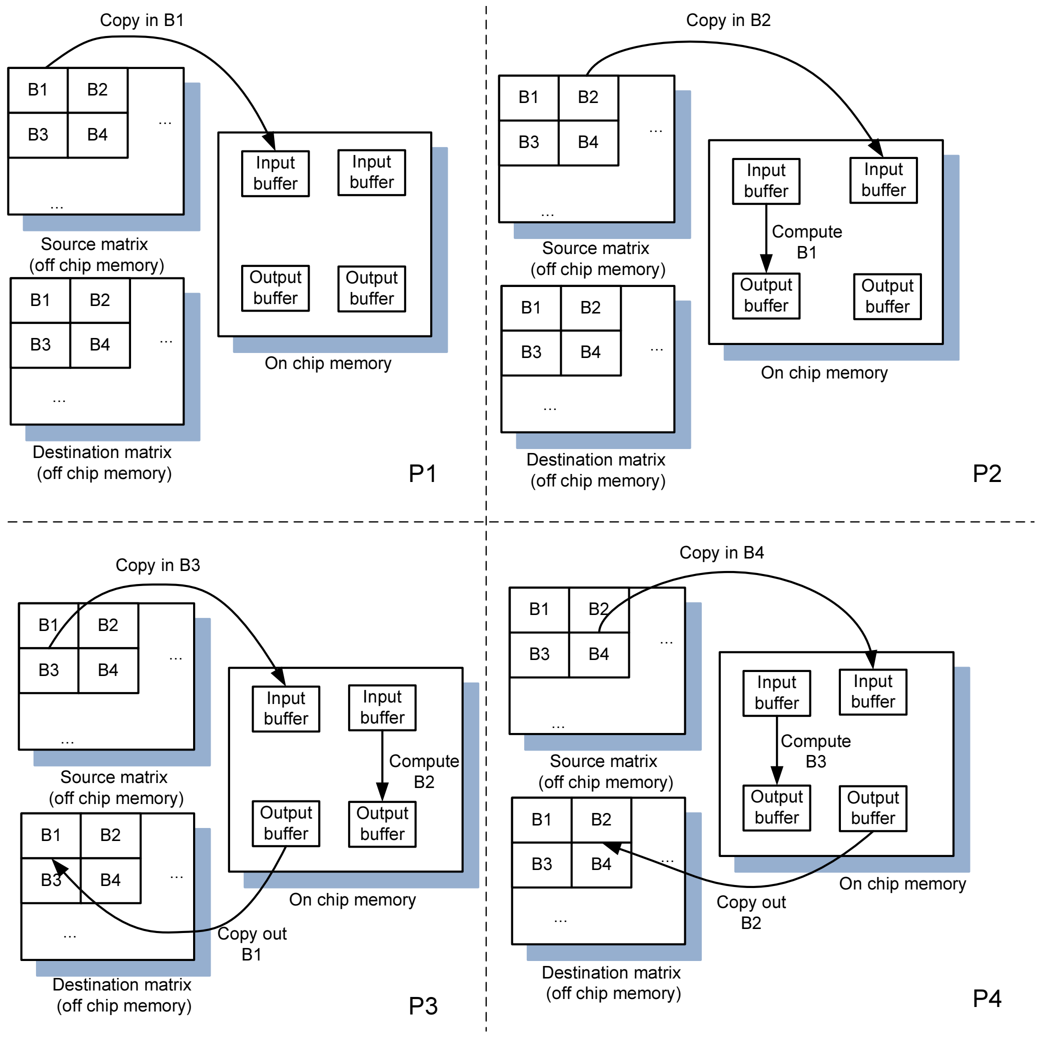


Figure 4.12: Overlapping Computation and Memory Transfer by Double Buffering

loops that has balanced computation to memory ratio. An illustration of double buffer rotation is shown in Figure 4.12.

An more detailed example of double buffer rotation method is shown in the pipeline model shown in Figure 4.13.

The Algorithm 4 illustrates the double buffer rotation algorithm. The input matrix I and output matrix O are tiled, each tile is associated with a tile label, for example Tile(I, 0) is the first tile of the input matrix I. Before the for loop, there are several operations that are used to fill in the double buffer rotation pipeline. The for loop is the double buffer rotation kernel. Each iteration of the for loop parallelizes

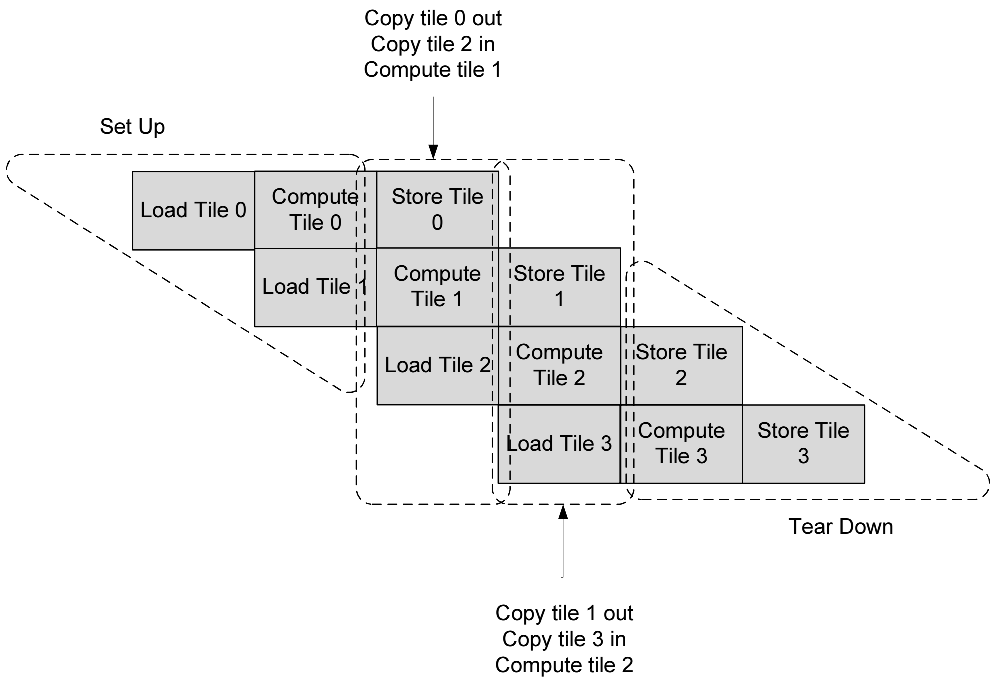


Figure 4.13: Illustration of the Double Buffer Rotation in Pipeline Diagram

the stencil on one tile, moving in and out the data for the another tile. The code after the loop is to finish the data remaining in the pipeline.

Tiling Geometry Optimization

We introduce the tiling and double buffer rotation strategy in the previous sections. The loop tiling is parameterized by the tiling size in x and y direction. In this section, we perform mathematical analysis to derive the equation for the optimal tiling geometry parameterization.

Our goal is to to find out the tiling size in x and y direction that minimizes the total memory access time. The total memory access time can be computed from the total size of the memory transfer (GB) divided by the memory bandwidth (GB/s). A constraint we can use is that the size of the on-chip is an constant C. If the tiling size in x direction is t_x, tiling size in y direction is t_y, we have $t_x \times t_y = C$.

If the size of the input matrix is $x \times y$, the margin of the stencil loop is m_x and

Algorithm 4 Double Buffer Rotation Algorthm

```
1: Input: Input Matrix I, tile size tx, ty
2: Output: Output Matrix O
3: Initialize EDMA module
4: Allocate scratchpad buffer $A[2]$, $B[2]$ from the frame heap
5: Compute the number of tiles $N$
6: Issue EDMA transfer: Tile(I, 0) to $A[0]$, Tile(I, 1) to $A[1]$
7: Wait for EDMA transfer complete
8: Compute Stencil A[0] to B[0]
9: for i = 1; i < N - 1; i++ do
10:     Issue EDMA transfer: Tile(I, i+1) to $A[(i+1)\%2]$, $B[(i+1)\%2]$ to Tile(O, i-1)
11:     Compute Stencil $A[i\%2]$ to $B[i\%2]$
12:     Wait for EDMA transfer complete
13: end for
14: Compute Stencil $A[i\%2]$ to $B[i\%2]$
15: Issue EDMA transfer: $B[(i+1)\%2]$ to Tile(O, N-2), $B[i\%2]$ to Tile(O, N-1)
16: Wait for EDMA transfer complete
```

m_y.

The area of each tile can be computed by the following equation

$$S_t = (t_x + 2m_x)(t_y + 2m_y) \tag{4.1}$$

The number of the tiles N_t can be computed by the following equation.

$$N_t = \frac{(x - 2m_x)(y - 2m_y)}{t_x t_y} \tag{4.2}$$

The total size of the memory transfer is $S_t \times N_t$.

Increasing the tiling in y direction lowers down the memory transfer speed (DMA memory transfer is more preferred to continuous memory transfer).

In Figure 4.14, we show when the total transfer size is fixed, the relationship between number of rows and the DMA bandwidth.

We use linear regression to fit the memory bandwidth w.r.t. the tiling size in y direction in Equation 4.3.

$$MB(t_y) = 8 - 0.08t_y \tag{4.3}$$

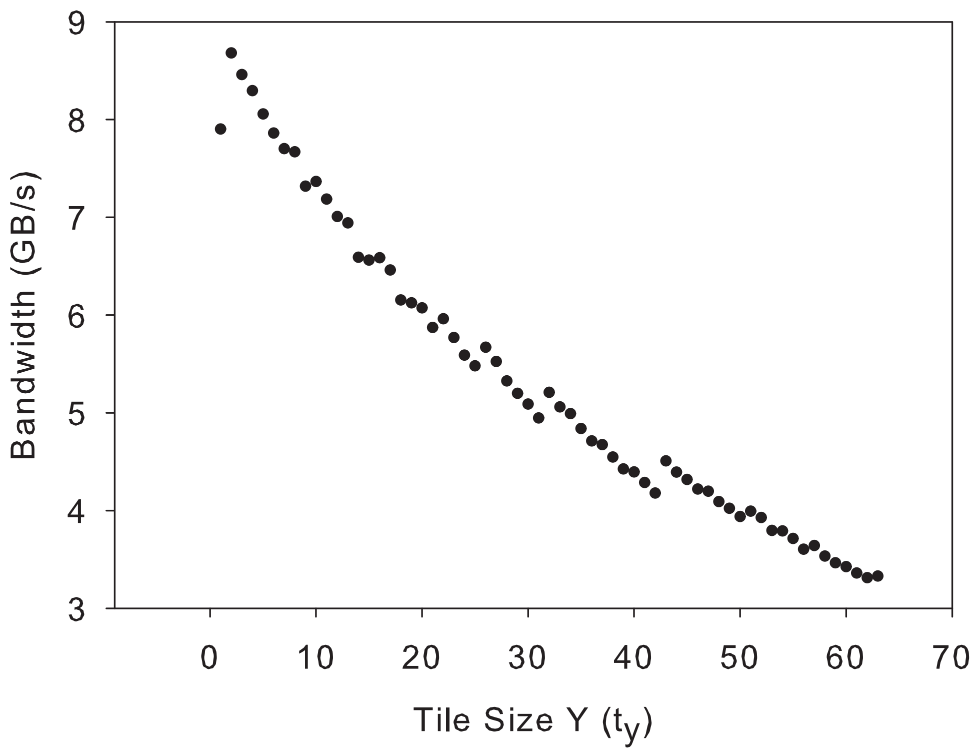


Figure 4.14: Relationship between EDMA bandwidth and the Tiling Size in y Direction (stride number)

Equation 4.4 gives the objective function that minimizes the tiling size t_y.

$$min\{F(t_y) = S_t N_t / MB(t_y)\} \tag{4.4}$$

Because $t_x = C/t_y$, the best tiling size in x and y direction can be decided by solving the optimization problem given in Equation 4.4. This would require solving $F'(t_y) = 0$.

The Figure 4.15 shows the total memory access time of tiling an 1024×1024 matrix. The curve shows that the best tiling parameter is located between 10 and 20.

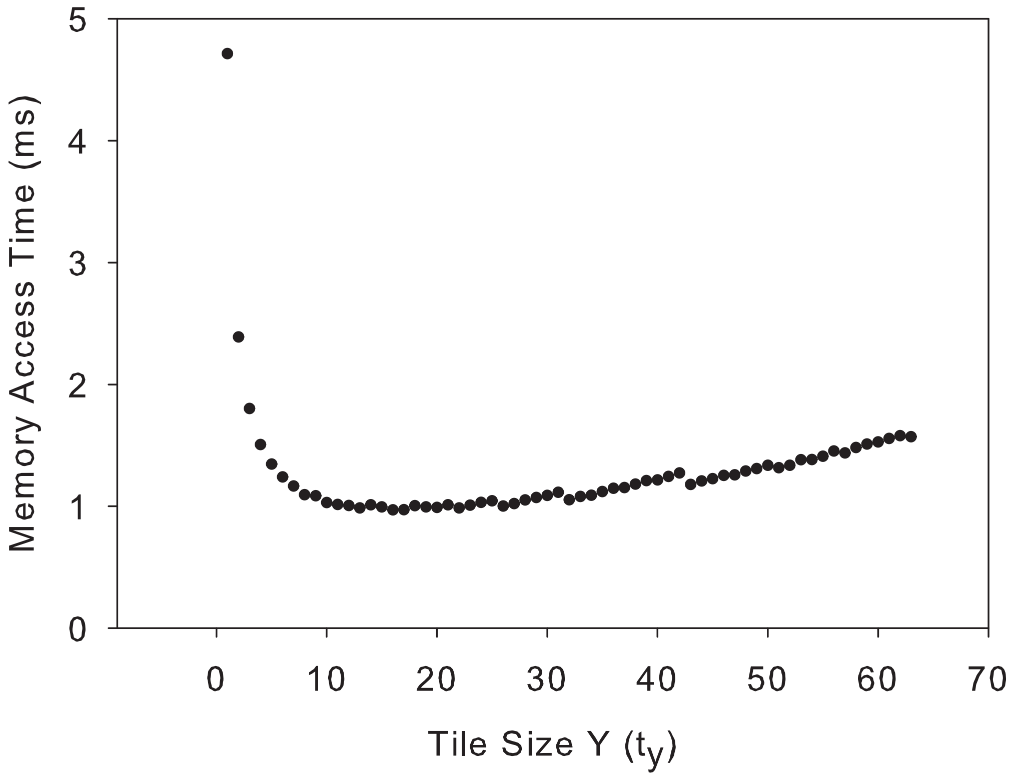


Figure 4.15: Relationship Between Memory Access Time and the Tiling Size in y Direction on an 1024x1024 Matrix

Chapter 5

Results and Analysis

In this chapter we use the selected stencil benchmarks from Chapter 4 to test and demonstrate the efficiency stencil code optimization and the tiling methodolog.

5.1 Performance of Loop Unrolling

We use the benchmark stencils introduced in Chapter 2 to test the performance improvement of loop unrolling. Among all the stencils, we do not use the Lucas Kanade and Harris Corner kernel because they have a relatively larger loop body that unrolling on them exceeds the maximum number of instructions that software pipeline can handle. However, we will show in the next section that they are able to be optimized by SIMD binding.

Figure 5.1 shows the memory operation bounds and the arithmetic operation bound on different kernel functions w.r.t. unroll factor. The unroll factor without data point means the loop is disqualified from software pipeline either because number of instructions exceeds 250, or the assembler fails to allocate enough registers to the loop. On the kernels with a relatively high memory to compute ratio, the memory operation bound becomes the performance bottleneck. The figure shows that the PIA loop unrolling optimizer is effective for improving the performance of the stencil kernel with high memory to compute ratio. The Gaussian kernel have lower MC ratio so that the loop unrolling does not reduce its memory operation bound significantly.

As we have introduced in Chapter 4, Iteration Interval, or II, measures the performance of the loop directly. The less II is, the faster the code is. We list the II

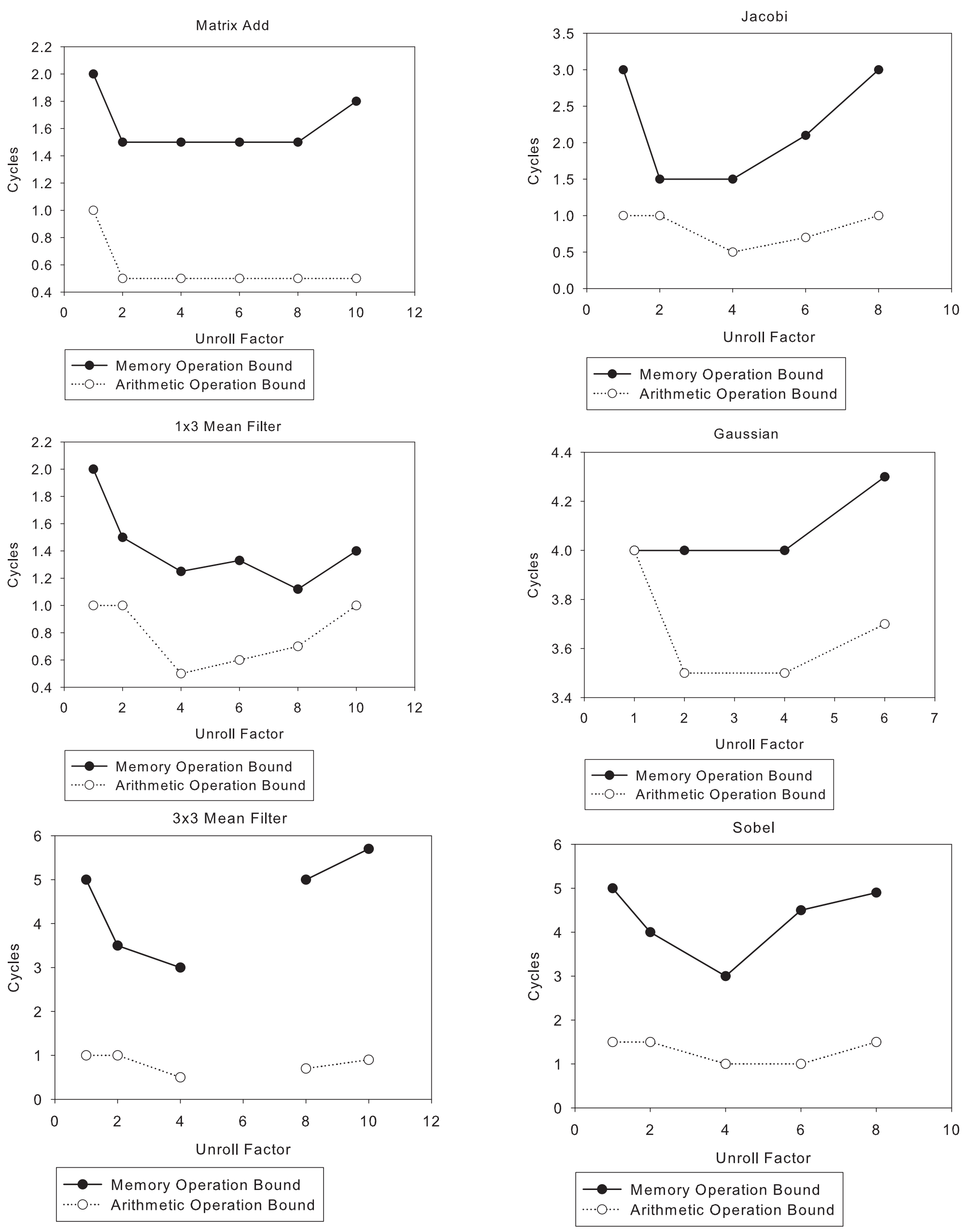


Figure 5.1: Memory and Arithmetic Operation Bounds Improvement from Loop Unrolling

improvement from loop unrolling on each stencil kernels in Figure 5.2. The unroll factor without data point means the loop is disqualified from software pipeline either because number of instructions exceeds 250, or the assembler fails to allocate enough registers to the loop. When the loop fails to be software pipelined, the II are considered to be the length of the loop in CPU cycles.

The loop unrolling performs better optimization on the kernels with higher memory to compute ratio. One reason for that is the TI's assembler backend has a limited ability to group up consecutive memory loads and stores automatically. This could improve the memory access bandwidth. The other reason for that is the kernels with less number of arithmetic operation per iteration usually do not fully utilized the multiple arithmetic functional units on the processor. Unrolling allows the compiler to group up those arithmetic operations into parallel VLIW instructions and filled in the functional unit with a higher occupation rate.

The two major reasons that the performance of loop unrolling is limited are listed below.

- Register Allocation: As the unroll factor increases, the loop body requires more registers. It may fail register allocation when the assemler attempts to software pipeline the loop reduce instruction parallelism.

- Addressing Operations: The unrolled loop needs more addressing operations, which are generated from the assembler as integer add instructions. The addressing operations occupy computational resources. As the unroll factor increases to a certain degree, they will become a factor that affects performance. In the worst case, it could cause software pipeline failure by exceeding the 250 instruction limitation or failing register allocation.

For example, the Figure 5.3 shows the relationship of register number and the loop unroll factor in all the kernel benchmarks. The unroll factor without data point

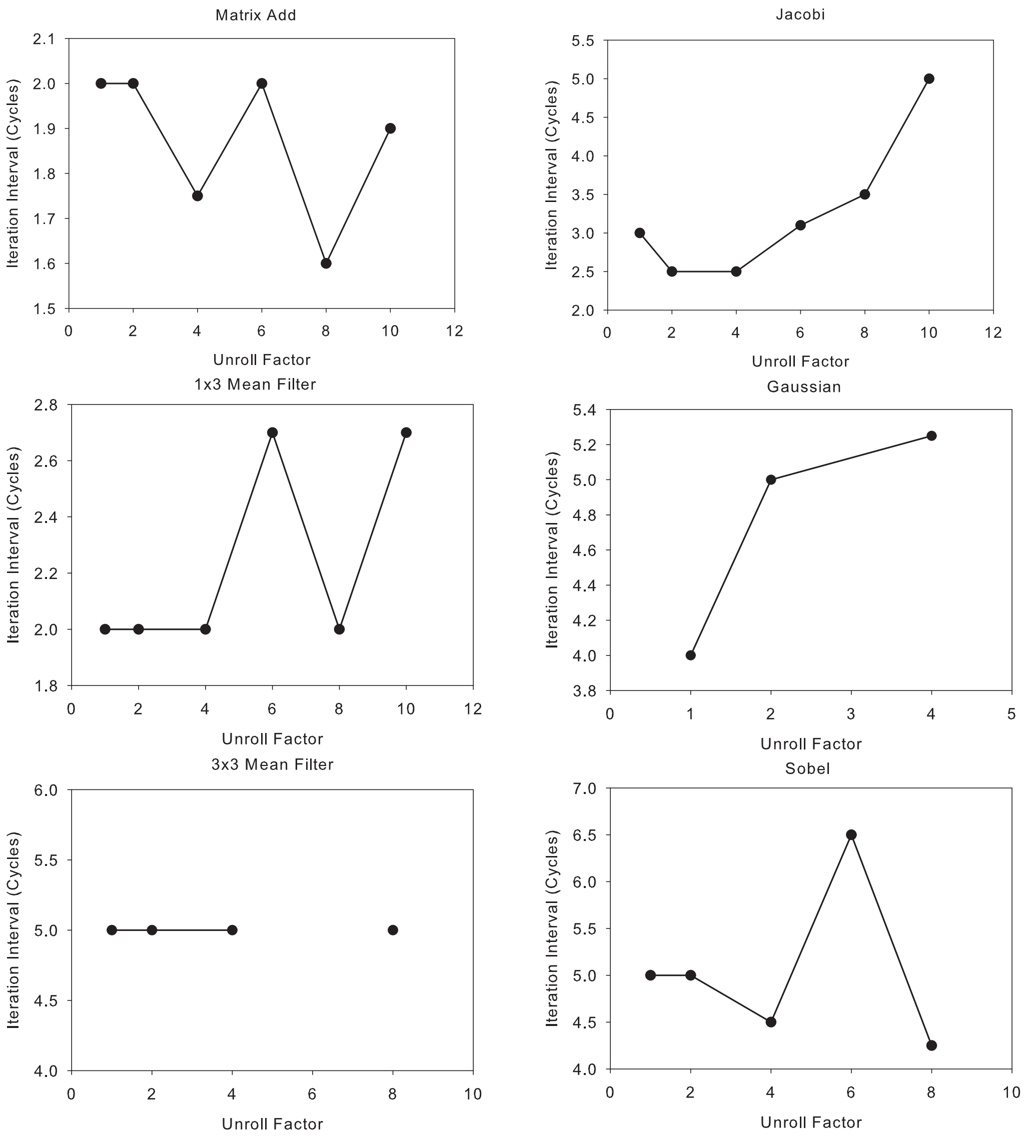


Figure 5.2: Improvement of Iteration Interval from Loop Unrolling

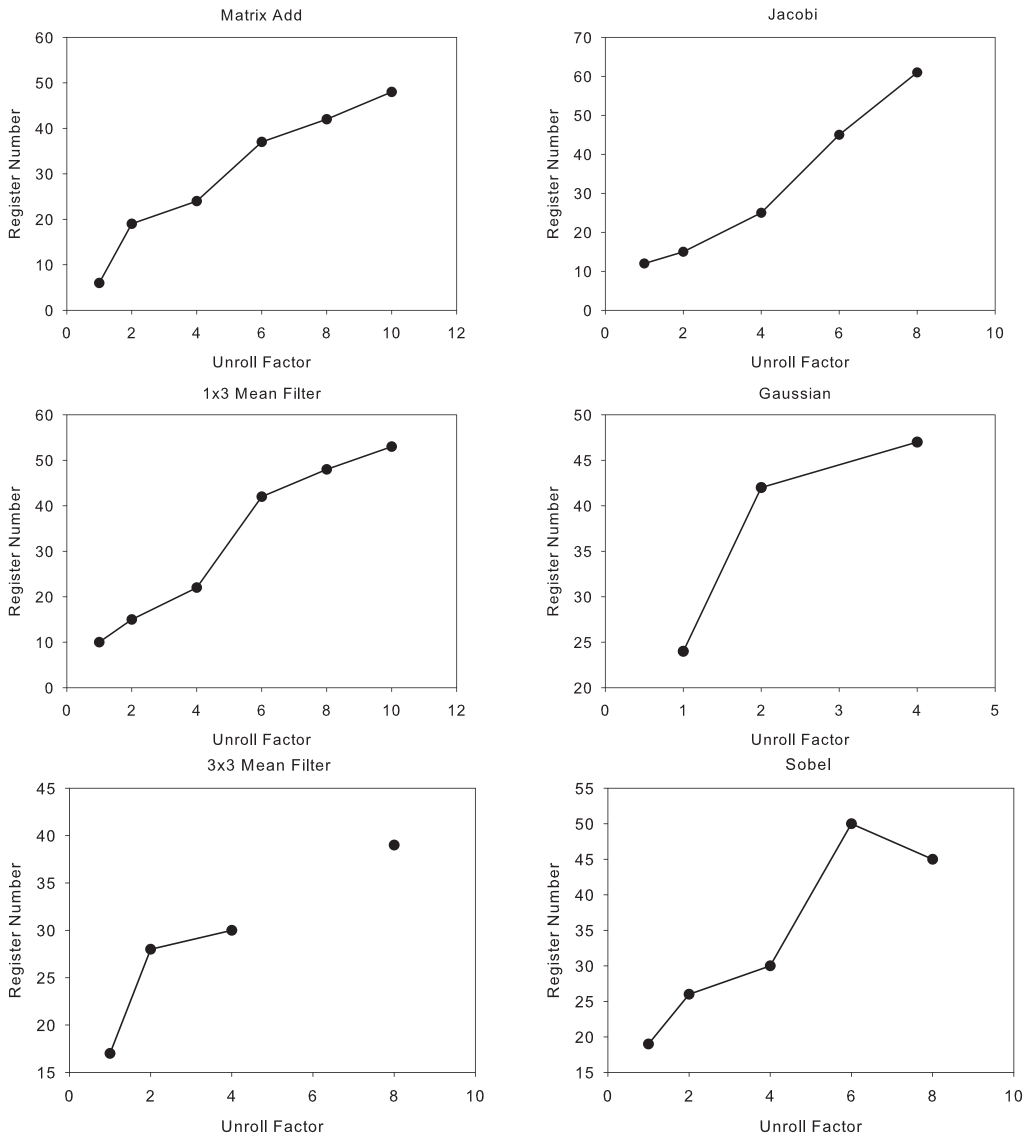


Figure 5.3: Register Usage from Loop Unrolling

means that the loop is disqualified from software pipeline either because number of instructions exceeds 250, or the assembler fails to allocate enough registers.

When the allocated register number reaches the maximum of the processor, the assembler fails to manage the loop in software pipeline. For example from the Figure 5.2 we observe that for the Gaussian kernel the unroll factor stops at 4. in the Figure 5.3 we can see that for the Gaussian kernel, unroll factor 4 uses 47 registers. That is the maximum register number that allows software pipeline on Gaussian kernel.

The Figure 5.4 shows the number of VLIW instructions that contain addressing operations for all the kernels. Instead of counting the total number of add instructions, we only count the VLIW instructions, or cycles that contain an interger add. The results show that the loop unrolling at 6x or 10x generates more addressing instructions, which explains why the II at 6x or 10x has a peak value. Additionally, we observe that the addressing instructions could disqualify software pipeline by adding register usage. An example is shown in 3x3 mean filter, unrolled at 6x.

Generally, the assembler generates better scheduled software pipeline code with an unroll factor equals to a power of 2. (1, 2, 4 or 8)

5.2 Performance of SIMD Binding

For those kernels with lower memory to compute ratio which loop unrolling does not show a significant improvement on II. We show in this section that our code optimizer is able to improve their performance with SIMD binding. By combining instructions into SIMD, ideally that the functional unit utilization could doubled.

Figure 5.5 shows the memory operation and arithmetic operation bound of the benchmark kernels with SIMD binding. From the figure we can see that the SIMD binding is able to reduce the floating point arithmetic operation bound for kernels with low memory to compute ratio significantly.

Figure 5.6 compares the results of performance improvement of iteration interval between SIMD binding and the baseline implementation. The results shows that the SIMD binding is able to further achieve an average 20% improvement the performance on the matrix add and Jacobi kernel. More importantly, The SIMD binding is able to achieve a significant amout of improvement on the large and complex stencils, in our experiments, these benchmarks are Harris Corner Method and Lucas Kanade Method. The SIMD binding is able to reduce the II on these two kernels by 50%.

The operations per iteration (OPI) reflects the computational length of the loop,

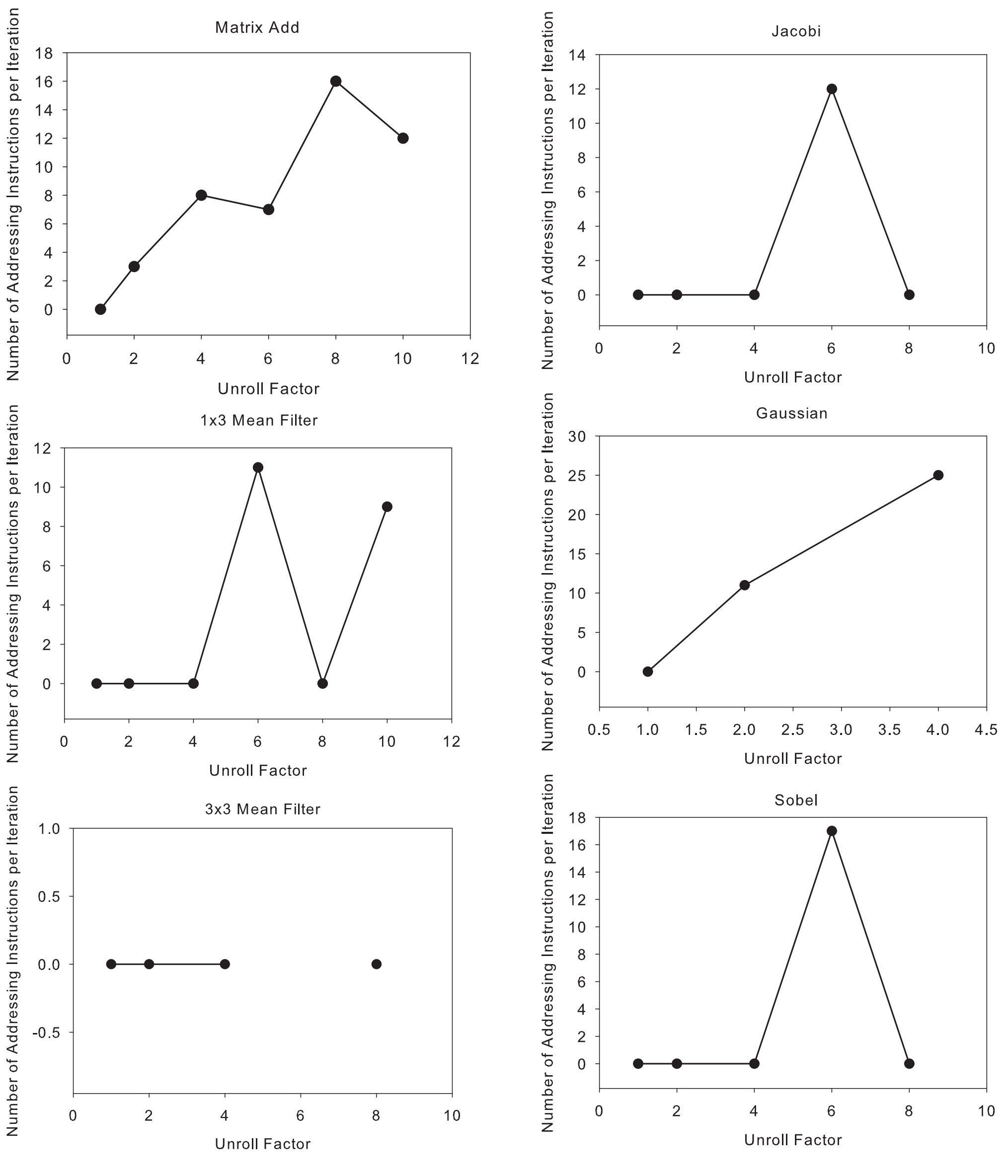


Figure 5.4: Addressing Instructions per Iteration from Loop Unrolling

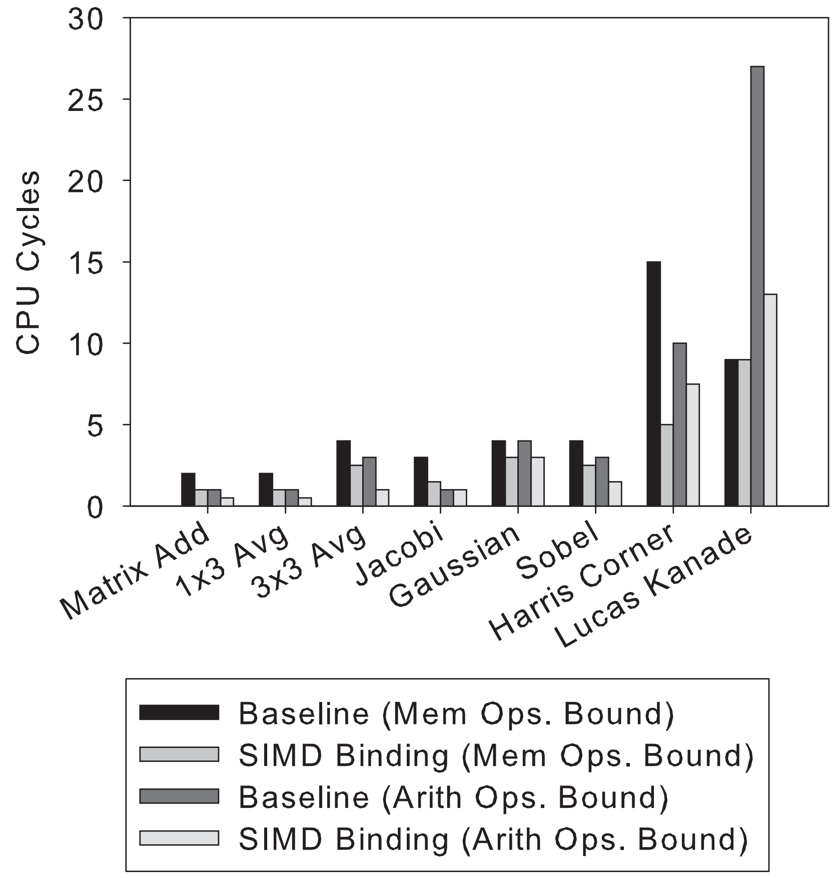


Figure 5.5: Memory and Arithmetic Operation Bound Improvement from SIMD Binding

which is mostly determined by the characteristic of the stencil and the efficiency of the code generation. The operations per cycle (OPC) reflects the execution throughput of the code. As the instructions are combined into SIMD, the OPC increases. However since the SIMD binding also requires addition data movement operations to be inserted, for example, in order to make a SIMD operand, values from different sides of the register files must be move into the same side. These extra data movement operations could potentially increase OPI. We perform an analysis on OPI and OPC between baseline and SIMD binding and show the efficiency of our SIMD binding optimization.

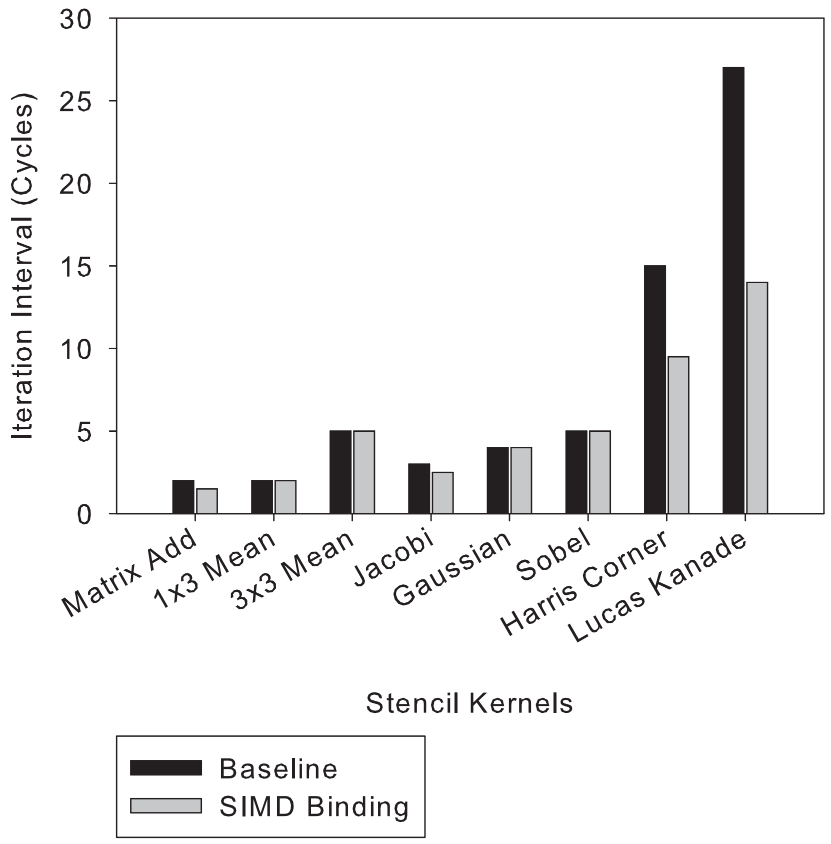


Figure 5.6: Compare the Iteration Interval between Baseline and SIMD Binding

Figure 5.7 shows the OPI results of the baseline code and SIMD optimized code. The length of the bar shows the complexity of the stencil kernel. As we have seen, the Harris Corner and Lucas Kanade method are the most expensive stencil loops among the test benchmarks. The difference of OPI between baseline and SIMD version is the traffic instruction that was caused from the data traffic overhead of the SIMD code generation. From the figure we can see that the generated traffic instructions are less than 10% of the total instruction.

Figure 5.8 shows the OPC value of the baseline code and SIMD optimized code. The IPC characterizes the throughput of the instruction execution. This figure shows that the SIMD binding is able to improve the instruction throughput of stencil loops in a ratio between 0% to 45% dependent on the complexity of the kernel.

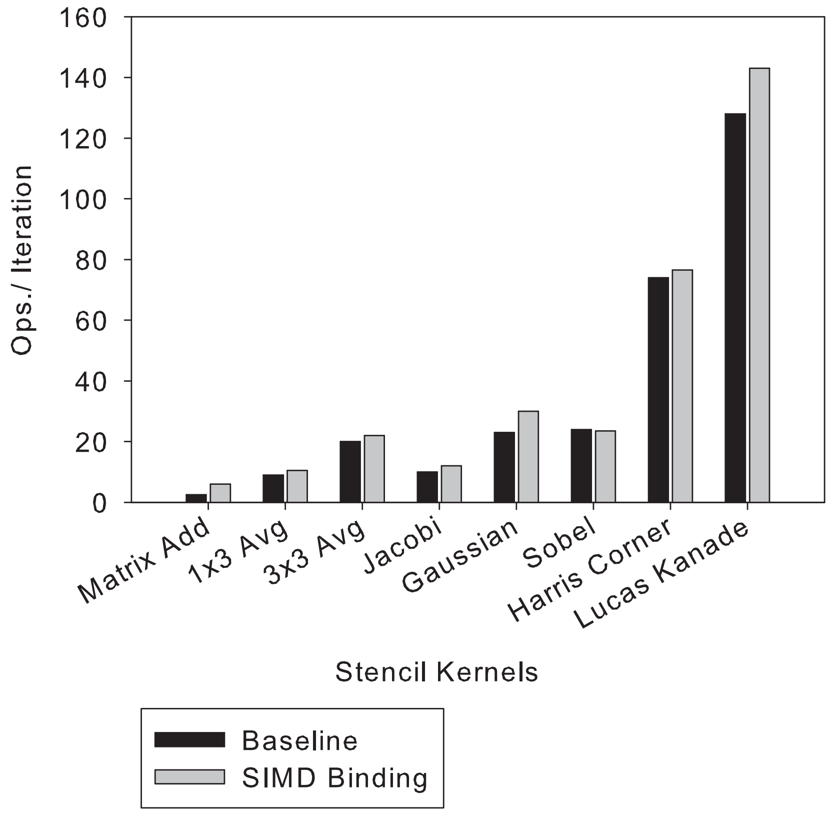


Figure 5.7: Compare the OPI between Baseline and SIMD Binding

Figure 5.9 shows the register usage of the SIMD optimized code. Compared with the baseline version, SIMD binding increases the register usage, but in a ratio less than 2.

As a conclusion of the experimental results shown above, we summarize the best optimization parameter that is found by PIA compiler and compare the best optimized II with the baseline in Table 5.1.

5.3 Performance of Compile-Time Alignment Detection

When additional information of the input matrices are provided from the user, our code optimization tool is able to use these informance to perform more aggressive optimization. If the input matrices are assumed to be row aligned with 8 bytes,

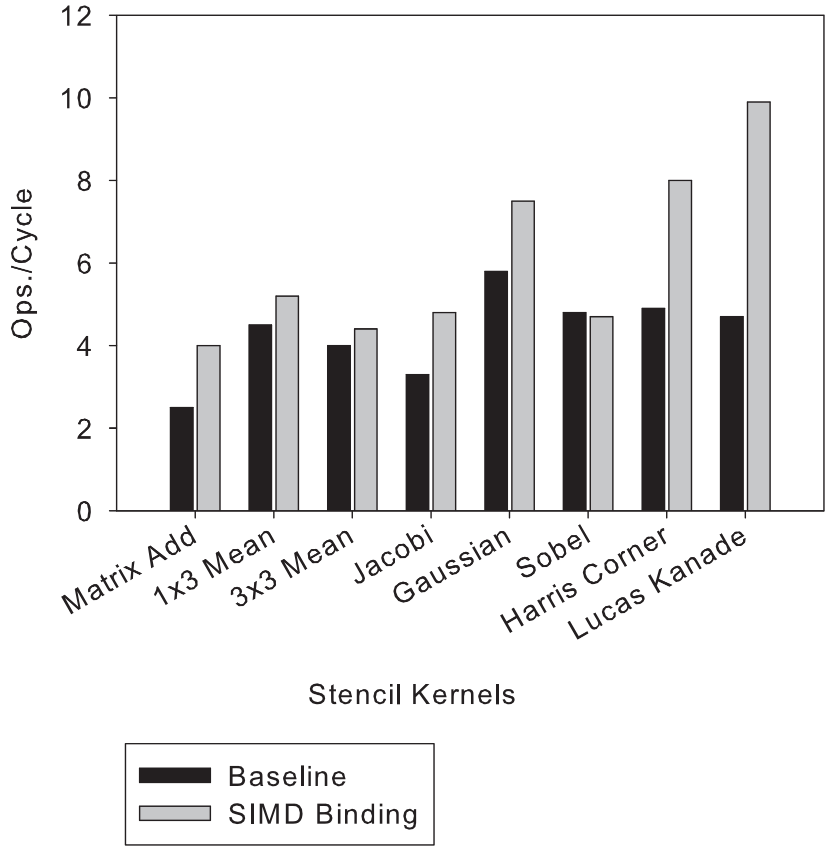


Figure 5.8: Compare the OPC between Baseline and SIMD Binding

Table 5.1: The Best Optimization Strategy for all the Test Benchmarks

	Baseline II	Best Optimized II	Best Optimization Strategy
Matrix Add	2	1.5	Unroll 2x + SIMD Binding
1x3 Mean Filter	2	2	Baseline
3x3 Mean Filter	5	4.5	Unroll 6x
Jacobi	3	2.5	Unroll 2x or Unroll 4x
Gaussian	4	4	Baseline
Sobel	5	4.25	Unroll 8x
Harris Corner	27	14	Unroll 2x + SIMD Binding
Lucas Kanade	15	9.5	Unroll 2x + SIMD Binding

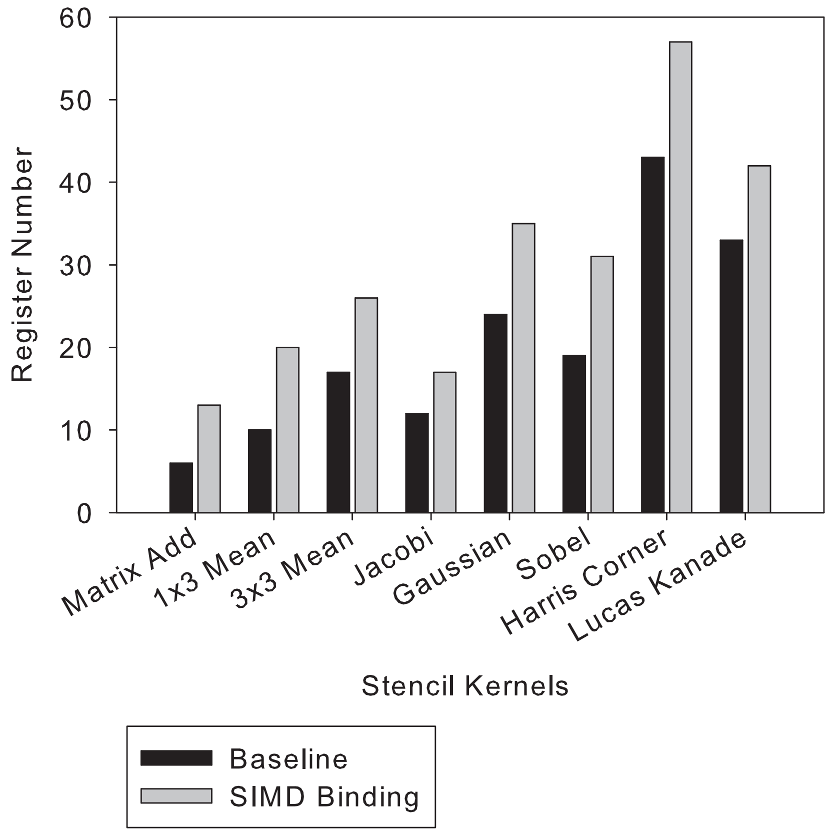


Figure 5.9: Compare the Register Usage between Baseline and SIMD Binding

the alignment detecton method is able to recognize and differentiate the aligned and unaligned SIMD memory instructions and reduce II from improveing the memory access throughput.

We show the performance of the code optimization after adding row align information into the PIA code in Figure 5.10. The results show that the alignment detection greatly improves the performance of the generated assembly code. The memory bound kernels such as Matrix Add, Mean Filter and Sobel Filter achieve the best performance improvement among all the bechmarks – the II is reduced by 30%. An average of 16.2% performance improvement is observed on all the test becnmarks.

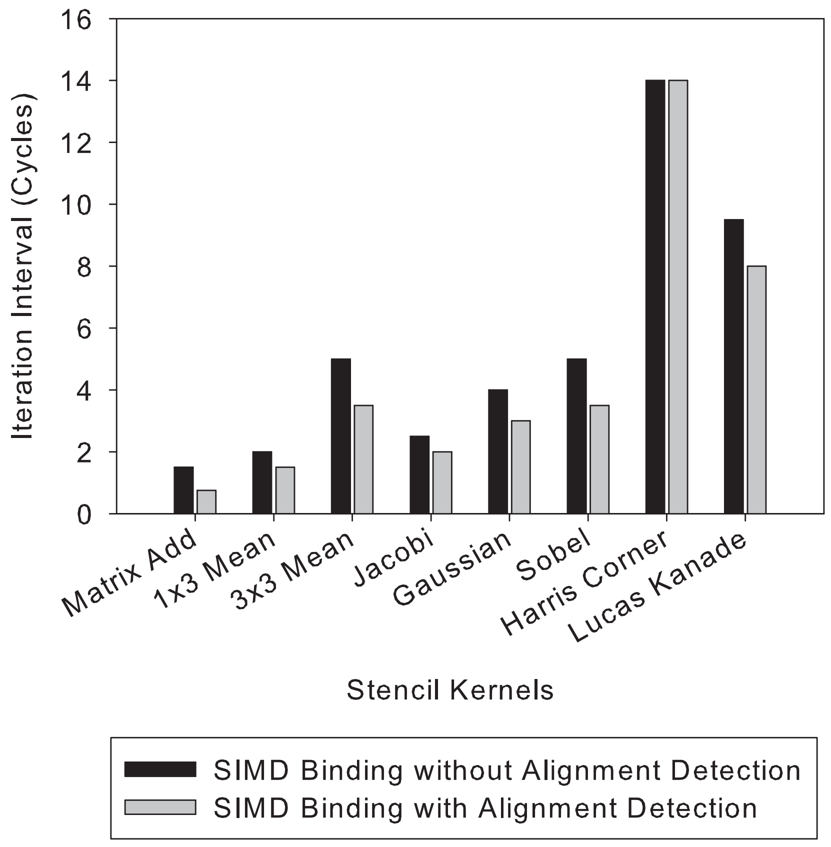


Figure 5.10: Performance Improvement from Compile-Time Alignment Detection

5.4 Performance of the Double Buffer Rotation Management

In this section, we show the experimental results of our stencil loop tiling and memory management methodology. In order to show the performance, We choose 4 kernel functions, Gaussian, Mean Filter, Lucas Kanade Method and Harris Corner Method. We use the halo region as the experimental variable to show the how the cache performance are affected by the size of the kernel matrix of the stencil.

We compare the performance between the baseline implementation (cache version) and the two versions of local buffer implementations. The baseline version is implemented with no loop tiling and buffering. The two versions of locally buffered implementations are singlely scheduled buffering and double buffer rotation, as introduced in the previous chapter.

The performance metrics we used for evaluating the memory management methodologies is Gflops, which is the giga number of floating point operation per second. The Gflops characterize the speed of the code directly.

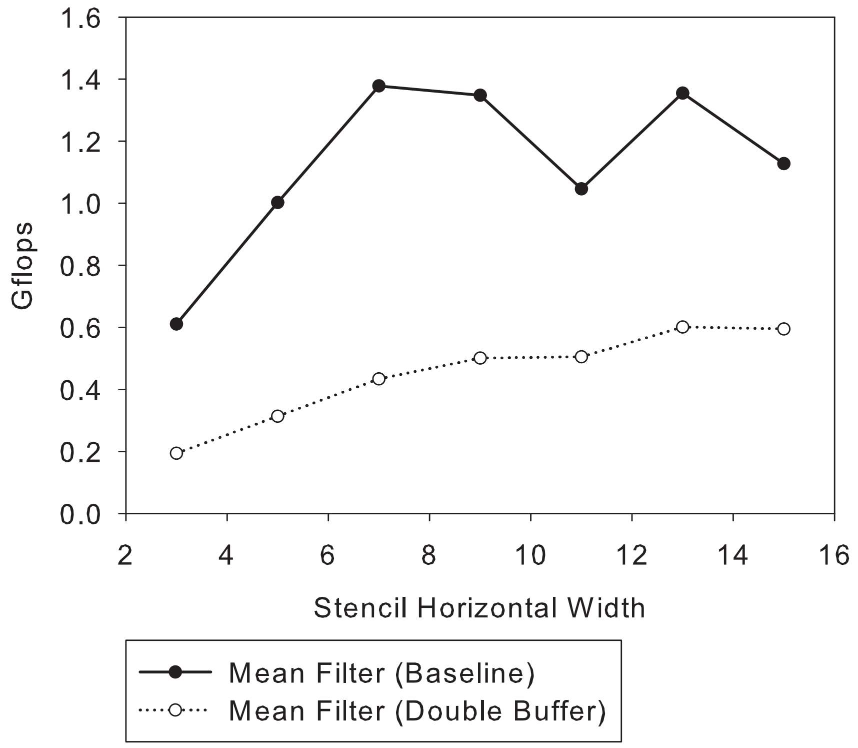
In order to show how efficient the double buffer rotation hides memory transfer latency. We collect the running time of memory transfer from single buffer and double buffer implemented loop tiling, which is represented as T_{mem_single} and T_{mem_double}. The double buffer efficiency is computer by the following equation.

$$E_{db} = \frac{(T_{mem\_single} - T_{mem\_double})}{T_{mem\_single}} \tag{5.1}$$

The Figure 5.11 and the Figure 5.12 shows the performance of double buffer rotation on Mean Filter and Gaussian Filter. The detailed description of these two filters can be found in Appendix A. In the figure, we can see that the double buffer rotation improves the performance of the stencil computation significantly when the vertical width of the filter increases. As the filter expends more rows. The cache miss rate is increasing. Using loop tiling and local scratchpad buffering could eliminates those overhead from cache misses.

The Figure 5.13 shows the performance results from two of the more complex stencil loops: Harris Corner Detector and Lucas Kanade Method. The results show that the double buffer rotation is able the improve the system performance of these complexstencil loops in a order of magnitude — up to 15x speedup for Lucas Kanade Method and 10x speedup for Harris Corner Detector.

The Figure 5.14, 5.15 and 5.16 show the DMA transfer time vs compute time on horizontal/vertical mean/Gaussian filters, Harris Corner detector and Lucas Kanade method. The x coordinate is the width of the stencil. The y coordinate is the spent time in millisecond. As the stecil width grows the percentage of time spent on computation increase, and the memory transfer time stays constant. This shows that



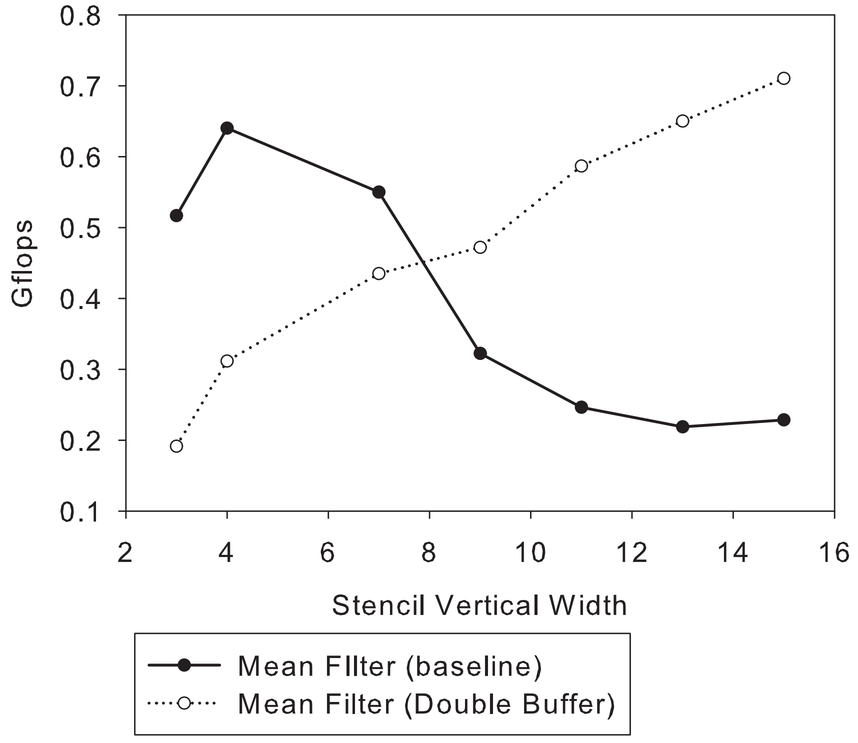
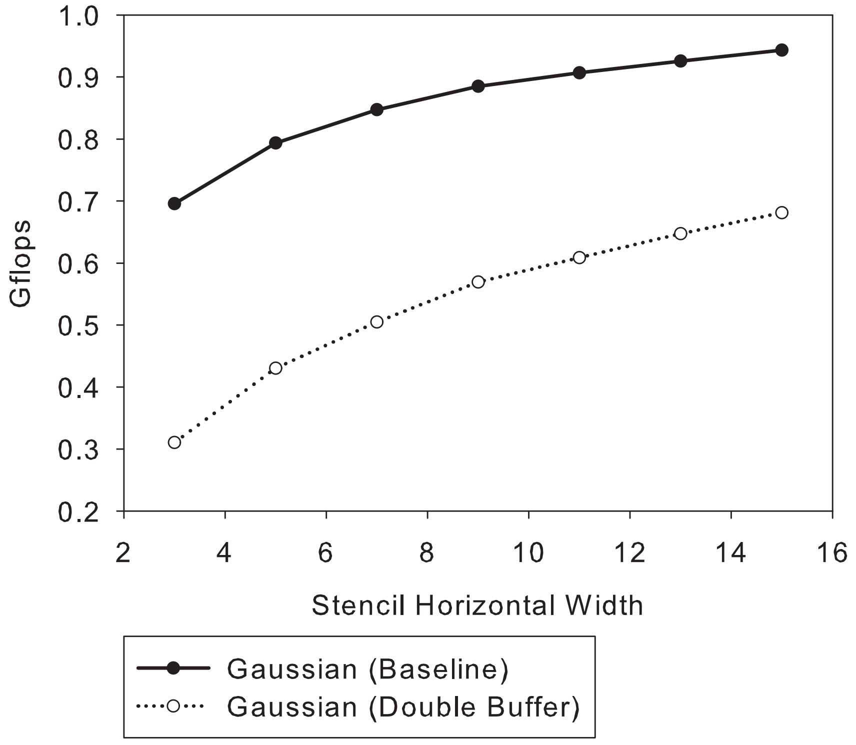


Figure 5.11: Performance of Double Buffer Rotation on Mean Filter



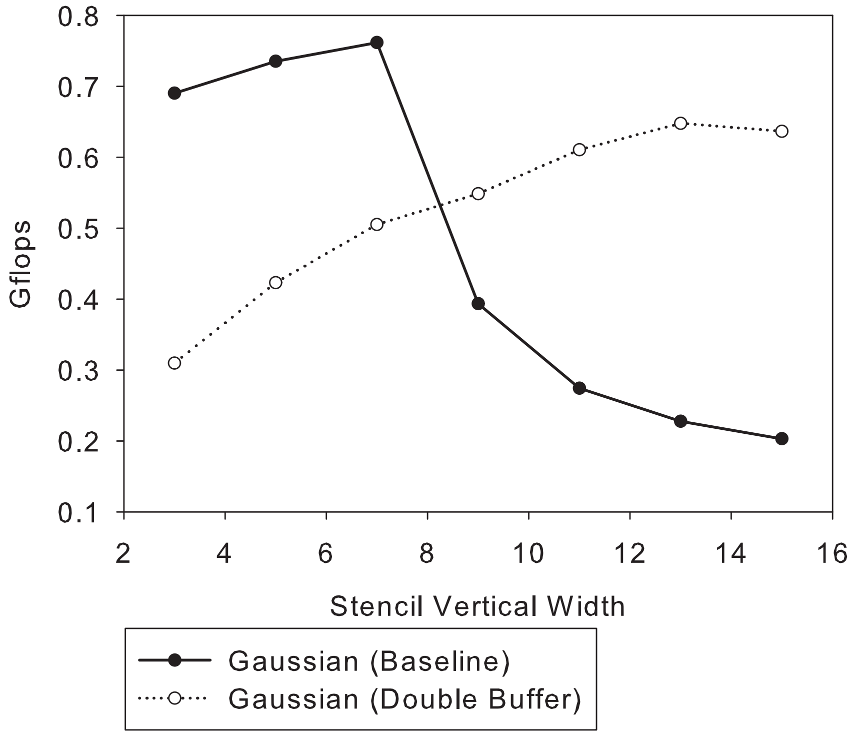
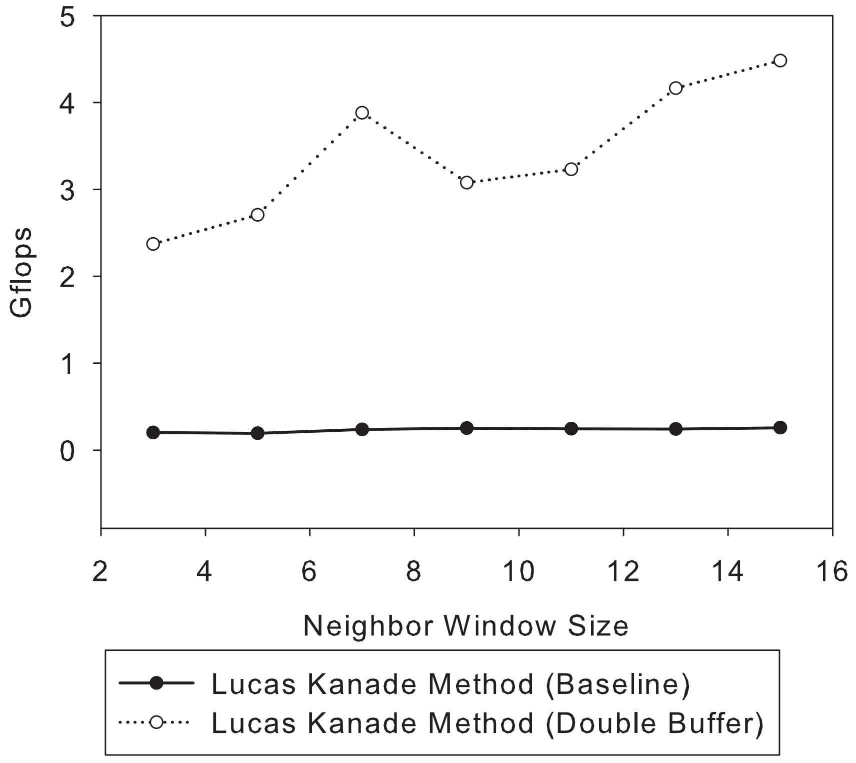


Figure 5.12: Performance of Double Buffer Rotation on Gaussian Filter



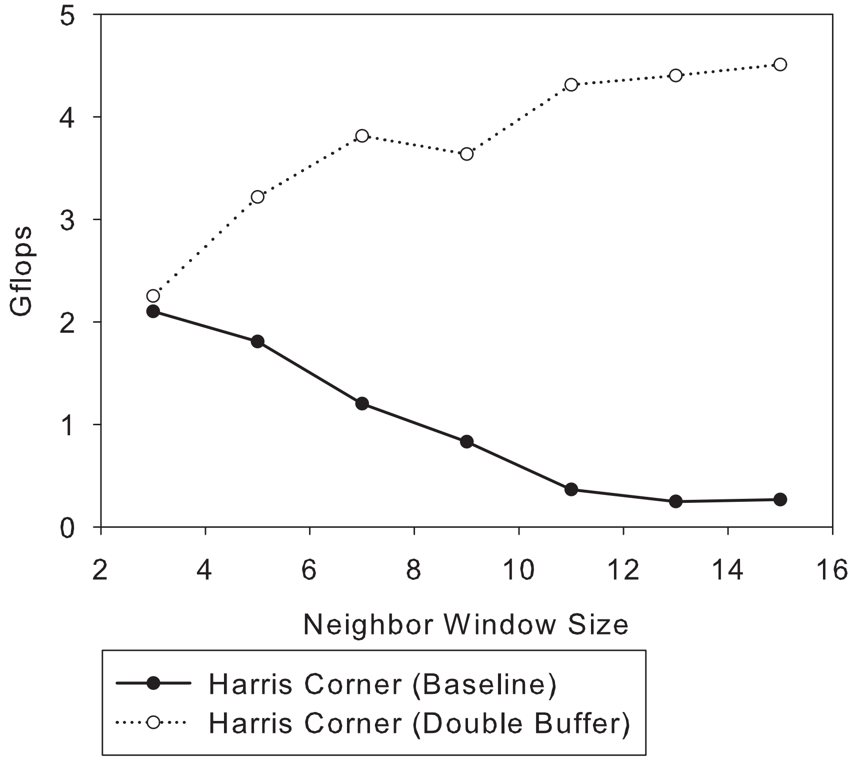


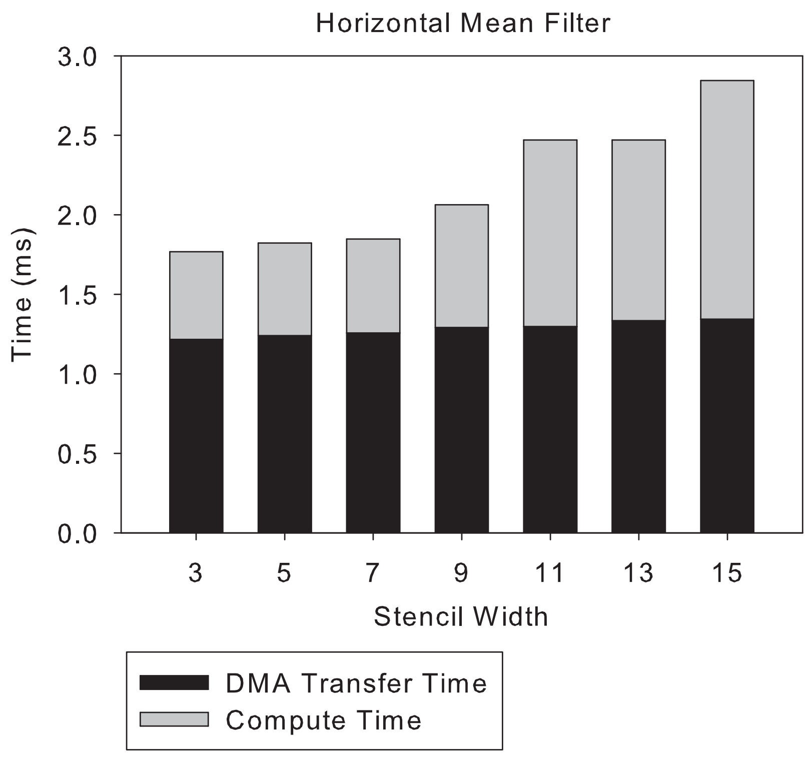
Figure 5.13: Performance of Double Buffer Rotation on Lucas Kanade Method and Harris Corner Detector

Table 5.2: The Improvement of Memory Access Time from Double Buffer Rotation

	3x	5x	7x	9x	11x	13x	15x
Mean Filter (H)	13.2%	13.1%	13.6%	15.0%	15.5%	16.3%	16.1%
Mean Filter (V)	13.2%	13.2%	13.5%	15.4%	14.6%	16.0%	16.2%
Gaussian (H)	15.0%	15.4%	15.7%	16.4%	15.9%	16.7%	16.3%
Gaussian (V)	15.0%	15.3%	16.1%	16.5%	16.5%	16.5%	16.8%
Lucas Kanade	16.1%	16.5%	16.5%	16.5%	16.8%	17.2%	17.3%
Harris Corner	15.4%	16.4%	16.2%	16.3%	16.4%	17.1%	17.0%

the memory transfer time of the double buffer tiling is not dependent to the data reuse pattern of the stencil.

The Table 5.2 shows the percentage of the improvement of the average memory access time after double buffer rotation is applied. These results are compared with the single buffer version. We list the improvement rate from horizontal and vertical Gaussian filter, horizontal and vertical mean filter, Harris Corner and Lucas Kanade kernels with different parameter setup.



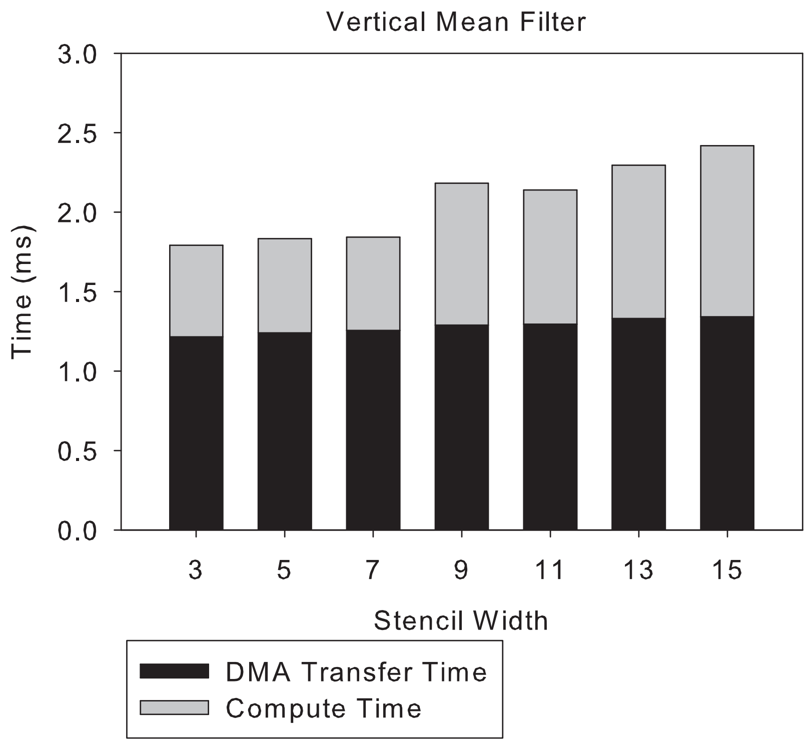


Figure 5.14: Time Split Analysis of Double Buffered Mean Filter

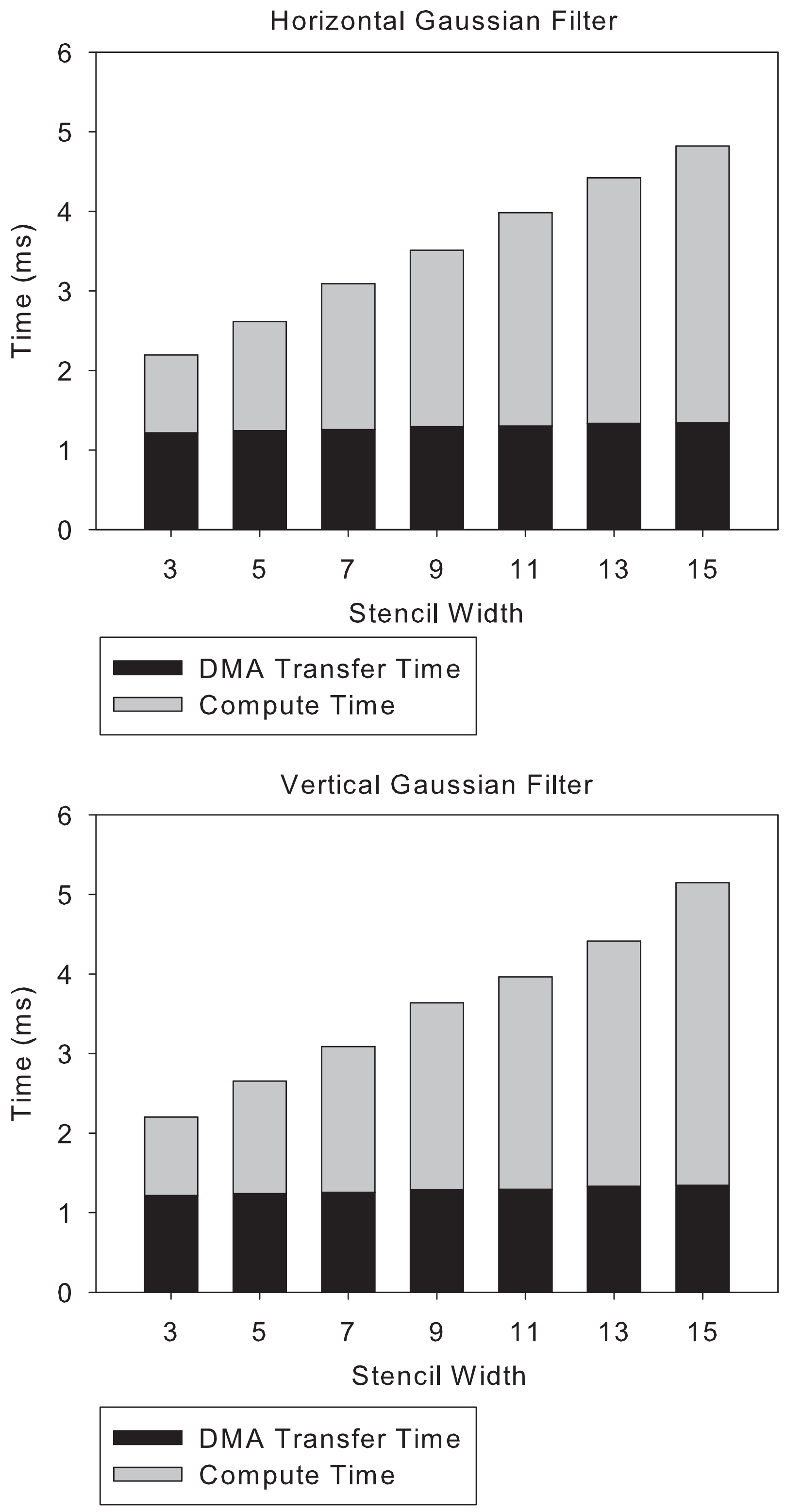
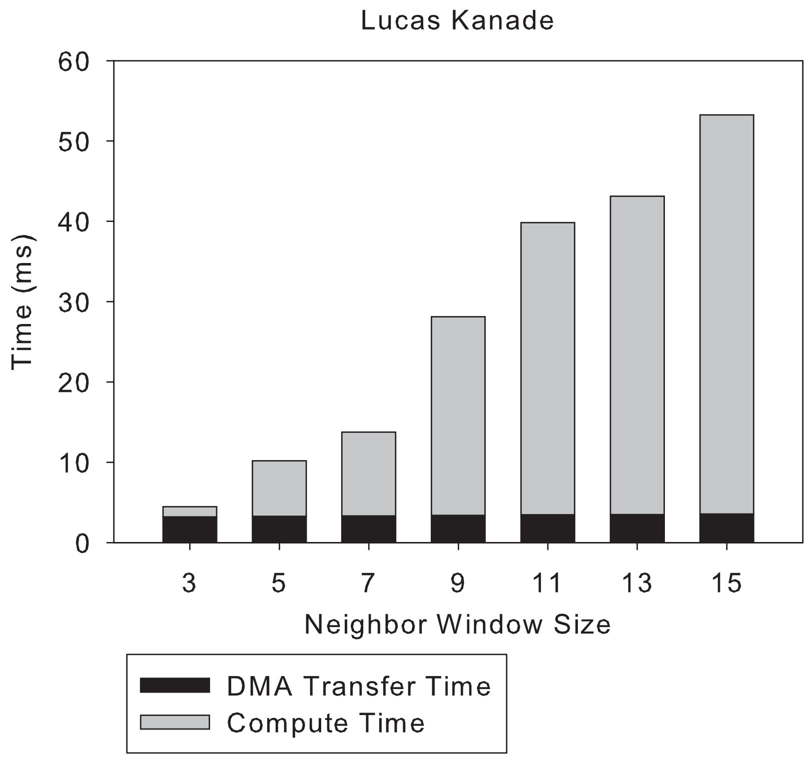


Figure 5.15: Time Split Analysis of Double Buffered Gaussian Filter



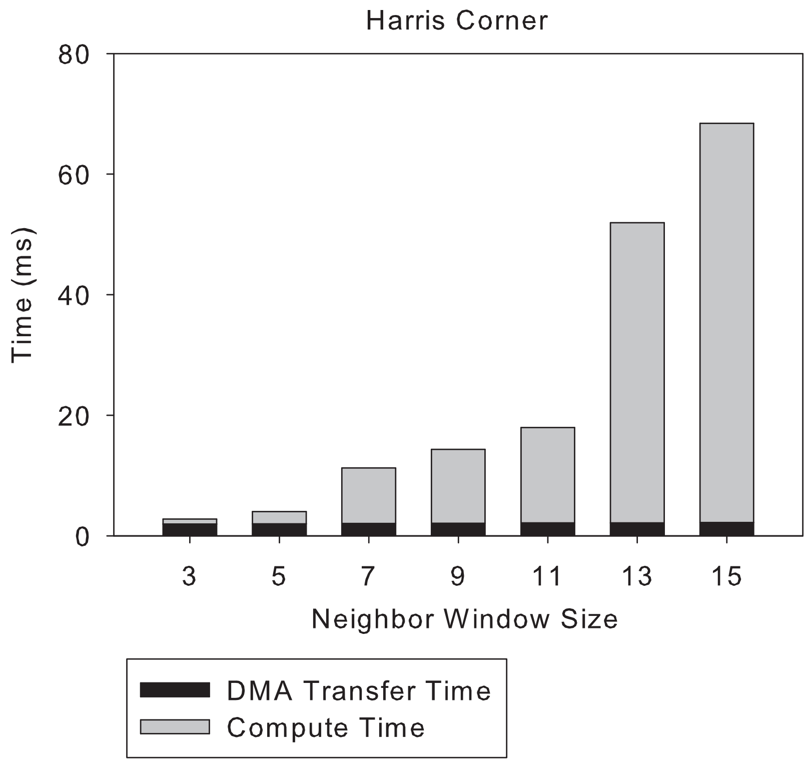


Figure 5.16: Time Split Analysis of Double Buffered Lucas Kanade Method and Harris Corner Detector

Chapter 6

Conclusion

In the first part of the dissertation, we describe a novel methodology for programming and optimizing of stencil loops. We target our design platform to be Texas Instrument C66x DSP. This design is able to automatically search the space of optimization strategy and find the best optimization parameter. It is shown in the results and analysis section that our designed code optimization is able to provide up tp 100% improvement in speed.

The speedup from code optimization depends on the computational characterization of the stencil loop. We discover that when the stencil loop is memory bound. Loop unrolling is more effective. When the stencil loop is compute bound, SIMD Binding is more effectiive. It raises the utilization rate of processor's functional units up to 2x. Both the loop unrolling and SIMD binding require more registers and may not be effective when the loop body is too large.

In the second part of the dissertation, we describe a memory manage methodology for stencil loop tiling. Our memory manage tool is able to automatically split the input and output matrices into proper blocks, and handles data copy in and out. The computation is performed entirely on the fast scratchpad memory. It is able to buffer data in two dimensions and reduce the memory access overhead from using cache. We also develop a double buffer rotation mechanism to hide the memory transfer latency from the hardware DMA.

The double buffer rotation is proven to be very effective on the stencils with a large halo region. Those stencils are frequently used in signal processing and computer vision, such as feature extraction and movement detection. Using double buffer rotation could achieve a speed up of 10x compared with cache. Compared with single buffer, double buffer rotation is able to reduce 17% of the average memory access time from the stencil computation.

Bibliography

[1] M. Anguita, J. Diaz, E. Ros, and F.J. Fernandez-Baldomero, *Optimization strategies for high-performance computing of optical-flow in general-purpose processors*, Circuits and Systems for Video Technology, IEEE Transactions on **19** (2009), no. 10, 1475–1488.

[2] Manjunath Basavaiah, *Development of optical flow based moving object detection and tracking system on an embedded dsp processor*, Journal of Advances in Computational Research: An International Journal **1** (2012), 15—25.

[3] Adrien Bernhardt, Andre Maximo, Luiz Velho, Houssam Hnaidi, and Marie-Paule Cani, *Real-time terrain modeling using cpu-gpu coupled computation*, Proceedings of the 2011 24th SIBGRAPI Conference on Graphics, Patterns and Images (Washington, DC, USA), SIBGRAPI '11, IEEE Computer Society, 2011, pp. 64–71.

[4] Ghulam H Bham and Rahim F Benekohal, *A high fidelity traffic simulation model based on cellular automata and car-following concepts*, Transportation Research Part C: Emerging Technologies **12** (2004), no. 1, 1–32.

[5] J. Chase, B. Nelson, J. Bodily, Zhaoyi Wei, and Dah-Jye Lee, *Real-time optical flow calculations on fpga and gpu architectures: A comparison study*, Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on, April 2008, pp. 173–182.

[6] M. Christen, O. Schenk, and H. Burkhart, *Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures*, Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International, May 2011, pp. 676–687.

[7] Florent Cohen, Philippe Decaudin, and Fabrice Neyret, *Gpu-based lighting and shadowing of complex natural scenes*, ACM SIGGRAPH 2004 Posters (New York, NY, USA), SIGGRAPH '04, ACM, 2004, pp. 91–.

[8] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken, *Logp:*

Towards a realistic model of parallel computation, SIGPLAN Not. **28** (1993), no. 7, 1–12.

[9] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, *Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures*, High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for, Nov 2008, pp. 1–12.

[10] N. Devi and V. Nagarajan, *Fpga based high performance optical flow computation using parallel architecture*, International Journal of Soft Computing and Engineering **2** (2012), no. 1, 433–437.

[11] Peng Di and Jingling Xue, *Model-driven tile size selection for doacross loops on gpus*, Proceedings of the 17th International Conference on Parallel Processing - Volume Part II (Berlin, Heidelberg), Euro-Par'11, Springer-Verlag, 2011, pp. 401–412.

[12] Wenbin Fang, Mian Lu, Xiangye Xiao, Bingsheng He, and Qiong Luo, *Frequent itemset mining on graphics processors*, Proceedings of the Fifth International Workshop on Data Management on New Hardware (New York, NY, USA), DaMoN '09, ACM, 2009, pp. 34–42.

[13] N. Ferrando, M.A. Gosalvez, J. CerdÃą, R. Gadea, and K. Sato, *Octree-based, {GPU} implementation of a continuous cellular automaton for the simulation of complex, evolving surfaces*, Computer Physics Communications **182** (2011), no. 3, 628 – 640.

[14] Nestor Ferrando and M. A. Gosalvezo, *Octree-based, gpu implementation of a continuous cellular automaton for the simulation of complex, evolving surfaces*, Computer Physics Communications (2011), 628–640.

[15] Yang Gao and J.D. Bakos, *Sparse matrix-vector multiply on the texas instruments c6678 digital signal processor*, Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on, June 2013, pp. 168–174.

[16] T. Gautama and M.M. Van Hulle, *A phase-based approach to the estimation of the optical flow field using spatial filtering*, Neural Networks, IEEE Transactions on **13** (2002), no. 5, 1127–1136.

[17] Dongni Han, Shixiong Xu, Li Chen, and Lei Huang, *Pads: A pattern-driven stencil compiler-based tool for reuse of optimizations on gpgpus*, Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on, Dec 2011, pp. 308–315.

[18] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan, *High-performance code generation for stencil computations on gpu architectures*, Proceedings of the 26th ACM International Conference on Supercomputing (New York, NY, USA), ICS '12, ACM, 2012, pp. 311–320.

[19] B. Hutchings, B. Nelson, S. West, and R. Curtis, *Optical flow on the ambric massively parallel processor array (mppa)*, Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on, April 2009, pp. 141–148.

[20] Texas Instruments, *The ti code generation tools*, `http://processors.wiki.ti.com/index.php/Production_Compiler_Releases`.

[21] ———, *Introduction to tms320c6000 dsp optimization*, 2014.

[22] ———, *Tms320c6678 multicore fixed and floating-point digital signal processor (rev. e)*, 2014.

[23] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, et al., *Kinectfusion: real-time 3d reconstruction and interaction using a moving depth camera*, Proceedings of the 24th annual ACM symposium on User interface software and technology, ACM, 2011, pp. 559–568.

[24] Díaz Javier and Ros Eduardo, *Superpipelined high-performance optical-flow computation architecture*, Comput. Vis. Image Underst. **112** (2008), no. 3, 262–273.

[25] Seunghun Jin, Dongkyun Kim, Dung Duc Nguyen, and Jae Wook Jeon, *Pipelined hardware architecture for high-speed optical flow estimation using fpga*, Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (Washington, DC, USA), FCCM '10, IEEE Computer Society, 2010, pp. 33–36.

[26] Zheming Jin and Jason Bakos, *Extending the beagle library to a multi-fpga platform*, BMC Bioinformatics, January 2013, pp. 14–25.

[27] ———, *A heuristic scheduler for port-constrained floating-point pipelines*, International Journal of Reconfigurable Computing, October 2013, pp. 41–52.

[28] ______, *Memory access scheduling on the convey hc-1*, Field Programmable Custom Computing Machines, 2013 IEEE 21st Annual International Symposium, April 2013, pp. 237–249.

[29] Mahmut Kandemir, J. Ramanujam, and A. Choudhary, *Exploiting shared scratch pad memory space in embedded multiprocessor systems*, Proceedings of the 39th Annual Design Automation Conference (New York, NY, USA), DAC '02, ACM, 2002, pp. 219–224.

[30] Yukihiro Komura and Yutaka Okabe, *Gpu-based single-cluster algorithm for the simulation of the ising model*, Journal of Computational Physics **231** (2012), no. 4, 1209–1215.

[31] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan, *Effective automatic parallelization of stencil computations*, SIGPLAN Not. **42** (2007), no. 6, 235–244.

[32] Zhiyuan Li and Yonghong Song, *Automatic tiling of iterative stencil loops*, ACM Trans. Program. Lang. Syst. **26** (2004), no. 6, 975–1028.

[33] Jose L. MartÃŋn and Aitzol Zuloaga, *Hardware implementation of optical flow constraint equation using fpgas*, Computer Vision and Image Understanding **98** (2005), no. 3, 462—490.

[34] Julien Marzat and Yann Dumortier, *Real-time dense and accurate parallel optical flow using cuda*, WSCG (2009), 105–111.

[35] Paulius Micikevicius, *3d finite difference computation on gpus using cuda*, GPGPU2 (2009), 79—84.

[36] J. Monson, M. Wirthlin, and B.L. Hutchings, *Implementing high-performance, low-power fpga-based optical flow accelerators in c*, Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on, June 2013, pp. 363–369.

[37] A. Nguyen, N. Satish, J. Chhugani, Changkyu Kim, and P. Dubey, *3.5-d blocking optimization for stencil computations on modern cpus and gpus*, High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for, Nov 2010, pp. 1–13.

[38] nvidia, *Nvidia cuda sdk*, `https://developer.nvidia.com/cuda-toolkit`, 2012.

[39] University of Illinois at Urbana-Champaign, *The llvm compiler infrastructure*, `llvm.org`.

[40] K. Pauwels, M. Tomasi, J. Diaz Alonso, E. Ros, and M.M. Van Hulle, *A comparison of fpga and gpu for real-time phase-based optical flow, stereo, and local image features*, Computers, IEEE Transactions on **61** (2012), no. 7, 999–1012.

[41] K. Pauwels and M.M. Van Hulle, *Realtime phase-based optical flow on the gpu*, Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on, June 2008, pp. 1–8.

[42] Gabriel Rivera and Chau-Wen Tseng, *A comparison of compiler tiling algorithms*, Proceedings of the 8th International Conference on Compiler Construction, Held As Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99 (London, UK, UK), CC '99, Springer-Verlag, 1999, pp. 168–182.

[43] H. Scharr, *Optimal operators in digital image processing*, (2000).

[44] G. D. Smith, Numerical Solution of Partial Differential Equations: Finite Difference Methods (1986), 247.

[45] J.C. Sosa, R. Gomez-Fabela, J.A. Boluda, and F. Pardo, *Change-driven image architecture on fpga with adaptive threshold for optical-flow computation*, Reconfigurable Computing and FPGA's, 2006. ReConFig 2006. IEEE International Conference on, Sept 2006, pp. 1–8.

[46] The GNU Operating System, `http://www.gnu.org/software/flex`.

[47] ______, `http://www.gnu.org/software/bison`.

[48] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson, *The pochoir stencil compiler*, Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures (New York, NY, USA), SPAA '11, ACM, 2011, pp. 117–128.

[49] M. Tomasi, M. Vanegas, F. Barranco, J. Diaz, and E. Ros, *High-performance optical-flow architecture based on a multi-scale, multi-orientation phase-based model*, Circuits and Systems for Video Technology, IEEE Transactions on **20** (2010), no. 12, 1797–1807.

[50] Zhaoyi Wei, Dah-Jye Lee, Dah-Jye Lee, Brent Nelson, and Michael Martineau, *A fast and accurate tensor-based optical flow algorithm implemented in fpga*, Applications of Computer Vision, IEEE Workshop on **0** (2007), 18.

[51] Fan Zhang, Yan Zhang, and J. Bakos, *Gpapriori: Gpu-accelerated frequent itemset mining*, Cluster Computing (CLUSTER), 2011 IEEE International Conference on, Sept 2011, pp. 590–594.

[52] Lei Zhang, Meikang Qiu, Wei-Che Tseng, and Edwin H.-M. Sha, *Variable partitioning and scheduling for mpsoc with virtually shared scratch pad memory*, J. Signal Process. Syst. **58** (2010), no. 2, 247–265.

[53] Yan Zhang, Fan Zhang, and J. Bakos, *Frequent itemset mining on large-scale shared memory machines*, Cluster Computing (CLUSTER), 2011 IEEE International Conference on, Sept 2011, pp. 585–589.

[54] Yan Zhang, Fan Zhang, Zheming Jin, and Jason D. Bakos, *An fpga-based accelerator for frequent itemset mining*, ACM Trans. Reconfigurable Technol. Syst. **6** (2013), no. 1, 2:1–2:17.

Appendix A

Code of the Benchmark Stencil Loops

```
void MatrixAdd(float* input1,
               float* input2,
               int size_x, int size_y,
               float* output) {
  int i, j;
  for (i = 0; i < size_y; ++i) {
    for (j = 0; j < size_x; ++j) {
      output[i * size_x + j] =
          input1[i * size_x + j] +
          input2[i * size_x + j];
    }
  }
}

void MeanFilter1x3(float* input,
                   int size_x, int size_y,
                   float* output) {
  int i, j;
  for (i = 0; i < size_y ; ++i) {
    for (j = 1; j < size_x - 1; ++j) {
      float v =
          input[(i) * size_x + (j - 1)] +
          input[(i) * size_x + (j)] +
          input[(i) * size_x + (j + 1)];
        output[i * size_x + j] = v * 0.33;
      }
    }
}

void MeanFilter3x3(float* input,
                   int size_x, int size_y,
                   float* output) {
  int i, j;
  for (i = 1; i < size_y - 1 ; ++i) {
    for (j = 1; j < size_x - 1; ++j) {
      float v =
          input[(i - 1) * size_x + (j - 1)] +
          input[(i - 1) * size_x + (j)] +
          input[(i - 1) * size_x + (j + 1)] +
          input[(i) * size_x + (j - 1)] +
          input[(i) * size_x + (j)] +
          input[(i) * size_x + (j + 1)] +
          input[(i + 1) * size_x + (j - 1)] +
          input[(i + 1) * size_x + (j)] +
          input[(i + 1) * size_x + (j + 1)];
      output[i * size_x + j] = v * 0.11;
    }
  }
}
```

```
void Jacobi(float* input,
            int size_x, int size_y,
            float* output) {
  int i, j;
  for (i = 1; i < size_y - 1 ; ++i) {
    for (j = 1; j < size_x - 1; ++j) {
      float v =
          input[(i - 1) * size_x + (j)] +
          input[(i + 1) * size_x + (j)] +
          input[(i) * size_x + (j - 1)] +
          input[(i) * size_x + (j + 1)];
      output[i * size_x + j] = v * 0.25;
    }
  }
}

void GaussianX7(float* input,
                int size_x, int size_y,
                float*restrict output) {
  int i, j;
  for (i = 3; i < size_y - 3; ++i) {
    for (j = 3; j < size_x - 3; ++j) {
      float v =
          input[(i) * size_x + (j - 3)] * 0.006+
          input[(i) * size_x + (j - 2)] * 0.062 +
          input[(i) * size_x + (j - 1)] * 0.242 +
          input[(i) * size_x + (j)] * 0.383 +
          input[(i) * size_x + (j + 1)] * 0.242 +
          input[(i) * size_x + (j + 2)] * 0.061 +
          input[(i) * size_x + (j + 3)]* 0.006;
        output[i * size_x + j] = v;
      }
    }
}

void Sobel(float* input,
           int size_x, int size_y,
           float* output1,
           float* output2) {
  int i, j;
  for (i = 1; i < size_y - 1; ++i) {
    for (j = 1; j < size_x - 1; ++j) {
      float a = input[(i - 1) * size_x + j - 1];
      float b = input[(i - 1) * size_x + j];
      float c = input[(i - 1) * size_x + j + 1];
      float d = input[(i) * size_x + j - 1];
      float e = input[(i) * size_x + j + 1];
      float f = input[(i + 1) * size_x + j - 1];
      float g = input[(i + 1) * size_x + j];
      float h = input[(i + 1) * size_x + j + 1];
      output1[i * size_x + j] = (a + d + f - c - e - h) * 0.5;
      output2[i * size_x + j] = (a + b + c - f - g - h) * 0.5;
    }
  }
}
```

```
void HarrisCorner(float* dx,
                  float* dy,
                  int size_x, int size_y,
                  float* s) {
  int i, j;
  for (i = 1; i < size_y - 1; ++i) {
    for (j = 1; j < size_x - 1; ++j) {
      float a1 = dx[(i - 1) * size_x + j - 1];
      float b1 = dx[(i - 1) * size_x + j];
      float c1 = dx[(i - 1) * size_x + j + 1];
      float d1 = dx[i * size_x + j - 1];
      float e1 = dx[i * size_x + j];
      float f1 = dx[i * size_x + j + 1];
      float g1 = dx[(i + 1) * size_x + j - 1];
      float h1 = dx[(i + 1) * size_x + j];
      float i1 = dx[(i + 1) * size_x + j + 1];

      float a2 = dy[(i - 1) * size_x + j - 1];
      float b2 = dy[(i - 1) * size_x + j];
      float c2 = dy[(i - 1) * size_x + j + 1];
      float d2 = dy[i * size_x + j - 1];
      float e2 = dy[i * size_x + j];
      float f2 = dy[i * size_x + j + 1];
      float g2 = dy[(i + 1) * size_x + j - 1];
      float h2 = dy[(i + 1) * size_x + j];
      float i2 = dy[(i + 1) * size_x + j + 1];
      float xx = a1 * a1 + b1 * b1 + c1 * c1 + d1 * d1 +
           e1 * e1 + f1 * f1 + g1 * g1 + h1 * h1 + i1 * i1;
      float xy = a1 * a2 + b1 * b2 + c1 * c2 + d1 * d2 +
           e1 * e2 + f1 * f2 + g1 * g2 + h1 * h2 + i1 * i2;
      float yy = a2 * a2 + b2 * b2 + c2 * c2 + d2 * d2 +
           e2 * e2 + f2 * f2 + g2 * g2 + h2 * h2 + i2 * i2;

      s[i * size_x + j] = xx + yy - 0.11 * xy;
    }
  }
}
void LucasKanade(float* dx,
                 float* dy,
                 float* dt,
                 int size_x, int size_y,
                 float* vx,
                 float* vy) {
  int i, j;
  for (i = 1; i < size_y - 1; ++i) {
    for (j = 1; j < size_x - 1; ++j) {
      float a1 = dx[(i - 1) * size_x + j - 1];
      float b1 = dx[(i - 1) * size_x + j];
      float c1 = dx[(i - 1) * size_x + j + 1];
      float d1 = dx[i * size_x + j - 1];
      float e1 = dx[i * size_x + j];
      float f1 = dx[i * size_x + j + 1];
      float g1 = dx[(i + 1) * size_x + j - 1];
      float h1 = dx[(i + 1) * size_x + j];
      float i1 = dx[(i + 1) * size_x + j + 1];

      float a2 = dy[(i - 1) * size_x + j - 1];
      float b2 = dy[(i - 1) * size_x + j];
      float c2 = dy[(i - 1) * size_x + j + 1];
      float d2 = dy[i * size_x + j - 1];
      float e2 = dy[i * size_x + j];
      float f2 = dy[i * size_x + j + 1];
      float g2 = dy[(i + 1) * size_x + j - 1];
      float h2 = dy[(i + 1) * size_x + j];
      float i2 = dy[(i + 1) * size_x + j + 1];

      float a3 = dt[(i - 1) * size_x + j - 1];
      float b3 = dt[(i - 1) * size_x + j];
      float c3 = dt[(i - 1) * size_x + j + 1];
      float d3 = dt[i * size_x + j - 1];
      float e3 = dt[i * size_x + j];
      float f3 = dt[i * size_x + j + 1];
      float g3 = dt[(i + 1) * size_x + j - 1];
      float h3 = dt[(i + 1) * size_x + j];
      float i3 = dt[(i + 1) * size_x + j + 1];

      float xx = a1 * a1 + b1 * b1 + c1 * c1 + d1 * d1 +
           e1 * e1 + f1 * f1 + g1 * g1 + h1 * h1 + i1 * i1;
      float xy = a1 * a2 + b1 * b2 + c1 * c2 + d1 * d2 +
           e1 * e2 + f1 * f2 + g1 * g2 + h1 * h2 + i1 * i2;
```

```
      float yy = a2 * a2 + b2 * b2 + c2 * c2 + d2 * d2 +
           e2 * e2 + f2 * f2 + g2 * g2 + h2 * h2 + i2 * i2;
      float xt = a1 * a3 + b1 * b3 + c1 * c3 + d1 * d3 +
           e1 * e3 + f1 * f3 + g1 * g3 + h1 * h3 + i1 * i3;
      float yt = a2 * a3 + b2 * b3 + c2 * c3 + d2 * d3 +
           e2 * e3 + f2 * f3 + g2 * g3 + h2 * h3 + i2 * i3;

      float det = xx * yy - xy * xy;

      vx[i * size_x + j] = (-yy * xt + xy * yt) / det;
      vy[i * size_x + j] = (xx * yt - xy * xt) / det;
    }
  }
}
```

Appendix B

Implementation of Code Generation with LLVM

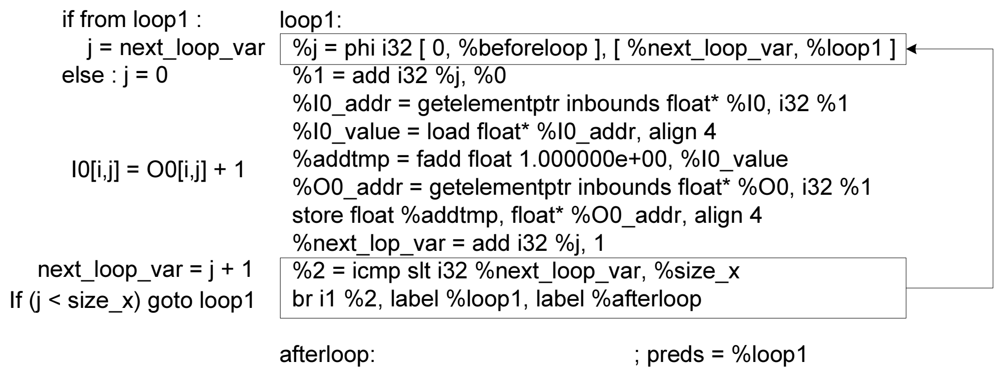
B.1 Generate LLVM Loop Structure in SSA format

The LLVM infrastructure uses Static Single Assignment (SSA) rule for its code generation. The SSA rule allow each variable be placed on the left side of the assignment only once. Because the loop variables are initialized at the beginning of the loop and updated at the end of the loop. They need to be semantically assigned at least twice. In order to generate loops that complies to SSA rule, there are many ways to solve this multiple assignment problem. We choose a code generation technique among these solutions called phi node shadowing to solve it. Phi node is a conditional assignment that allows the code to assign value to a variable based on which predecessor block that the control flow is passed from. It allows the current block to inherit the updated values of the loop variables from a predecessor block (The predecessor of a basic block could be itself). The reassignment of the loop variables can be solved by using phi node in a proper way, as shown in Figure B.1.

We use a recursive algorithm to generate nested loop structure. The input of the algorithm is the margin sizes of x and y direction from margin detector and the abstract syntax tree.

B.2 Generate Stencil Body from Abstract Syntax Tree

Each node in the abstract syntax tree is implemented with a CodeGen method that is used to generate and output LLVM IR code that is used to perform the arithmetic



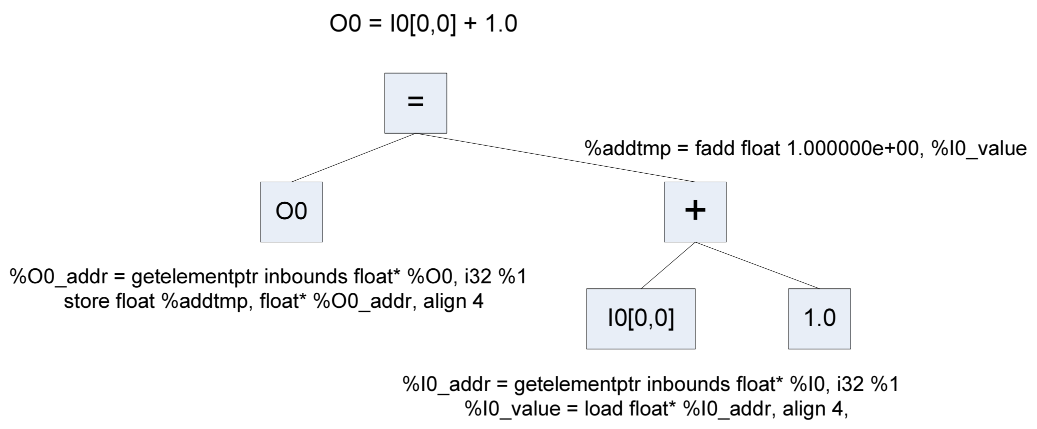
```
loop1:
 %j = phi i32 [ 0, %beforeloop ], [ %next_loop_var, %loop1 ]
 %1 = add i32 %j, %0
 %I0_addr = getelementptr inbounds float* %I0, i32 %1
 %I0_value = load float* %I0_addr, align 4
 %addtmp = fadd float 1.000000e+00, %I0_value
 %O0_addr = getelementptr inbounds float* %O0, i32 %1
 store float %addtmp, float* %O0_addr, align 4
 %next_lop_var = add i32 %j, 1
 %2 = icmp slt i32 %next_loop_var, %size_x
 br i1 %2, label %loop1, label %afterloop

afterloop:                               ; preds = %loop1
```

Figure B.1: Using Phi Node to Eliminate Reassignments in LLVM Loops

operation defined by the node. The methodologies of LLVM code generation for different types of nodes are listed below.

- Constant node: Call LLVM Type::getFloatTy to generate a floating point constant initialization instruction.
- Input field node: Compute the read address from the offsets using "getelementptr" instruction and generate load instructions for reading in the value from input matrix.
- Output field node: Compute the write address from the offsets using "getelementptr" instruction and generate store instruction to assign the result value into the output matrix.
- Arithmetic node: Generate arithmetic instruction that perform computation on the two input operands.

Figure B.2 shows an example of how the LLVM code is generated from traversing and calling CodeGen from the abstract syntax tree. The stencil code generation are called after the loop structure is generated. The loop structure generation procedure creates a basic block inside the most inner loop, calls the stencil code generation



O0 = I0[0,0] + 1.0
=
%addtmp = fadd float 1.000000e+00, %I0_value
O0
+
%O0_addr = getelementptr inbounds float* %O0, i32 %1
store float %addtmp, float* %O0_addr, align 4
I0[0,0]
1.0
%I0_addr = getelementptr inbounds float* %I0, i32 %1
%I0_value = load float* %I0_addr, align 4,

Figure B.2: Example of a Stencil Syntax Tree and LLVM Code Generation

procedure and insert the generated code into the basic block. At the end of the code generation, we use the LLVM function validation module to make a sanity check of the generated code.