2011

A study of the scale-invariant feature transform on a parallel pipeline

Phaneendra Vinukonda
Louisiana State University and Agricultural and Mechanical College

A STUDY OF THE SCALE-INVARIANT FEATURE TRANSFORM ON A PARALLEL PIPELINE

Thesis

Submitted to the Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

in

The Department of Electrical and Computer Engineering

by
Phaneendra Vinukonda
B.TECH., JNTU University, 2007
May 2011

Acknowledgements

I am indebted to my major advisor Dr. Ramachandran Vaidyanathan for his exemplary patience, guidance and support. During my stay here at LSU, he taught me the skills of problem solving, provided me with some good motivations, helped me through the difficulties I have gone through on the way towards this degree. It was with his kind support, I overcame all the obstacles on my way towards completing my thesis.

I would also like to thank my committee members, Dr. S. Rai and Dr. Gunturk for their valuable suggestions and kind support. Furthermore, I thank the Dept. of Electrical and Computer Engineering for making me concentrate on my research without any other deviations.

I wish to endow my earnest gratitude to my parents, who believed in me and have been thorough all the rough times. I also want to thank my brother V. Hareendra, my entire family and friends for their affection, support and compassion.

Table of Contents

List of Figures

List of Tables

Abstract

In this thesis we study the running of the Scale Invariant Feature Transform (SIFT) algorithm on a pipelined computational platform. The SIFT algorithm is one of the most widely used methods for image feature extraction.

We develop a tile based template for running SIFT that facilitates the analysis while abstracting away lower-level details. We formalize the computational pipeline and the time to execute any algorithm on it based on the relative times taken by the pipeline stages. In the context of the SIFT algorithm, this reduces the time to that of running the entire image through a bottlenecked stage and the time to run either the first or last tile through the remaining stages. Through an experimental study of the SIFT algorithm on a broad collection of test images, we determined image feature fraction values, that relate the sizes of the image extracts as it the computation proceeds through the stages of the SIFT algorithm.

We show that for a single chip uniprocessor pipeline, the computational stage is the bottleneck. Specifically we show that for an $N \times N$ image with $n \times n$ tiles the overall time complexity is $\Theta\left(\frac{(n+x)^2}{p_i}\Gamma_0 + \alpha\beta N^2 x^2 \Gamma_1 + \frac{(\alpha\beta+\gamma)\, n^2 \log x}{p_o}\Gamma_2\right)$; here x is the neigborhood of the tile, p_i, p_o are the number of input, output pins of the chip, α, β, γ are the feature fractions, and $\Gamma_0, \Gamma_1, \Gamma_2$ are the input, compute, output clocks. The three terms in the expression represents the time complexities of input, compute and output stages. The input and output stages can be slowed down substantially without appreciate degradation of the overall performance. This slowdown can be traded off for lower power and higher signal quantity.

For multicore chips, we show that for an $N \times N$ image on a P-core chip, the overall time complexity to process the image is $\Theta\left(\frac{N^2}{p_i}\Gamma_0 + \frac{(n^2w^2 + \alpha\beta n^2 x^2)}{P}\Gamma_1 + \frac{(\alpha\beta+\gamma)\, n^2 \log x}{p_o}\Gamma_2\right)$; in addition to the quantities described earlier w is the window size used for the Gaussian blurring. Overall we establish that without improvements in the input bandwidth, the power of multicore processing cannot be used efficiently for SIFT.

Chapter 1

Introduction

The speed of processors has increased exponentially in modern systems but the rate at which data enters and exits a processor has not kept up with this increase. This is because while technological improvements have been able to keep pace with Moore's law for many decades, the physical size of input/output pins of a chip cannot be reduced beyond a point due to mechanical stability reasons. Jordon [12] presents a comparative chart of number of transistors and pins in Intel chips over the last 20 years. While the number of transistors has gone up by a factor of 20000, the number of pins has increased only by a factor of 30 during the same period. Three-dimensional stacking [2] [5] that allows better connectivity within chips and optical input/output [26] are promising future possibilities.

Currently, applications requiring high input/output bandwidth are mostly executed on a single chip environment. This is not because there is not enough computational need to spread the algorithm across chips, but because its not economical for the data to leave a chip and go to the next. As a result, one cannot exploit the benefits of high parallelism that result in better speeds and more sustainable use of power. Applications that deal with image and video processing (particularly those with real time constraints) require large input/output bandwidth. It is one of these applications that we study in this thesis.

We study the scale invariant feature transform (SIFT) algorithm [13] that extracts features of an image in a manner that is stable over image translation, rotation, scaling, illumination and camera viewpoint. We have selected SIFT as it is one of the most widely used algorithms for object recognition, that has been employed in many applications such as face/object recognition [13] [14] [15], robot localization and mapping [16], 3D-scene modelling, and action recognition [3]. SIFT accepts an $N \times N$ image as input and produces a set of features. The input bandwidth of N^2 pixels can be very high for large values of N

(modern household cameras produce images with N^2 is more than 10M so $N \geq 1000$ is not unreasonable).

In this thesis we develop a c-chip (or $(2c+1)$-stage) pipeline as a basic platform to which we execute the SIFT algorithm for an $N \times N$ image. It should be pointed out that this work focuses on running SIFT on the pipeline platform, rather that improve the performance of the original algorithm of Lowe [13]. The choice of a pipeline platform suits the structure of SIFT and many real time applications that stream in image data. More specifically, we develop general results for this pipeline and apply this to single- or two-chip (3- or 5-stage) pipeline. Both uniprocessor and multicore chips are considered. We use the single-chip uniprocessor (3-stage) pipeline model as a base and we study the case where the processing platform is a multicore chip. We extend this to a two-chip (5-stage) pipeline model. where each chip could be uniprocessor or multicore. To analyze SIFT in a manner that abstracts lower-level details, we introduce a decomposition of the $N \times N$ image into smaller $n \times n$ tiles. We consider two orders in which these tiles are fed as input to the pipeline. The order makes a difference in the processing time for certain cases. We derive expressions for the time and memory complexities for SIFT on a pipeline model. We also study several images each at 20 different resolutions, using the SIFT implementation of Hess [19]. This helps us further refine some of the constraints in the time complexity of SIFT.

This thesis has contributions in many directions. We show that for the uniprocessor case, the input/output bandwidth is not critical, as the computation is the bottleneck. However as we move to a multicore platform, the input/output bandwidth becomes a bottleneck, particularly when the pipeline (sequence of chips) is deeper. Specifically, we show that the time to run SIFT on an image on a uniprocessor pipeline is essentially the time to compute SIFT for the entire image. The only additional contribution due to the input and output stages is the time to input the first tile and the time to output the features of the last tile (these are very small fractions of the image). As we move to the multicore platform, the time to run SIFT essentially equals the time to input the image. The additional times due the compute and output stages are only the times for computation and feature output of the last tile (again a very small fraction of the image). In the uniprocessor case, the computation time is the bottleneck and other stages (input and output) idle much of the time. As the number of cores increases, the input is the bottleneck and cores begin to idle. We also develop general results for the running time on a $c-$chip pipeline which may be of independent interest.

The results we develop in this work also point to directions in which the performance of SIFT can be improved when run on the pipeline model. In the single-chip uniprocessor case, where the input is not a bottleneck, one could slow the input clock rate to save power or transmit additional bits to improve the bit error rate without affecting the overall time. In the multicore case, the full computational power of multicore chips cannot be used unless

the input bandwidth is improved.

Recent research on SIFT has followed many directions including parallel implementation and optimization, application areas and modifications to the SIFT technique of Lowe [13]. Sinha *et al.* [22] and Heymann *et al.* [8] proposed SIFT implementation on GPUs. Wen *et al.* [23] proposed a CUDA based implementation for a GPU framework and analyzed its parallelism. Ko *et al.* [11] analyzed the performance and cost of SIFT for visual classification and he discussed tradeoffs among system parameters that affect the energy, accuracy and latency. Nasir *et al.* [7] proposed a method that improves SIFT. Lin *et al.* [27] proposed a tracking method using SIFT for recording the trajectory of the human motion in an image sequence. Mikolajczyk and Schmid proposed gradient location-orientation histogram (GLOH) [17] as an extension of the SIFT descriptor designed to increase its robustness and distinctiveness. Zhong *et al.* [24], presented an improvement on the basic SIFT that is geared toward palmprint recognition. Shekar *et al.* [21] proposed improved descriptor representation in the face recognition context. Zhang *et al.* [28] proposed two parallel SIFT algorithms and presented some optimization techniques to improve the performance on multicore systems. However their results focus less on the analysis for a general model such as the HM model [4] that we consider. We are not aware of any work similar to this one that theoretically analyzes the performance of SIFT on a general pipeline platform including uniprocessor and multicore chips.

The remainder of this thesis is organized as follows. In Chapter 2, we explain the major phases of the SIFT algorithm and analyze it to determine the number of operations needed to perform these phases. In Chapter 3, we describe experiments involving running SIFT on different images to further refine our theoretical analysis of the running time of SIFT. We also introduce the concept of feature fractions in this chapter that play an important part of subsequent analysis. In Chapter 4, the two tile ordering methods are introduced (row major and diagonal). In Chapter 5, we describe the pipeline model. Chapter 6, is devoted to an analysis of the input complexity of SIFT. In Chapter 7, we derive the expressions for time and memory to run SIFT on a single-chip uniprocessor pipeline model. We extend this in Chapter 8, to a single-chip multicore pipeline. In Chapter 9, a two-chip model is considered where each chip can be single or multicore. Finally in Chapter 10, we summarize the work and identify directions for future research.

Chapter 2

Scale Invariant Feature Transform

In this chapter we briefly discuss the Scale Invariant Feature Transform (SIFT) algorithm introduced by Lowe [13]. This algorithm is one of the most widely used one for image feature extraction. SIFT extracts image features, that are stable over image translation, rotation and scaling and somewhat invariant to changes in the illumination and camera viewpoint. The SIFT algorithm has four major phases (as illustrated in Figure 2.1) (a) Extrema Detection, (b) Keypoint Localization, (c) Orientation Assignment, (d) Keypoint Descriptor Generation. The first phase, Extrema Detection, examines the image under various scales and octaves (explained in detail later) to isolate points of the picture that are different from their surroundings. These points, called *extrema*, are potential candidates for image features.

The next phase, Keypoint Detection, starts with the extrema and selects some of these points to be *keypoints*, that are a whittled down a set of feature candidates. This refinement rejects extrema, that are caused by edges of the picture and by low contrast points.

The third phase, Orientation Assignment, converts each keypoint and its neighborhood into a set of vectors by computing a magnitude and a direction for them. It also identifies other keypoints that may have been missed in the first two phases; this is done on the basis of a point having a significant magnitude without being an extremum. The algorithm now has identified a final set of keypoints.

The last phase, Keypoint Descriptor Generation, takes a collection of vectors in the neighborhood of each keypoint and consolidates this information into a set of eight vectors called the *descriptor*. Each descriptor is converted into a feature by computing a normalized sum of these vectors.

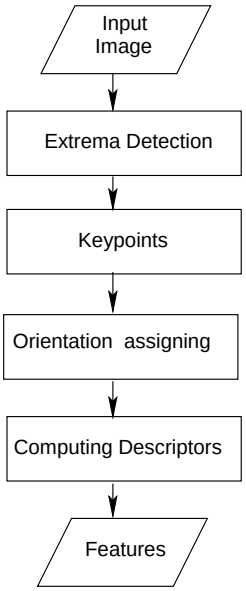


Figure 2.1: Major phases of the SIFT algorithm

2.1 Flow of Data in SIFT

In this section, we describe the nominal number of data items traversing each phase of the SIFT algorithm. This is used later to determine the algorithm complexity and in the experimental study (see Chapter 3). Recall that the Extrema Detection and Keypoint Detection phases reduce the number of feature candidates. The Orientation Assignment phase potentially adds points to this number of feature candidates.

The input to the SIFT algorithm is a set of N^2 pixels of an $N \times N$ image. Only a small fraction of these pixels typically turn out to be extrema. Let $0 < \alpha < 1$ be this fraction. So αN^2 extrema will move on to the next Keypoint Detection phase. Only a small fraction of these extrema will qualify as keypoints. Let $0 < \beta < 1$ be this fraction. So nominally there are $\alpha\beta N^2$ keypoints at this stage. Orientation assignment reexamines all the N^2 points in the image to check if any points of significant magnitude have been missed. If so, they are added to the set of keypoints. Let a fraction γ of the image pixels qualify to be these added keypoints. That is, γN^2 new keypoints are added. The Compute Descriptors phase converts these points into vectors which are then turned into features. The number of feature descriptors output by SIFT algorithm is nominally $(\alpha\beta + \gamma)N^2$ for an $N \times N$ image. We call the quantities α, β and γ as *feature fractions*.

While α, β and γ will depend on the picture in question, we will consider nominal values

averaged over many pictures to guide this work. This part of the study is described in Chapter 3. In the remaining sections of this chapter, we describe the major four phases of SIFT in detail.

2.2 Scale-Space Extrema Detection

This is the first phase of the SIFT algorithm. Here the algorithm identifies the points that are stable with respect to image rotation, translation and those that are minimally affected by noise and small distortions. Detecting these points can be accomplished by searching for stable features across all possible scales (defined below). Figure 2.2 shows the internal stages of the Extrema Detection phase. The algorithm compute "scale," "difference of gaussians," and "extrema" over several "octaves." We now discuss these ideas, before explaining the order in which they are computed.

Scale: Let I be an $N \times N$ image and for $0 \leq x, y < N$, let $G(x, y, \sigma) = \frac{1}{2\pi\sigma^2}\ e^{-\frac{x^2+y^2}{2\sigma^2}}$ be the discrete two-dimensional Gaussian function. Then the *scale* of the image I is defined as $L(\sigma) = \{\ G(x, y, \sigma) * I(x, y) : 0 \leq x, y < N\ \}$ where $*$ is the two-dimensional convolution operation and $I(x, y)$ is the pixel at row x and column y of image $I(x, y)$.

In general, the k^{th} scale of the image, for $k \geq 1$ is defined as $L(k\sigma) = \{\ G(x, y, k\sigma) * I(x, y) : 0 \leq x, y < N\ \}$. For each image point $I(x, y)$, the scale is computed by applying a scalar product between the point $I(x, y)$ and a $w \times w$ Gaussian weighted window placed over that point. For example, suppose that point $I(x, y)$ is the central point $i_{0,0}$ of the 5×5 window shown in Figure 2.3(a). The figure also shows the pixel values of the 5×5 neighborhood of this point. If $\sigma = \sqrt{2}$ and $w = 5$, then the Gaussian filter $G(u, v)$ can be shown to be the one in Figure 2.3 (b). Applying this filter to the central point is computing the quantity

$$G(i_{0,0}) = 0.001 * i_{-2,2} + 0.003 * i_{-1,2} + \cdots + 0.145 * i_{0,0} + \cdots + 0.003 * i_{1,-2} + 0.001 * i_{2,-2}$$

In general for a $w \times w$ window with odd w, the image points located around the point $I(x, y)$ are $I(x + u, y + v)$ where $-\frac{w-1}{2} \leq u, v \leq \frac{w-1}{2}$. Here the the scale of $I(x, y)$ is

$$L(x, y, \sigma) = \sum_{u=-\frac{w-1}{2}}^{\frac{w-1}{2}} \sum_{v=-\frac{w-1}{2}}^{\frac{w-1}{2}} G(u, v) I(x + u, y + v)$$

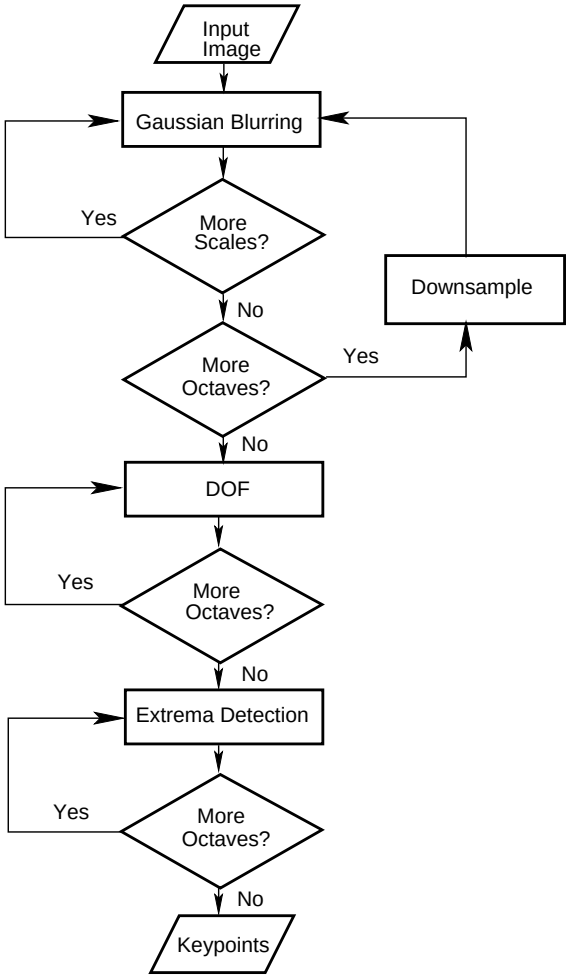


Figure 2.2: The internal stages of Scale-Space Extrema Detection

Thus, computing scale for each point $I(x, y)$ requires w^2 multiplications and w^2-1 additions which has $\Theta(w^2)$ complexity. For the entire $N \times N$ image the complexity for this step is $\Theta(w^2N^2)$.

Let $s \geq 1$ be an integer and $k = 2^{\frac{1}{s}}$. The SIFT algorithm repeatedly computes the scale of the image as described below.

Let σ_0 be the initial value of σ in the Gaussian filter. Define $\sigma_i = k^i\sigma$ for $0 \leq i < s + 3$. Let $L_0^0 = I$ be the original image (the superscript is explained later). For image element $I(x, y)$, define $L_{i+1}^0(x, y) = G(x, y, \sigma_i) * L_i^0(x, y)$ where $0 \leq i < s + 3$. In this fashion the

$i_{-2,2}$	$i_{-1,2}$	$i_{0,2}$	$i_{1,2}$	$i_{2,2}$
$i_{-1,2}$	$i_{-1,2}$	$i_{0,1}$	$i_{1,1}$	$i_{1,2}$
$i_{0,-2}$	$i_{0,-1}$	$i_{0,0}$	$i_{0,1}$	$i_{0,2}$
$i_{-1,-2}$	$i_{-1,-2}$	$i_{0,-1}$	$i_{1,-1}$	$i_{1,-2}$
$i_{-2,-2}$	$i_{-1,-2}$	$i_{0,-2}$	$i_{1,-2}$	$i_{2,-2}$

(a) Pixel Values

0.001	0.003	0.004	0.003	0.001
0.003	0.008	0.113	.008	0.003
0.004	0.113	0.145	0.113	0.004
0.003	0.008	0.113	0.008	0.003
0.001	0.003	0.004	0.003	0.001

(b) Gaussian window

Figure 2.3: An example of applying a 5×5 Gaussian window on a point

algorithm computes the scales as shown below (also see Figure 2.4).

$$L_0^0 \overset{G_\sigma}{\rightarrow} L_1^0 \overset{G_{\sigma_1}}{\rightarrow} L_2^0 \cdots \overset{G_{\sigma_s}}{\rightarrow} L_{s+1}^0 \overset{G_{\sigma_{s+1}}}{\rightarrow} L_{s+2}^0 \overset{G_{\sigma_{s+2}}}{\rightarrow} L_{s+3}^0 \tag{2.1}$$

Octaves: The sequence of scales in Equation (2.1) is called an *octave*. As discussed above we computed L_{s+1}^0 as part of the first octave. This is a blurred image from the original image I. The next step requires a reduction in image resolution. The resolution of an image can be reduced[1] by a factor of 2 in each dimension by sampling every other pixel of the image in a checkerboard pattern. Let L_0^1 be L_{s+1}^0 reduced in resolution by a factor of 2 (the superscript j here denotes Octave 1 for L_0^1 and Octave 0 for L_{s+1}^0).

We now define a new octave (second octave) analogous to the Octave 0 (see Equation (2.1)).

$$L_0^1 \overset{G_\sigma}{\rightarrow} L_1^1 \overset{G_{\sigma_1}}{\rightarrow} L_2^1 \cdots \overset{G_{\sigma_s}}{\rightarrow} L_{s+1}^1 \overset{G_{\sigma_{s+1}}}{\rightarrow} L_{s+2}^1 \overset{G_{\sigma_{s+2}}}{\rightarrow} L_{s+3}^1 \tag{2.2}$$

If there are $\widehat{s}$ octaves, then in general for $0 < j \leq \widehat{s} - 1$, L_0^j is L_{s+1}^{j-1} reduced in resolution by a factor of 2 in each dimension and

$$L_0^j \overset{G_\sigma}{\rightarrow} L_1^j \overset{G_{\sigma_1}}{\rightarrow} L_2^j \cdots \overset{G_{\sigma_s}}{\rightarrow} L_{s+1}^j \overset{G_{\sigma_{s+1}}}{\rightarrow} L_{s+2}^j \overset{G_{\sigma_{s+2}}}{\rightarrow} L_{s+3}^j \tag{2.3}$$

The time complexity for computing all $(s + 3)$ scales of the image over one octave is

[1]Image resolution can be reduced in other ways, example averaging over a 2×2 pixel set.

$\Theta\left(N^2w^2(s+3)\right)$ and repeating this for $\widehat{s}$ octaves gives a time complexity of $\Theta\left(\sum_{j=0}^{\widehat{s}-1}\frac{N^2}{2^j}w^2s\right)$ $=\Theta\left(N^2ws\right)$ where the down sampling needed to start each octave requires another $\Theta\left(\frac{N^2}{2^j}\right)$ operations. Therefore the overall complexity of this phase is $\Theta\left(N^2w^2s\right)$. Considering constants and unit time for all operations, this phase requires approximately $4N^2w^2s$ time.

Difference of Gaussians: At this point, we have $(s+3)$ scales L_i^j over all $\widehat{s}$ octaves where $0 \leq i < s+3$ and $0 \leq j < \widehat{s}$. For any fixed octave j and $0 \leq i < s+2$, define the i^{th} *difference of Gaussians* over octave j as:

$$D_i^j = L_{i+1}^j - L_i^j$$

where the difference is for each pair of corresponding pixels $L_{i+1}^j(x,y)$ and $L_i^j(x,y)$. Figure 2.4 illustrates these ideas.

The scales in octave j are $\frac{N}{2^j} \times \frac{N}{2^j}$ images. Then each of L_{i+1}^j and L_i^j is an $\frac{N}{2^j} \times \frac{N}{2^j}$ image and so computing D_i^j requires finding $\frac{N^2}{2^j}$ differences. Thus, computing all $s+2$ difference of Gaussians in octave j requires $\frac{(s+2)N^2}{2^j} = \Theta\left(\frac{sN^2}{2^j}\right)$ units of time. The complexity to compute this quantity across all $\widehat{s}$ octaves is $\Theta\left(\sum_{j=0}^{\widehat{s}-1}\frac{sN^2}{2^j}\right) = \Theta(sN^2)$. With normally used values of SIFT parameters, the number of operations is approximately $4N^2s$.

Extrema Detection: Suppose that we have the sets D_{i-1}^j, D_i^j, D_{i+1}^j of difference of Gaussian images in an octave j. For each octave j where $0 \leq j < \widehat{s}$ and for $1 < i < s+2$, place the difference of gaussians $D_{i-1}^j, D_i^j, D_{i+1}^j$ in three adjacent layers (as shown in Figure 2.5). Now element $D_i^j(x,y)$ (shown in Figure 2.5 as a dark square) has 26 neighboring difference of Gaussian elements (as shown as gray squares in Figure 2.5). Element $D_i^j(x,y)$ is an *extremum* iff it is strictly larger (in pixel value) than all of the neighboring elements or it is strictly smaller than all of the neighboring elements.

Detecting whether $D_i^j(x,y)$ is an extremum takes at most 26 comparisons each requiring constant time. For all elements of D_i^j in an octave j, the time needed is $\Theta\left(\frac{N^2}{2^j}\right)$. For all difference of Gaussians over all $\widehat{s}$ octaves, the time needed is $\Theta\left(\sum_{j=0}^{\widehat{s}-1}\frac{(s+2)N^2}{2^j}\right) =$ $\Theta\left(sN^2\right)$. Taking the proportionality constant into the account the overall complexity

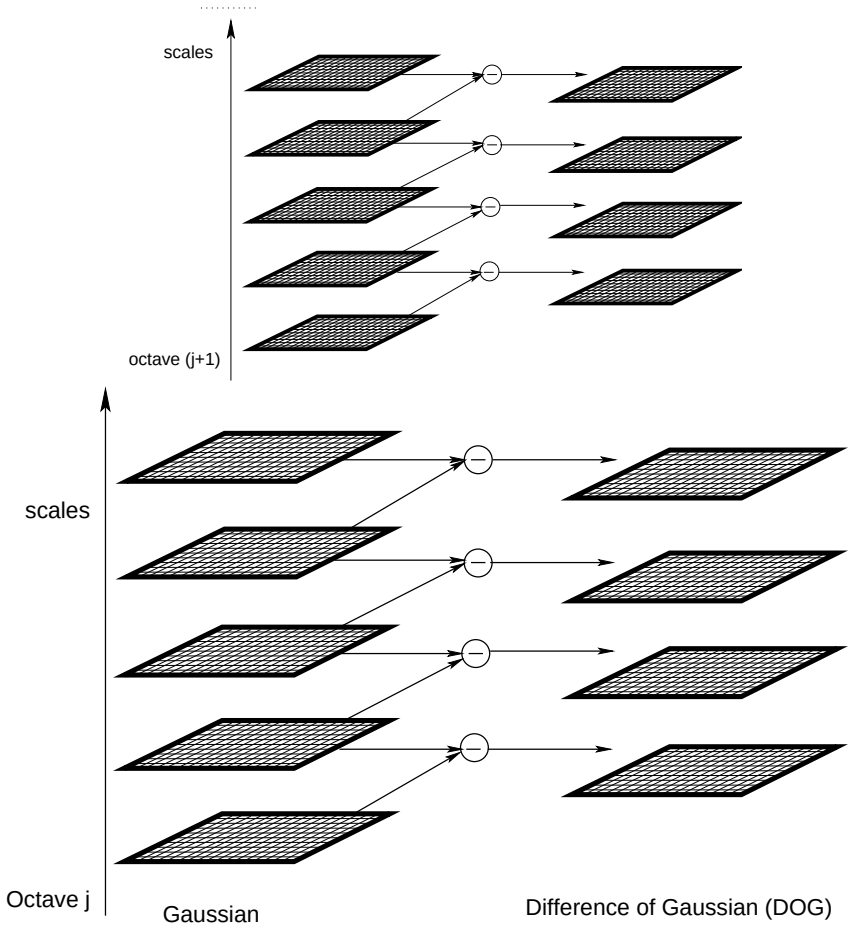


Figure 2.4: Scale, octaves and difference of Gaussians

can be shown to be approximately $104sN^2$. As discussed earlier we nominally have αN^2 extrema at this point of the algorithm.

Before we proceed to the next phase (Keypoint Detection), we touch upon how the Scale-space Extrema Detection phase is executed. As we noted in this section, the algorithm determines, scales (see Section 2.2), difference of Gaussians (see Section 2.2), and extrema (see Section 2.2) independently for $\widehat{s}$ octaves. Thus, it is possible to have an outer loop over octaves and determine the other quantities one after the other within this outer loop (see Figure 2.2). However each of these (scales, difference of Gaussians, extrema) have relatively local dependencies. That is, to determine the scale of a point, one needs to know only the $w \times w$ neighborhood of the point. To determine the difference of Gaussian, we only need two corresponding points and to check whether a point is extremum, we only need

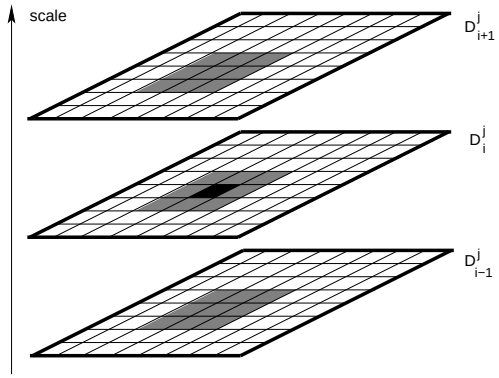


Figure 2.5: Extrema detection on octave j

26 difference of Gaussians points spread over three scales around it. Thus, it is possible to execute these operations over octaves in many different ways. The original algorithm of Lowe [13] use the structure in shown in Figures 2.2. The program we used [19] for this work uses the modified flow of algorithm [28] as shown in Figure 2.6.

2.3 Keypoint Detection

Recall that the algorithm first determines αN^2 extrema and then further distills them into $\alpha\beta N^2$ keypoints, that will ultimately become keypoints of the image. In this section, we discuss the phase of selection of keypoints from extrema. The Scale-Space Extrema Detection phase of the algorithm identifies αN^2 potential candidates for keypoints. Some of these candidates may lie along an edge of the image or may corresponds to points of low contrast. These are generally not useful as features as they are unstable over image variation [13]. Hence these points are rejected. For rejecting low contrast points, each extremum is examined using a method that involves solving a system of 3×3 linear equations and so it takes constant time. To detect the extrema on edges, a 2×2 matrix is generated and simple computations performed on it (including finding the determinant and the trace of 2×2 matrix all requiring $\Theta(1)$ time), to generate a ratio of principle of curvatures. This quantity is simply compared with a threshold value to decide whether an extremum is to be rejected or not. Thus, this phase runs in $\Theta(\alpha s N^2)$ time over all octaves. Considering constants into the account this phase takes approximately $100 s \alpha N^2$ operations.

After the elimination of extrema points, the points that remain are called *keypoints*. We now nominally have $\alpha\beta N^2$ keypoints.

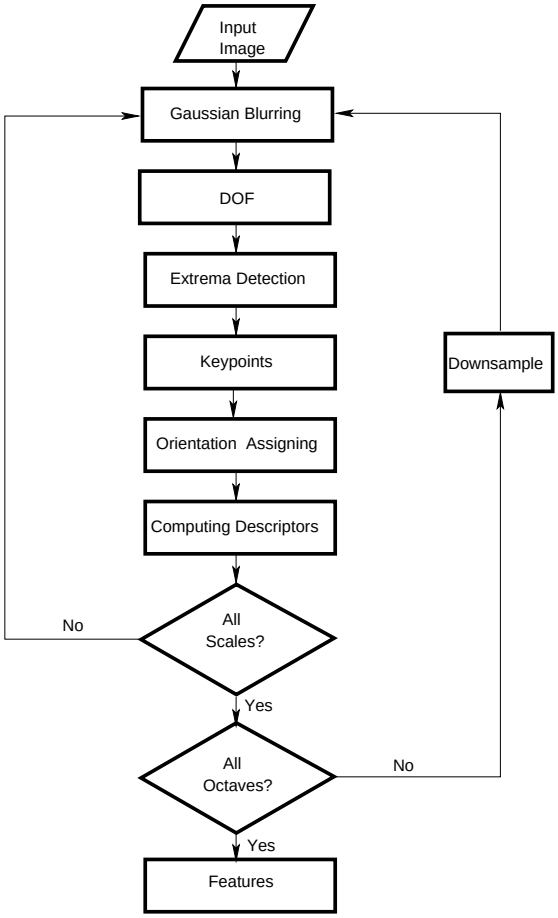


Figure 2.6: Restructured SIFT algorithm flow of data

2.4 Orientation Assignment

The nominal number of keypoints at the start of this phase is $\alpha\beta N^2$. This phase adds to the set of keypoints (those that may be missed in the previous phases) on the basis of their magnitude and orientation. The magnitude $m_i^j(x, y)$ and orientation $\theta_i^j(x, y)$ for each point $L_i^j(x, y)$ can be calculated as follows:

$$m_i^j(x,y) = \sqrt{(L_i^j(x+1,y) - L_i^j(x-1,y))^2 + (L_i^j(x,y+1) - L_i^j(x,y-1))^2}$$

$$\theta_i^j(x,y) = \tan^{-1} \frac{(L_i^j(x,y+1) - L_i^j(x,y-1))}{(L_i^j(x+1,y) - L_i^j(x-1,y)}$$

Non-keypoint points whose magnitudes are close to the peak magnitude are added as new keypoints. The number of points examined is $N^2 - \alpha\beta N^2 \cong N^2$ as α and β are small fractions. Of these, a fraction γ are added back. Thus, the total number of keypoints at the end of this phase is $\alpha\beta N^2 + \gamma\left(N^2 - \alpha\beta N^2\right) = \alpha\beta N^2(1-\gamma) + \gamma N^2 \cong N^2\left(\alpha\beta + \gamma\right)$ again because γ is a small fraction. Clearly the computation for $m_i^j(x,y)$ and $\theta_i^j(x,y)$ can be done over constant time. The overall complexity for all points over all octaves is $\Theta(sN^2)$. Considering the constants, the number of operations is approximately $48sN^2$

2.5 Keypoint Descriptor Generation

In this phase, the algorithm computes a descriptor for each keypoint identified so far. The descriptor is a collection of information in an $2x \times 2x$ neighborhood of the keypoint (the work of Lowe [13] considers a 16×16 neighborhood, which we generalize to $2x \times 2x$). The following tasks are undertaken for each keypoint.

- The magnitudes of all the points in the neighborhood are smoothed by a normalized Gaussian filter with $\sigma = x$. This requires $\Theta(x^2)$ multiplications for each point.
- The neighborhood is divided into 4×4 regions. In each region the vectors (magnitude and direction of points) are histogrammed into 8 buckets covering 360°using trilinear interpolation [13]. Again this requires $\Theta(x^2)$ time for the neighborhood.
- The feature is computed from these descriptors in the neighborhood by computing a normal of the descriptors in the neighborhood.
- The resulting descriptor is represented as an normalized $\frac{x}{2} \times \frac{x}{2}$ descriptor array each with an 8 bucket histogram of vectors. Thus, the feature is $\log_2 \frac{8x^2}{4} = 2\log x + 1$ bits long.

As the time complexity is $\Theta(x^2)$ for each keypoint identified so far, then the overall time complexity for all the keypoints is $\Theta\left(x^2(\alpha\beta + \gamma)N^2\right)$. Considering the constants[2], the number of operations is approximately $1520x^2(\alpha\beta + \gamma)N^2$.

The overall time complexity of the SIFT algorithm is determined from the complexities of the phases discussed so far. This is shown in Table 2.1.

[2]For this phase the constants were obtained through the experiments described in Chapter 3

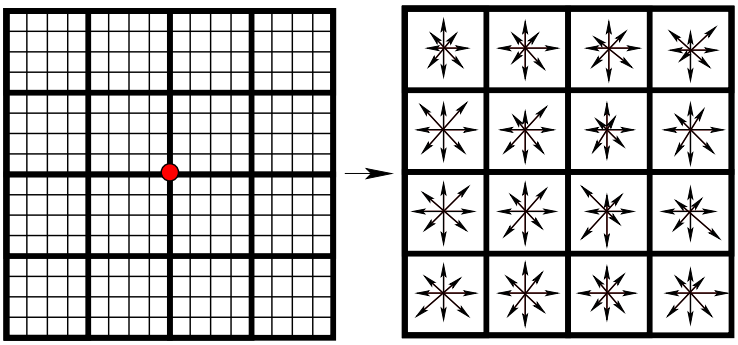


Figure 2.7: Keypoint descriptor generation

Table 2.1: The time complexity and the number of operations required by the different phases of the SIFT algorithm for N^2 pixels

Phase	Complexity	Number of operations
Gaussian Blurring	$\Theta\left(N^2w^2s\right)$	$4N^2w^2s$
Difference of Gaussian	$\Theta\left(sN^2\right)$	$4N^2s$
Scale-space Extrema Detection	$\Theta\left(sN^2\right)$	$104sN^2$
Keypoint Detection	$\Theta\left(\alpha sN^2\right)$	$100s\alpha N^2$
Orientation Assignment	$\Theta\left(sN^2(1-\alpha\beta)\right)$	$48sN^2$
Keypoint Descriptor Generation	$\Theta\left(x^2N^2(\alpha\beta+\gamma)\right)$	$1520x^2(\alpha\beta+\gamma)N^2$

Chapter 3

Experimental Study

The purpose of this chapter is to experimentally study the time requirement for different phases of the SIFT algorithm and to detect, if possible, any trend in the values of feature fractions α, β and γ (see Section 2.1). We performed this study using a SIFT implementation by Hess [19]. In Section 3.1, we describe the pictures used in our study, in Section 3.2 we discuss the time taken by the different phases of the SIFT algorithm and in Section 3.3 we describe the feature fraction values used in this work.

3.1 Images Used in Study

We selected a range of pictures (see Figure 3.1) to test the SIFT algorithm. These pictures were obtained from the Internet [30]– [54].

Images fall in the following categories.

1. Airplanes (four pictures)
2. Spheres (four pictures)
3. Portraits (four pictures each of Einstein and Gandhi, and three pictures of Sandra Day O'Connor)
4. Vehicles (three pictures)
5. Palm-like trees (four pictures)

The pictures represent a variety of themes with enough elements within each category to facilitate detection of patterns. Each picture was converted into a gray scale portable network graphics (PNG) image with 2000×2000 pixels. We also considered 19 reduced resolutions of each original picture. These resolutions are at 1900×1900, 1800×1800, $\cdots$, 100×100. Thus we have 26 "pictures," each at 20 resolutions, totalling 520 "images" in the study. We use the term "image" to represent any of these 520 elements. The term "picture" is used for the 26 sets, each of 20 images of varying resolutions. Figure 3.1 shows a representative picture of each of these sets.

3.2 Time Taken by Different Phases

The SIFT implementation [19] that we used in this work is a large piece of code (over 2500 lines) with several calls to routines, including openCV [18] functions whose code is not explicitly available in the program. This makes it virtually impossible to trace through all parts of the code to determine their execution times.

Our first aim is to identify the major parts of the code that fit a sequential pipeline (see Chapter 5). We do not wish to separately consider functions that are called by multiple parts of the program. We proceed as follows.

(a) We introduce a "level counter" that is initialized to zero and incremented every time a function is called. The counter is decremented each time a function returns to the calling function. Functions with multiple levels of invocation can immediately be excluded for our consideration.

(b) Functions with very small execution times need not be considered separately. These times (even with multiple calls) can typically be rolled into the lines of their parent routines.

(c) With these two filters we identified the functions shown in Table 3.1 that need separate consideration.

(d) For each routine in Table 3.1, we modified the SIFT code to record the starting clock time and ending clock time. The time taken by each routine is calculated by finding a difference of these times.

(e) Of the routines in Table 3.1, `create_init_img` is excluded because it is a preprocessing stage that converts a color image to a gray scale image in PNG format. In our experiments we start with such an image. Routine `release_pyr` which related to displaying the output on the screen is also excluded. Of the remaining routines `calc_feature_scales` has a much smaller execution time than the others and

(a) 1 (b) 2 (c) 3 (d) 4 (e) 5

(f) 6 (g) 7 (h) 8 (i) 9 (j) 10

(k) 11 (l) 12 (m) 13 (n) 14 (o) 15

(p) 16 (q) 17 (r) 18 (s) 19 (t) 20

(u) 21 (v) 22 (w) 23 (x) 24 (y) 25

(z) 26

Figure 3.1: Pictures considered

Table 3.1: The time values in the table corresponds to the average values of Image 9 (Gandhi 4) over its all resolutions.

Routine Name	Level	Time	Description
create_init_img	2	0.116	Convert color to grayscale image
build_gauss_pyr	2	0.871	Create scales (see Section 2.2 Page 6)
build_dog_pyr	2	0.180	Difference of Gaussians (see Page 9)
scale_space_extrema	2	1.571	Extrema and Keypoint detection (see Section 2.3)
calc_features_scales	2	0.016	part of Orientation assignment
calc_features_oris	2	2.204	Orientation assignment (see Section 2.4)
compute_ descriptors	2	11.341	Computing descriptors (see Section 2.5)
release_pyr	2	0.008	Related to Display
_sift_features	1	16.603	Sift Routine
Main Routine	0	16.646	start of main

functionally can be rolled together with `calc_feature_oris`. Therefore we use only `calc_feature_oris` with the understanding that this also includes `calc_feature_scales`.

We now have the following routines that have a significant time contribution in the overall time of the program. The mnemonics in parenthesis are used in graphics to identify these routines.

1. build_gauss_pyr (Gauss) [Gaussian blurring]
2. build_dog_pyr (DoG) [Difference of gaussians]
3. scale_space_extrema (SSext) [Scale space extrema and keypoint detection]
4. calc_features_oris (Orien) [Orientation assignment]
5. compute_ descriptors (Descr) [Keypoint descriptor generation]

The above stages are in close correspondence with the main phases of the SIFT algorithm described in Lowe [13].

We now examine the behavior of these major phases. Figure 3.2 shows the percentage of the total time taken by the major phases of the SIFT algorithm. The Gaussian phase and Computing Descriptor phase need most of the time (around 70%). Figures 3.3, 3.4 show the normalized and absolute times taken by the different phases of the SIFT algorithm across all the images. The absolute values are averaged over their resolutions. The Gaussian blurring phase and Keypoint descriptor generation phase still take a large amount of time (around

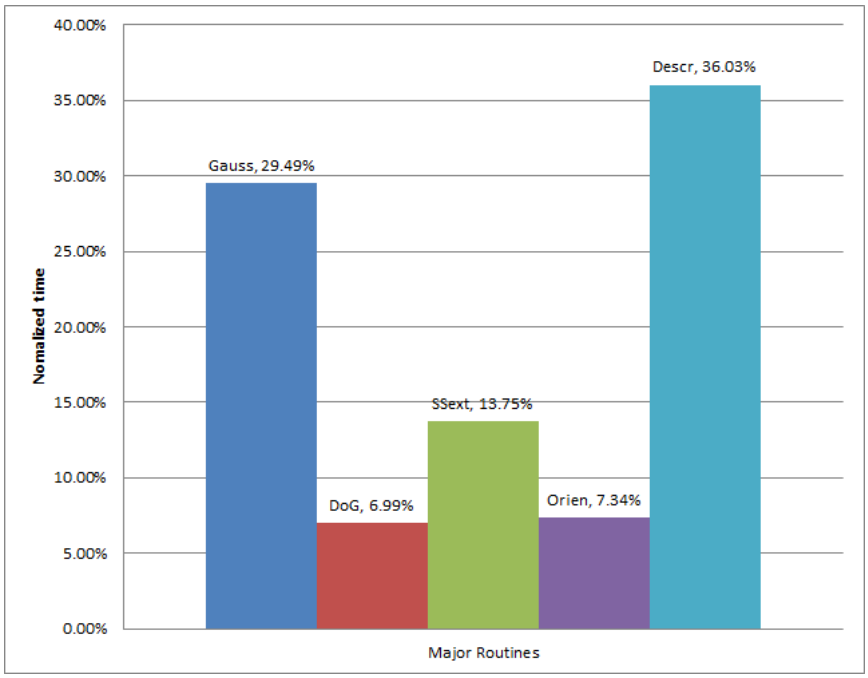


Figure 3.2: Percentage time for major stages of SIFT (averaged over all images)

70%). However there is a large variation in the time taken by Gaussian blurring vis-a-vis the Computing Descriptor phase. This variation also does not seem to be correlated with picture categories.

In this thesis we consider that the Gaussian blurring phase and Keypoint descriptor generation phase are the most time consuming stages (as bases for decoupling the algorithm across chips). These stages also exhibit high parallelism to facilitate efficient execution on multicore chips. In fact for the two chip pipeline mode we assume that Chip 1 consists of phases Gaussian blurring to Keypoint Detection, and Chip2 consists of the Orientation Assignment and keypoint descriptor generation phases.

We now examine how individual phases perform across different pictures.

Figure 3.5 shows the time taken by the Gaussian stage for all 2000×2000 images (highest resolution), 1000×1000 images (intermediate resolution) and 100×100 images (lowest resolution). The average time taken by the Gaussian stage across these 20 resolutions is also shown. Notice the strong correlation between times taken for different resolutions. They all indicate a similar pattern. Thus the average over all resolutions of a fixed image is a reasonable indicator for that picture. We will henceforth consider only the average of

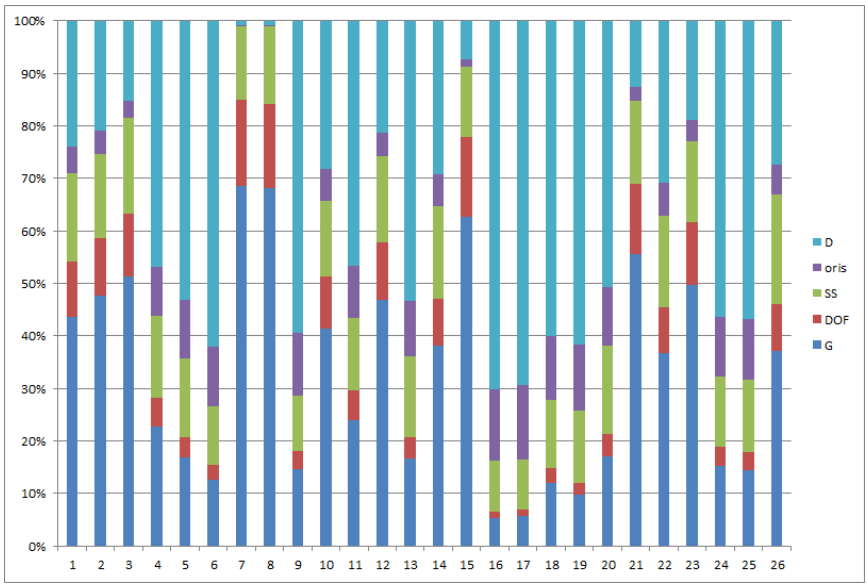


Figure 3.3: Percentage time of SIFT major phases over different pictures (averaged over all resolutions)

all image resolutions of each picture.

As Figure 3.5 shows the amount of time taken by the Gaussian blurring stage across all images is almost constant. This is because, all the images have same number of pixels and no distinguishing features have been detected yet. The difference of Gaussians stage also shows the similar behavior (see Figure 3.6).

Figure 3.7 shows the Scale-space extrema and keypoint detection phase, Orientation assignment phase and keypoint descriptor generation phase times across pictures. While these times vary largely across pictures, these are correlated within images of a pictures. This is because the times for these stages depend on the number of features in the image.

In this thesis we will consider 1- and 2-chip pipelines for running SIFT (see Chapters 7, 8). As noted earlier for the 2-chip case we will broadly divide the algorithm between the Gaussian blurring phase and keypoint descriptor generation phase. The Gaussian blurring phase takes nearly constant time across pictures, while the keypoint descriptor generation phase does not. To even out these variations, we place the difference of Gaussians phase and scale-space extrema and keypoint detection phases with the Gaussian blurring phase and Orientation assignment phase with the keypoint descriptor generation phase. If additional chips are to be used in the pipeline (future work) then a similar approach can be used to further subdivide the Gaussian blurring phase and keypoint descriptor generation phase.

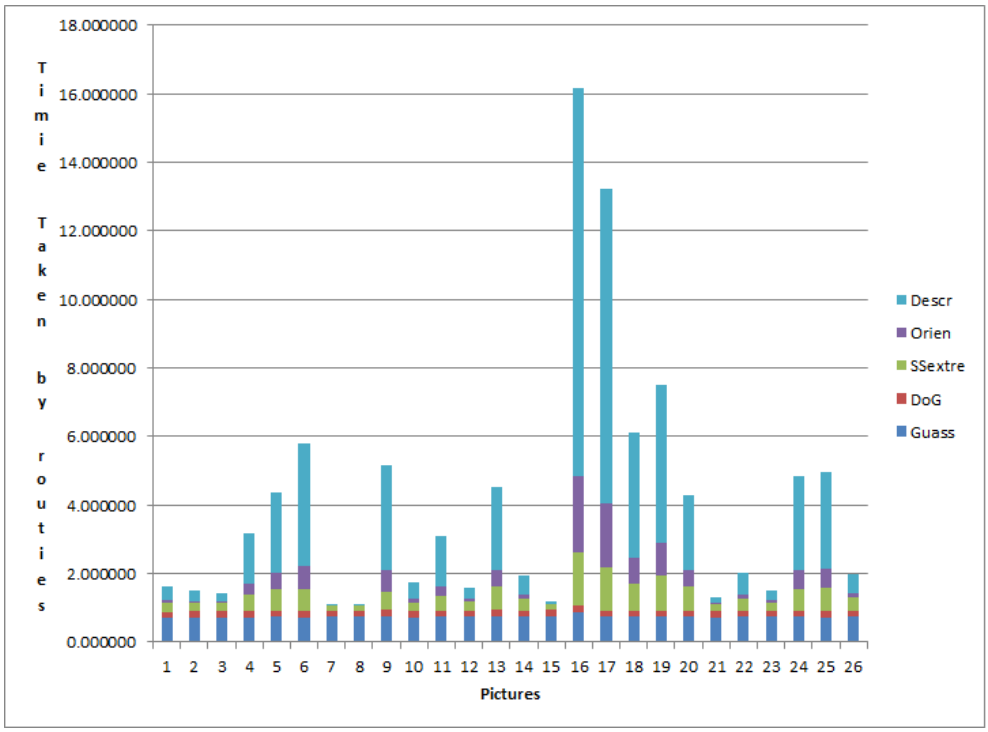


Figure 3.4: The absolute times of major SIFT phases (averaged over all image resolutions of a picture)

3.3 Feature Fractions

Figure 3.8 shows, for each resolutions size, the number of extrema detected, the number of keypoints and the number of features, averaged over all pictures. Clearly, the number of extrema is more than than the number of keypoints, which, in turn, is slightly less than the number of features. This brings us to the idea of feature fractions of an image (see Section 2.1).

To recap, an N^2 pixel image produces αN^2 extrema, $\alpha\beta N^2$ keypoints and $(\alpha\beta + \gamma)N^2$ features. Figures 3.9, 3.10, 3.11 show the values of α, β and γ for Picture number 26 (all 20 image resolutions of the picture). In these graphs the number 1-20 on the X-axis represents the 20 resolutions with 1 as the 2000×2000 image (largest resolution) and 20 as the 100×100 image (smallest resolution). That is, a smaller picture number represents a larger resolution. Notice that α, β and γ are nearly constant for large resolutions[1] of the

[1]We give more importance to larger images because parallel SIFT is more useful for large images that require higher speed to keep up with real time constraints

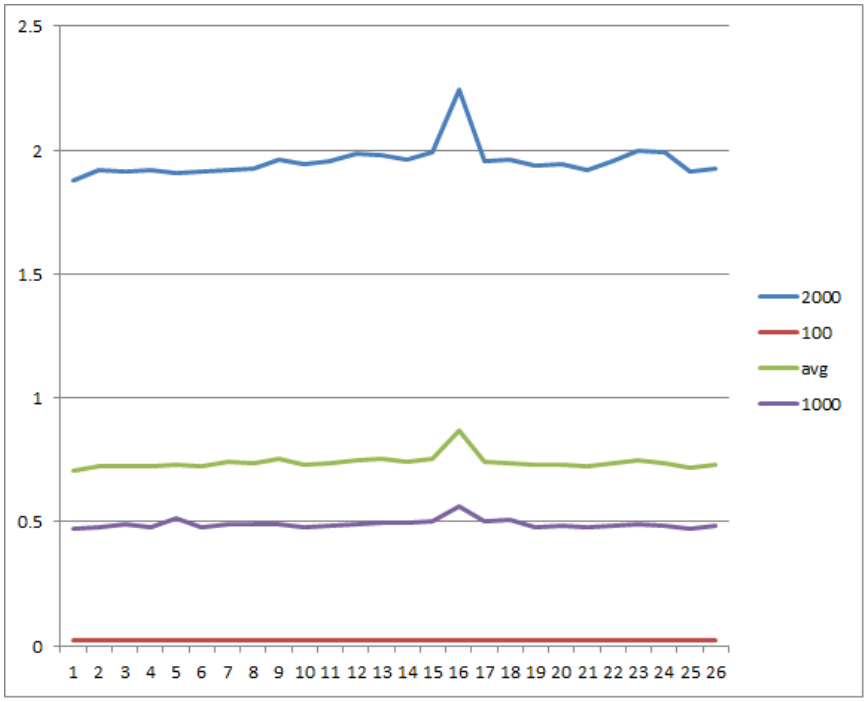


Figure 3.5: The time taken by the Gaussian stage for 2000×2000 (the highest resolution), 1000×1000 (intermediate resolution) and 100×100 (lowest resolution) and the average value over all these resolutions

pictures (left end of the X-axis). Also α, β and γ are all small fractions. These observations hold in general over all these pictures. Figures 3.12, 3.13, 3.14 show the values of α, β and γ for Pictures 1–25. These graphs are the same as there in Figures 3.9, 3.10, 3.11 expect that the axes are not labeled. Figures 3.15, 3.16, 3.17 show the average values of α, β and γ.

From these will assume nominal values of α, β and γ around $0.6\%, 35\%$ and 0.04% (see Figures 3.15, 3.16, 3.17)for every image considered in this work. In the pictures considered, Picture 16 did not resemble with the remaining pictures, so we took it out from the determination of feature fraction values.

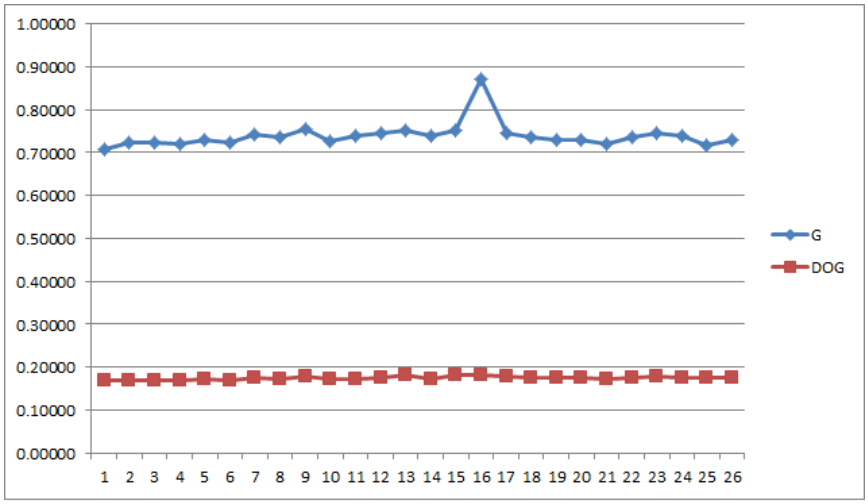


Figure 3.6: Average time for Gaussian blurring and difference of Gaussian phases

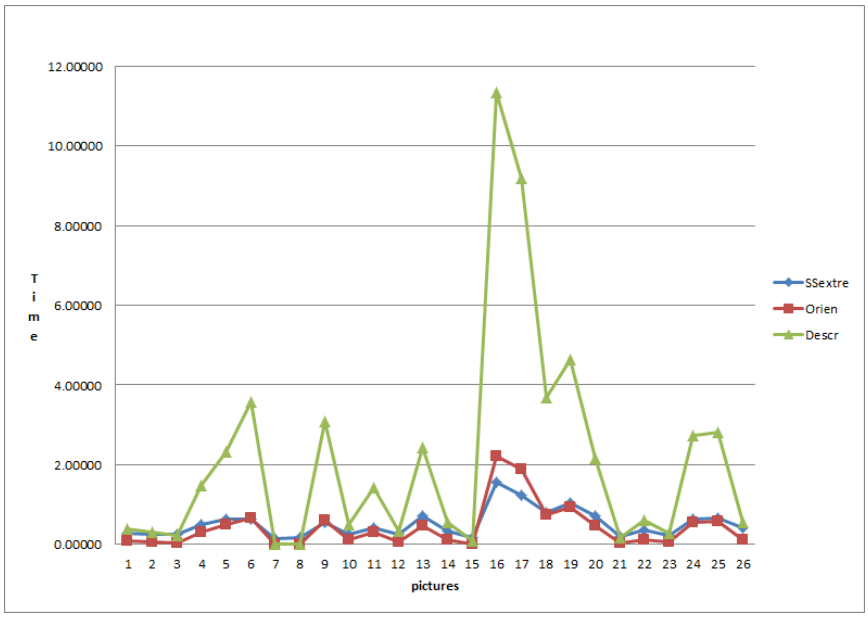


Figure 3.7: Times taken for scale-space extrema detection, orientation assignment and keypoint descriptor generation phases

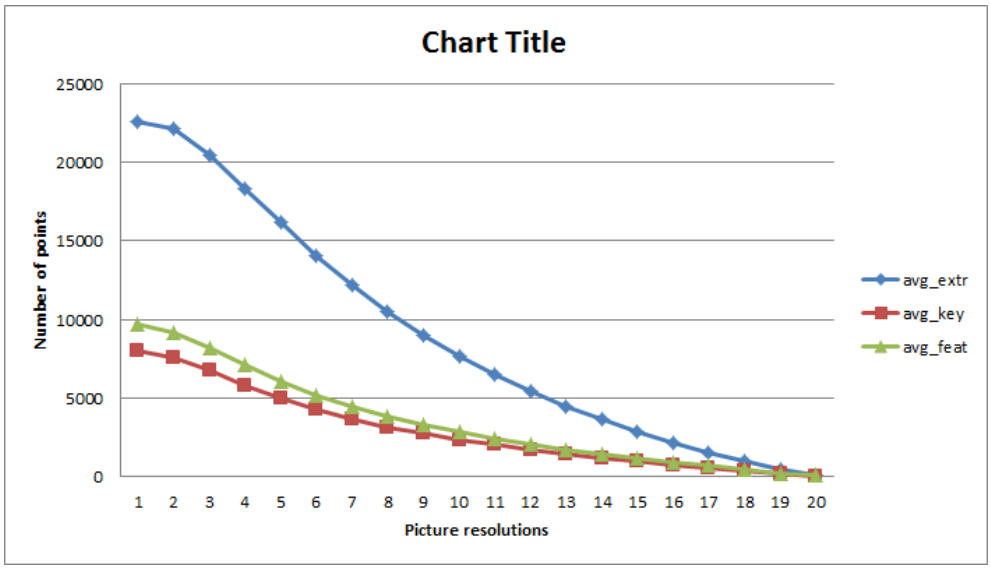


Figure 3.8: Nominal number of extrema, keypoints and features

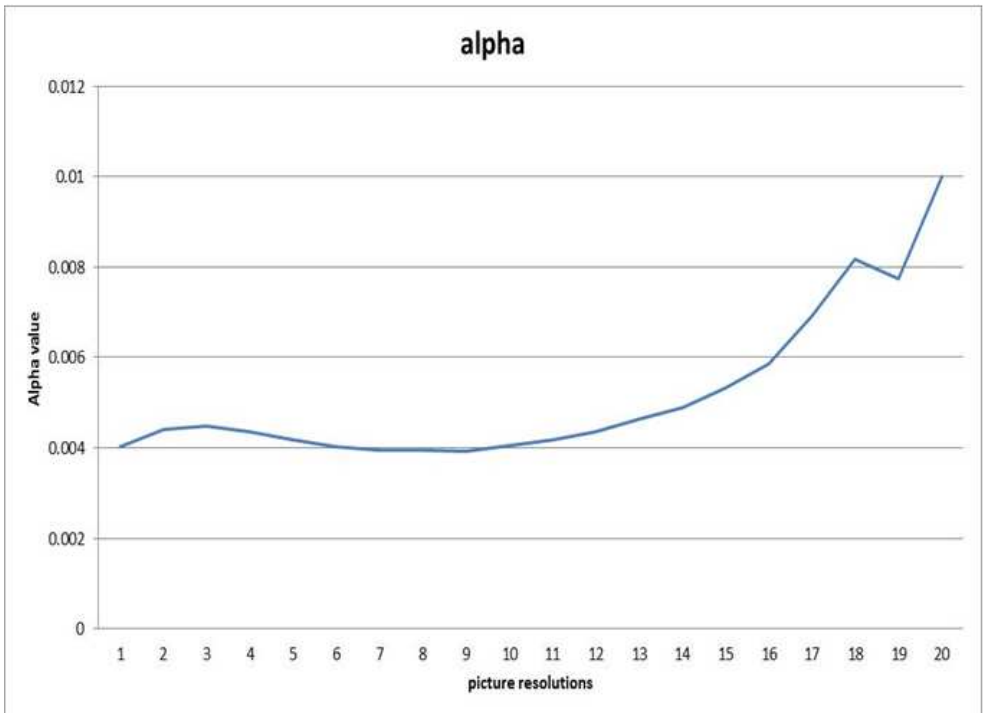


Figure 3.9: The value of α for all the image resolutions of picture number 26

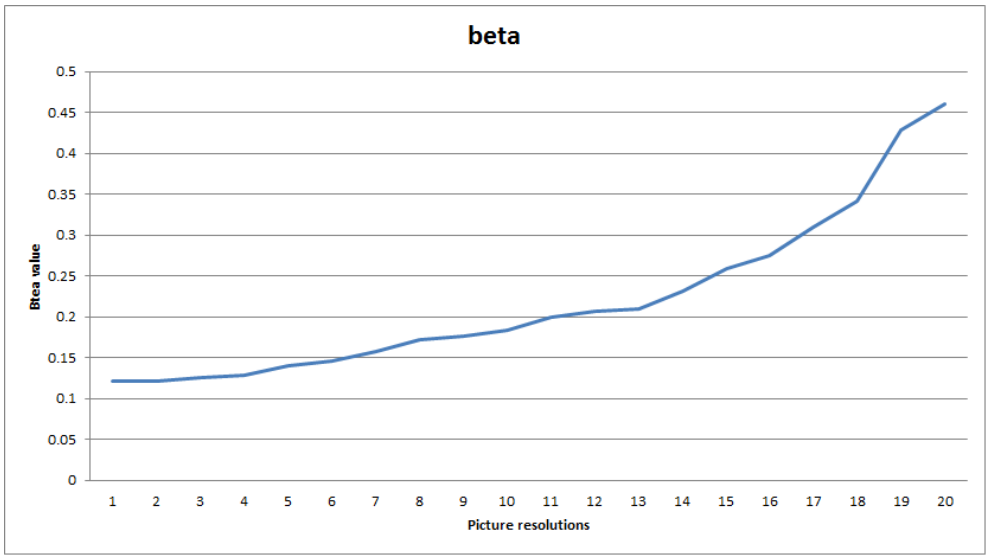


Figure 3.10: The value of β for all the image resolutions of picture number 26

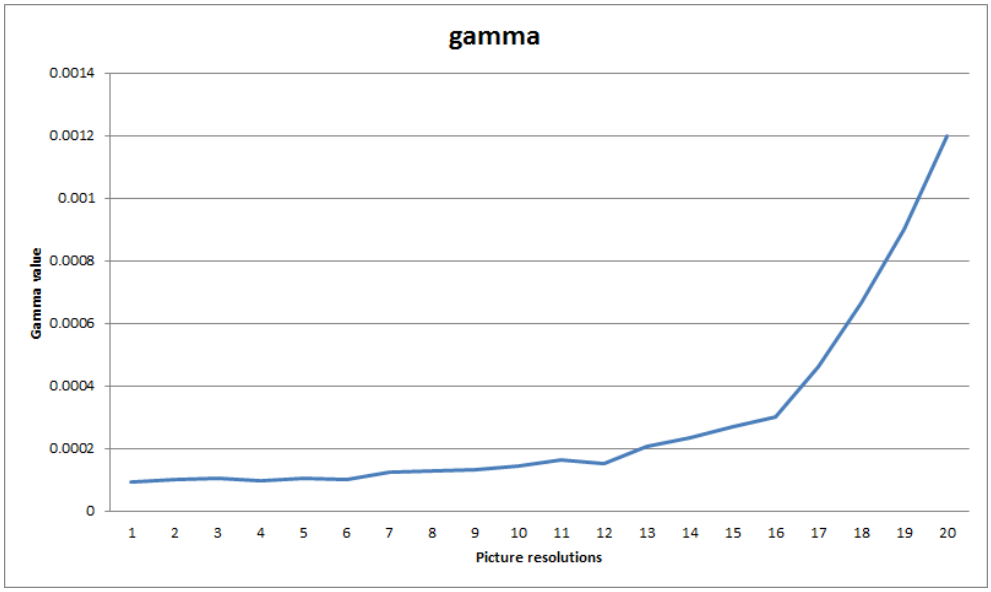
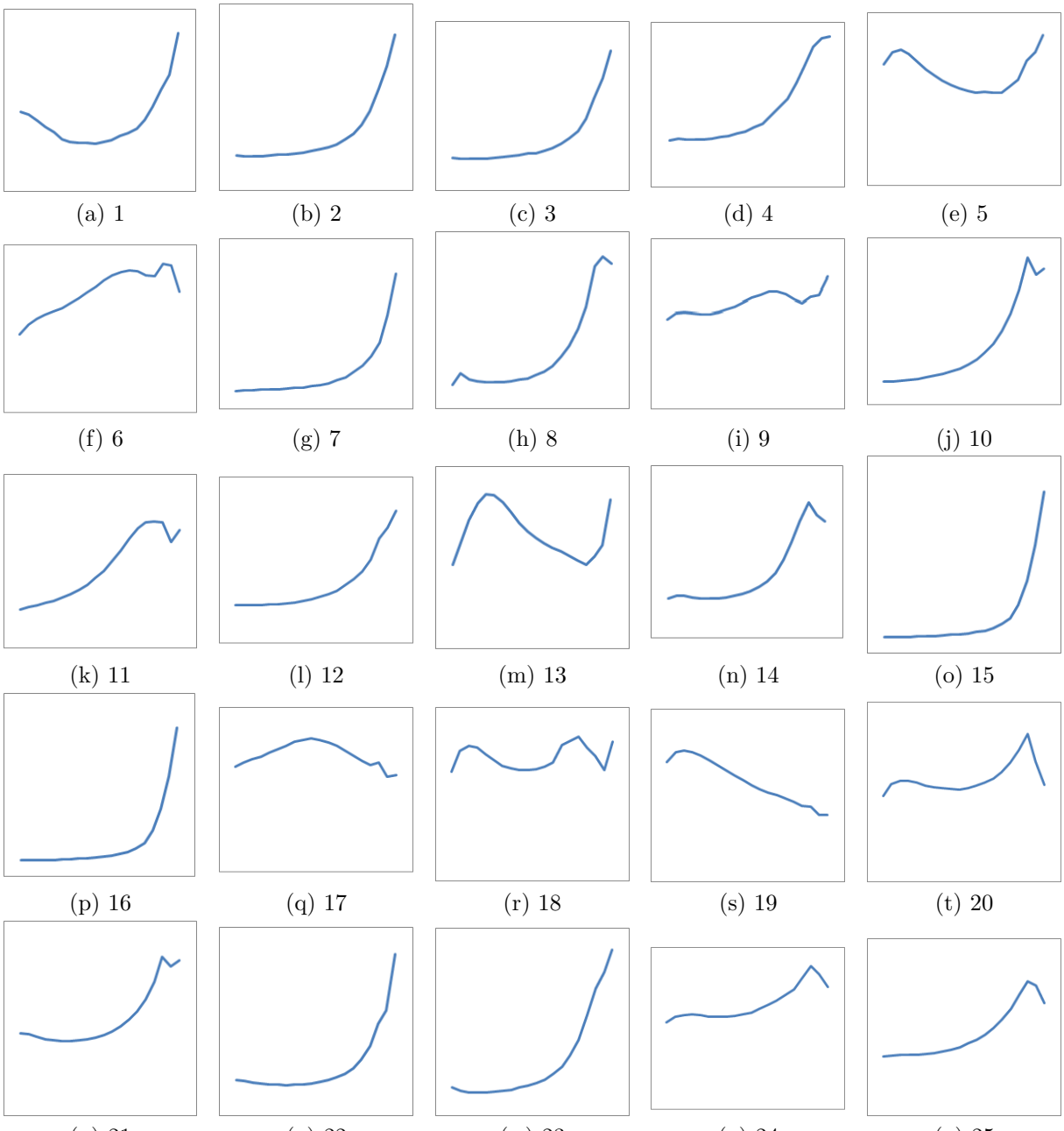


Figure 3.11: The value of γ for all the image resolutions of picture number 26



(a) 1 (b) 2 (c) 3 (d) 4 (e) 5

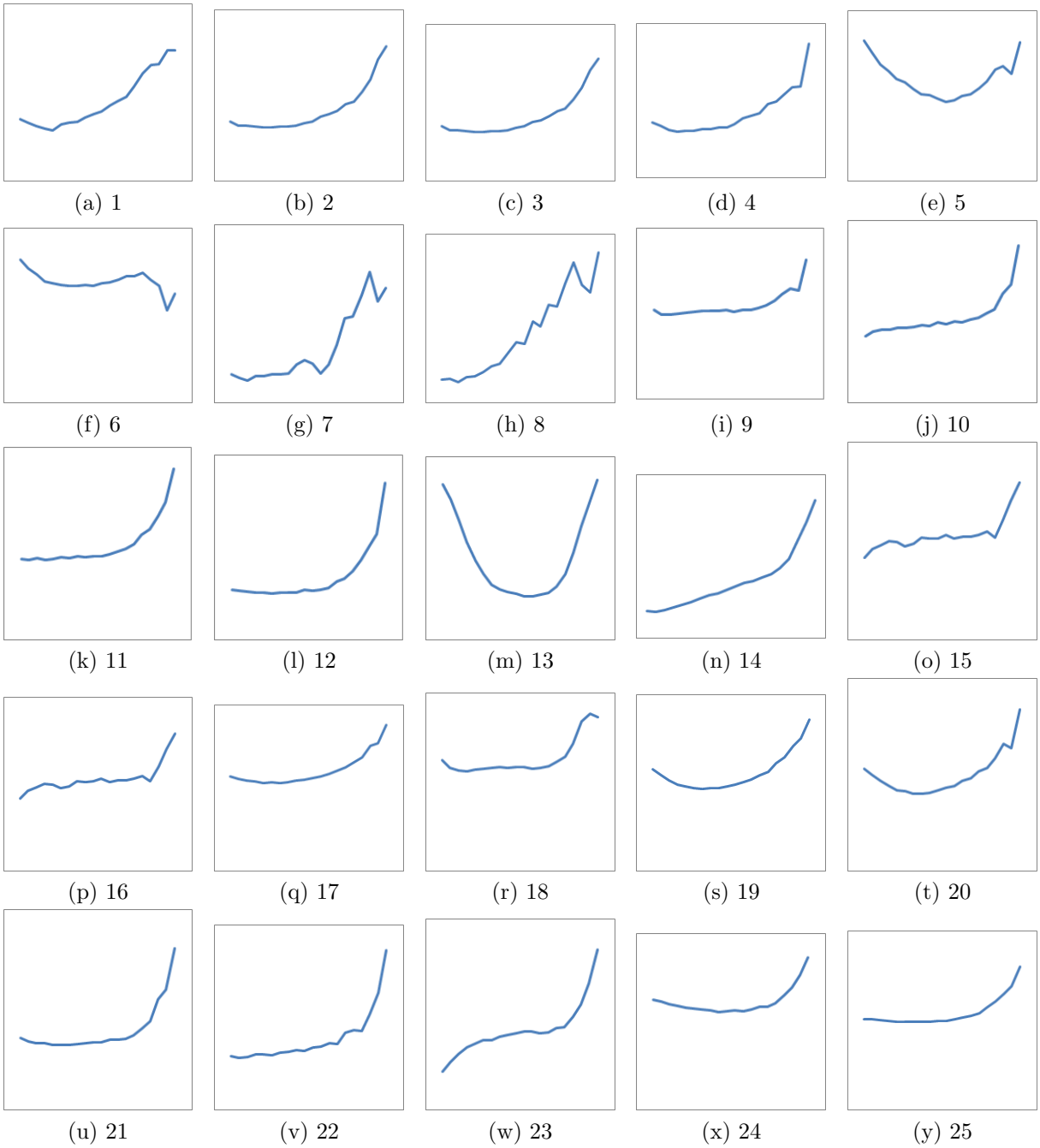
(f) 6 (g) 7 (h) 8 (i) 9 (j) 10

(k) 11 (l) 12 (m) 13 (n) 14 (o) 15

(p) 16 (q) 17 (r) 18 (s) 19 (t) 20

(u) 21 (v) 22 (w) 23 (x) 24 (y) 25

Figure 3.12: The value of α for all the image resolutions of pictures numbered 1-25



(a) 1 (b) 2 (c) 3 (d) 4 (e) 5

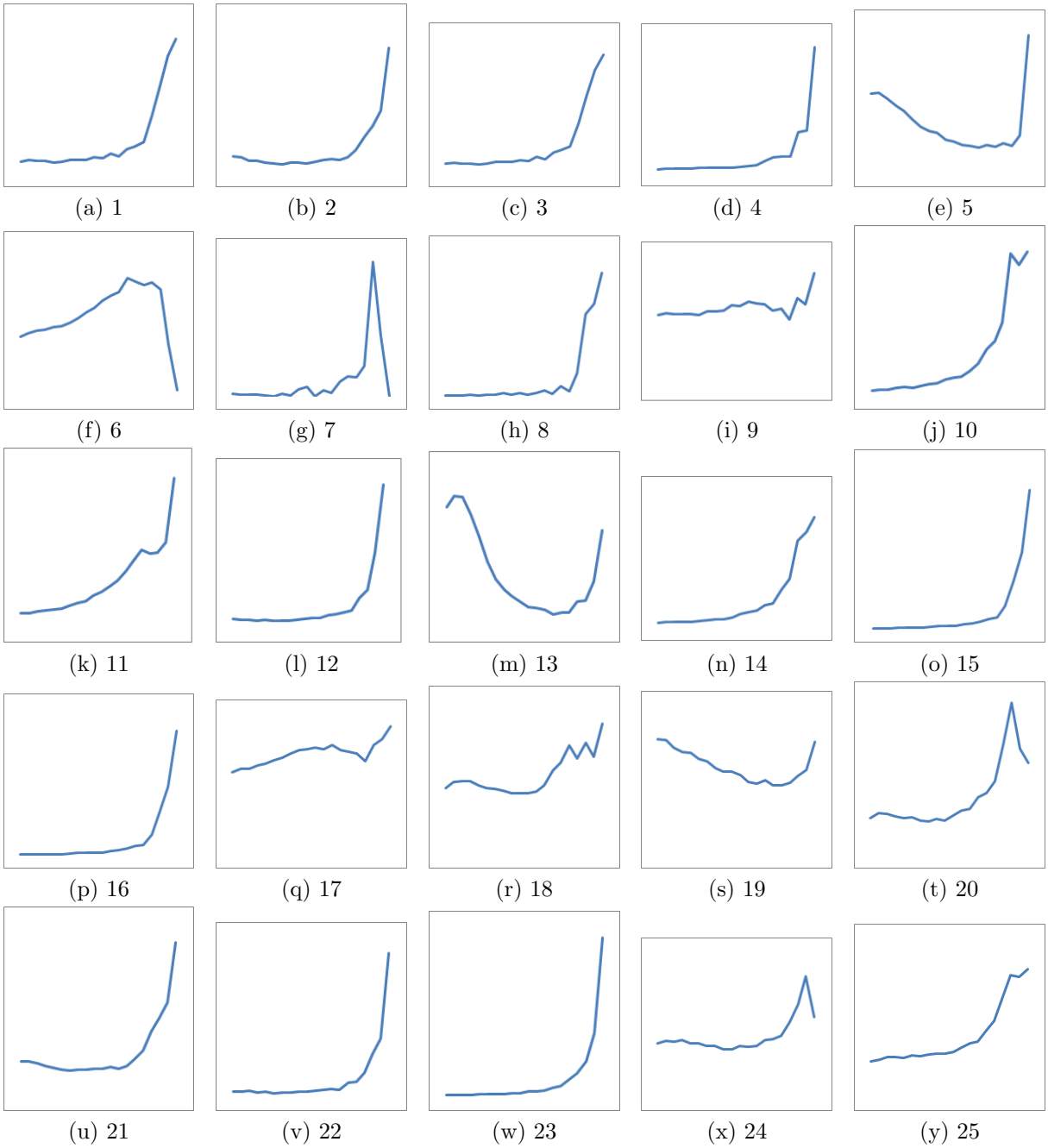
(f) 6 (g) 7 (h) 8 (i) 9 (j) 10

(k) 11 (l) 12 (m) 13 (n) 14 (o) 15

(p) 16 (q) 17 (r) 18 (s) 19 (t) 20

(u) 21 (v) 22 (w) 23 (x) 24 (y) 25

Figure 3.13: The value of β for all the image resolutions of pictures numbered 1-25



(a) 1 (b) 2 (c) 3 (d) 4 (e) 5

(f) 6 (g) 7 (h) 8 (i) 9 (j) 10

(k) 11 (l) 12 (m) 13 (n) 14 (o) 15

(p) 16 (q) 17 (r) 18 (s) 19 (t) 20

(u) 21 (v) 22 (w) 23 (x) 24 (y) 25

Figure 3.14: The value of γ for all the image resolutions of pictures numbered 1-25

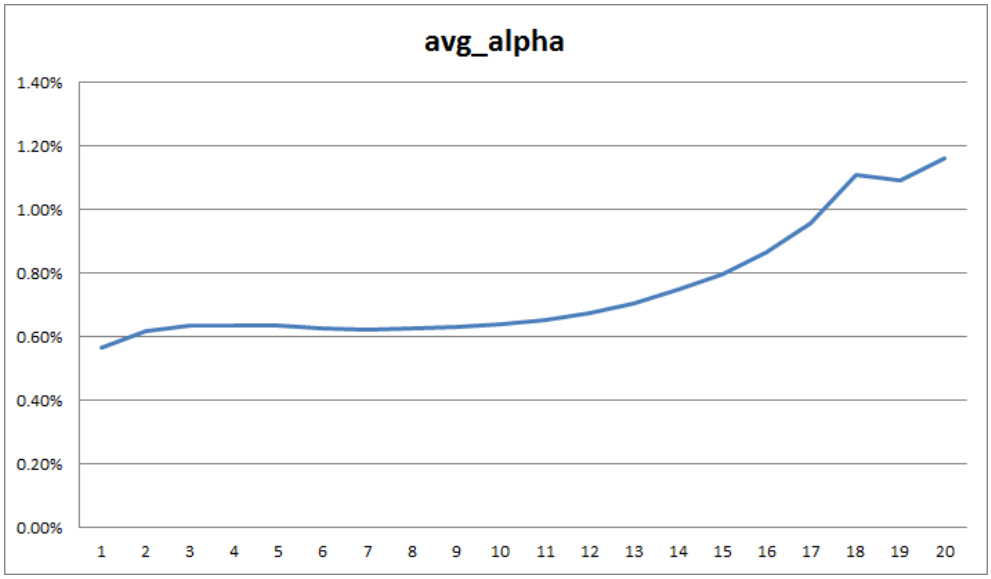


Figure 3.15: The value of α across all the images averaged over their resolutions

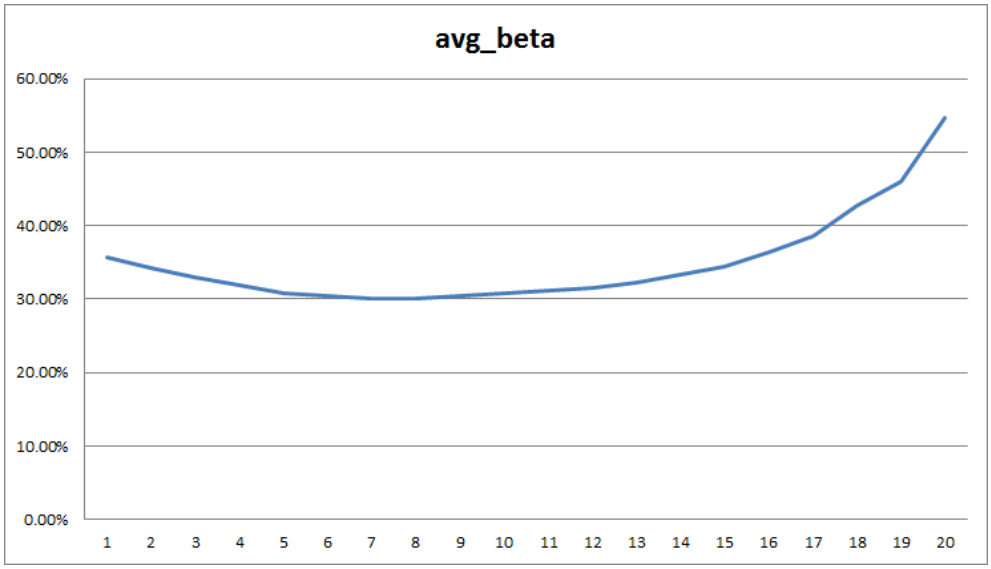


Figure 3.16: The value of β across all the images averaged over their resolutions

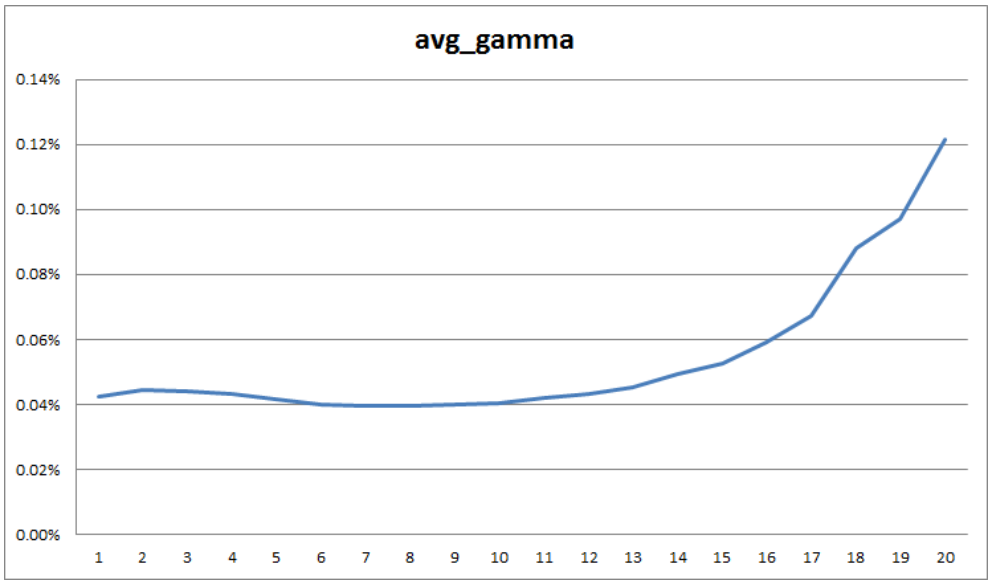


Figure 3.17: The value of γ across all the images averaged over their resolutions

Chapter 4

Tile Ordering

As mentioned earlier, we assume the SIFT algorithm to process (n^2)-pixel tiles of the image in each iteration. We can process the given image of size $N \times N$ in one iteration (if $n = N$) or at the other extreme, we can process one pixel at a time (if $n = 1$). In general $1 \gg n \gg N$. The given image is decomposed into subimages each with n^2 pixels. These subimages are called as *tiles*. For a given image of size $N \times N$, there will be a total of $\xi^2 = \frac{N^2}{n^2}$ tiles. These ξ^2 tiles are provided as input to the computation pipeline (discussed in Chapter 6) one at a time to Stage S_0 of the pipeline.

The order in which the image tiles are sent as an input to the pipeline is called *tile ordering*. This tile ordering plays a major role on the performance of the algorithm where there are restrictions on the input format. In this chapter, we introduce two tile orderings used in the later chapters to analyze the performance of the SIFT algorithm namely *Row Major Ordering* and *Diagonal Ordering*.

The remainder of this chapter is organized as follows. In the Section 4.1, we describe the notation for a tile. In Sections 4.2, 4.3, we describe the two tile orderings.

4.1 Tile Notation

In this section we define some notation for a tile and its numbering in a tile ordering. Each tile is an $n \times n$ square array of n^2 distinct pixels from the given image. For a given $N \times N$ image, there will be $\xi = \frac{N}{n}$ rows and columns of tiles. We refer to this $\xi \times \xi$ array of tiles as *tile array*. The tiles in this array are represented with two co-ordinates (r, c) denoting

the row r and column c in the tile array. Figure 4.1 shows the tiles coordinates in 5 $\times$ 5 tile array. For $0 \leq i < \xi$, let $\tau_{i,j}$ represents the tile at position (r, c). Let $\circledcirc$ be the given ordering where $\circledcirc \in \{R, D\}$ (denoting row major and diagonal ordering) then the k^{th} tile in this ordering is denoted by $\tau_k^{\circledcirc}$. If this happens to be the tile in position (r, c) then $\tau_{r,c}^{\circledcirc}$. This quantity k associated with tile $\tau_{r,c}$ with respect to ordering $\circledcirc$ is called the rank of $\tau_{r,c}$, where the ordering $\circledcirc$ is clear or unimportant, we will write $\tau_{r,c}^{\circledcirc}$ or $\tau_k^{\circledcirc}$, simply as $\tau_{r,c}$ or τ_k.

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)

Figure 4.1: Coordinate representation of tiles in a tile array

4.2 Row Major Ordering

As mentioned earlier, this ordering deals with square tiles of size $n \times n$. Here the order of the tiles is by rows and the direction is from left to right within a row. Figure 4.2 shows this order for a 5 $\times$ 5 tile array.

Lemma 1. For any $0 \leq r, c < \xi$, the rank of tile $\tau_{i,j}$ in Row major order is $rank(\tau_{r,c})^{RM} = r\xi + c$

Proof: Any given tile $\tau_{r,c}$ can only be enumerated after all tiles before it are enumerated. The number of tiles enumerated before Tile $\tau_{r,c}$ is $r\xi$ before row r and c in the row r. Thus $\tau_k = r\xi + c$
Note: Here $\tau_{r,c} = \tau_k^R$ where $k = rank(\tau_{r,c})^R = r\xi + c$ ■

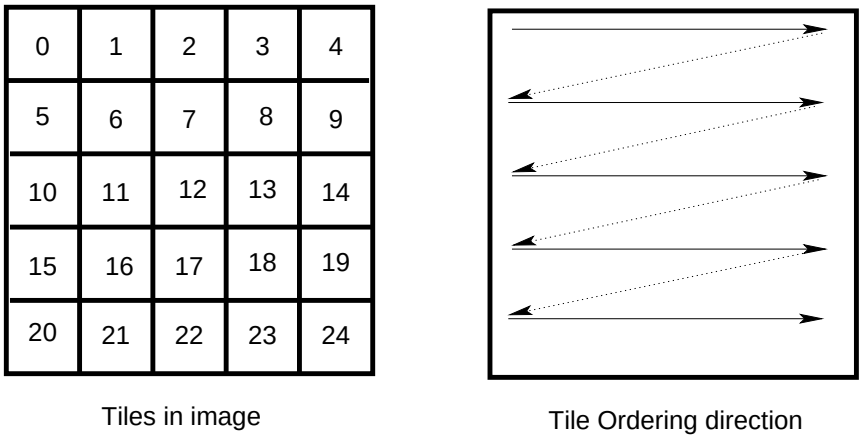


Figure 4.2: Row-major ordering

4.3 Diagonal Ordering

Here the order of tiles is by the value of $r + c$. Consider two tiles τ_{r_1,c_1} and τ_{r_2,c_2}. If $(r_1+c_1) < (r_2+c_2)$ then τ_{r_1,c_1} is enumerated before τ_{r_2,c_2}. If $(r_1+c_1) = (r_2+c_2)$, then τ_{r_1,c_1} is enumerated before τ_{r_2,c_2} if and only if $r_1 < r_2$. Figure 4.3 shows this order for a 5×5 tile array. The proof of following lemma can be proved in a manner similar to Lemma 1. Its proof is omitted for brevity.

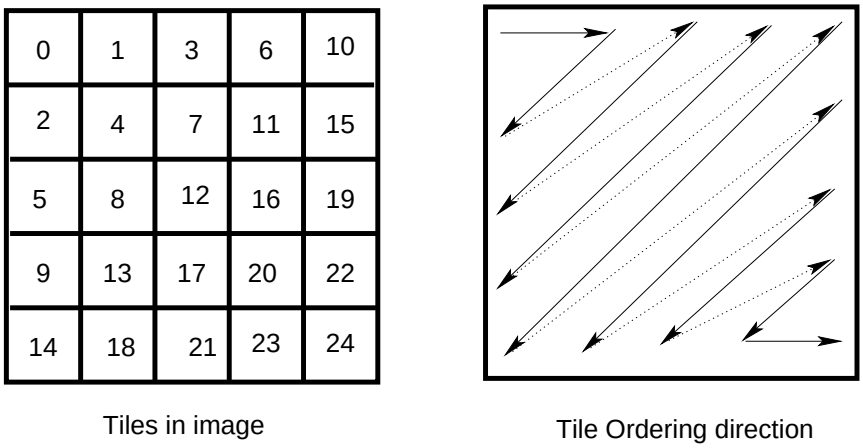


Figure 4.3: Diagonal ordering

Lemma 2. For any $0 \leq r < \xi$, the rank of the tile $\tau_{i,j}$ in Diagonal ordering is to

$$rank(\tau_{r,c})^D = \begin{cases} \frac{(r+c)(r+c+1)}{2} + r & \text{if } r + c < \xi; \\ (\xi^2 - 1) - \left(\frac{(r'+c')(r'+c'+1)}{2} + r'\right) & \text{if } r + c \geq \xi \end{cases}$$

where $r' = \xi - c$ and $c' = \xi - c$

■

Chapter 5

The Computation Pipeline

The computational platform on which we study SIFT is a pipeline of chips. The fact that the SIFT algorithm consists of sequential stages and that many real-time applications of SIFT stream in data make a pipeline a suitable model for SIFT. Each chip in the model has three internal stages, each stage feeding its partial result to the next. So the computational model can be viewed as a pipeline of stages, rather than a pipeline of chips. Let $C = \{C_0, C_1, C_2, \cdots, C_{c-1}\}$ be a set of c chips (see Figure 5.1). For each $1 \leq i < c-1$ Chip C_i receives its input from the Chip C_{i-1} and sends its output to the Chip C_{i+1}. Chip C_0 receives the input of the algorithm and Chip C_{c-1} produces the algorithm's final output (see Figure 5.1).

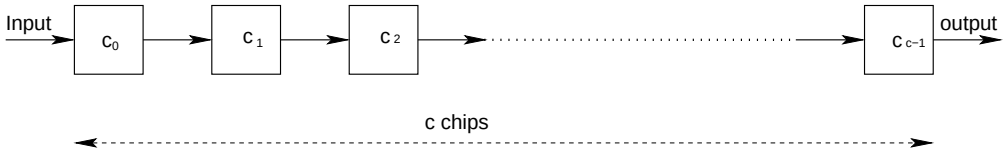


Figure 5.1: A c-chip pipeline

As noted above, each chip is assumed to consist of three internal stages, *input*, *compute* and *output*, that can work simultaneously (see Figure 5.2). The c-chip model is a $(2c+1)$-stage pipeline. The output stage of Chip C_i must be compatible with the input stage of Chip C_{i+1} in terms of bandwidth, pins, etc. We will consider these two stages (namely

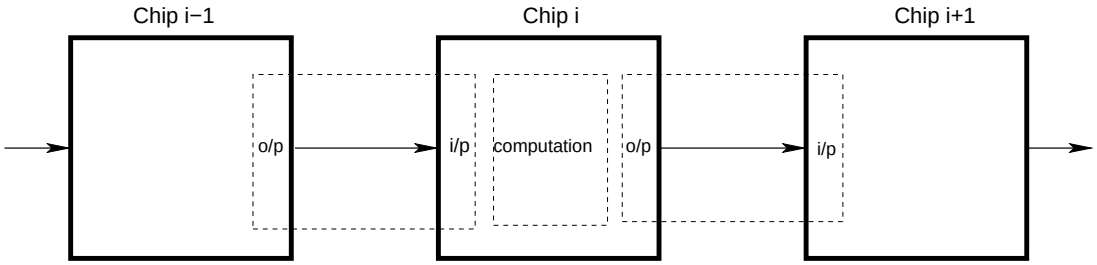


Figure 5.2: A chip in the pipeline model

output stage of Chip C_i and input stage of Chip C_{i+1}) to be identical in the pipeline. To keep the notation clear and consistent on this $(2c+1)$ stage pipeline, we will denote these stages as S_0, S_1, S_2, $\cdots$, S_{2c} where S_{2i} denotes the input stage of Chip C_i and the output of stage of Chip C_{i-1} (if it exists) and S_{2i+1} denotes the compute stage of Chip C_i (see Figure 5.3).

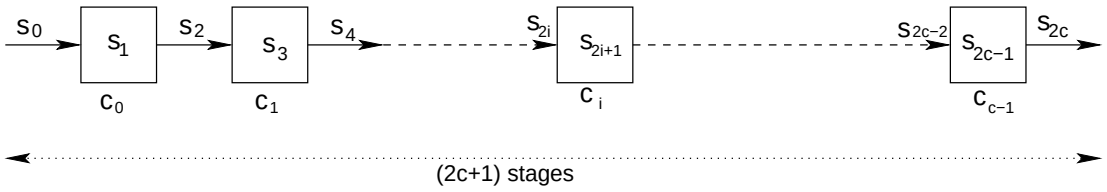


Figure 5.3: Stages in the pipeline model

The input stage of Chip C_i receives the input from the output stage of Chip C_{i-1} and holds this in local memory until the compute stage of Chip C_i requires it. The output stage of Chip C_i receives the results of the computation from the compute stage.

Now we describe how the SIFT algorithm runs on the c-chip pipeline. Recall that the given image of size $N \times N$ is decomposed into (n^2)-pixel tiles. These tiles are fed into the pipeline in some order as described in Chapter 4. For a given tile order, $\circledcirc$, let $\tau_k^{\circledcirc}$ (or simply τ_k) be the k^{th} tile input to the pipeline.

Let us trace the traversal of the tiles through the pipeline in this ordering. For ease of explanation, assume each tile traverses each stage in 1 unit of time, starting from $T = 1$. At time $t = 1$ the first tile τ_k enters Chip C_0 through Stage S_0. Additional information necessary to process the tile also enters stage S_0. For reasons that will apparent later, let this tile entering be denoted by $\tau_{k,0}$. At time $t = 2$, Stage S_1 receives τ_k from S_0 and processes it. We call this input to Stage S_1 as $\tau_{k,1}$. After processing it, it produces an output $\tau_{k,2}$ that Stage S_2 receives at time $t = 3$. At time $t = 2i$, Stage S_{2i}(an input/output stage) receives tile $\tau_{k,2i}$. Note that $\tau_{k,2i}$ may be just an extract of the original image tile τ_k.

5.1 Notation

In this section we formalize the notation used earlier to describe "tiles" at each stage or chip in the pipeline model. A given tile τ_k goes through the $(2c + 1)$-stage pipeline. The notation $\tau_{k,\ell}$ represents the data input to stage S_ℓ of the pipeline. Tile $\tau_{k,\ell+1}$ represents the result of the stage S_ℓ which is fed to Stage $S_{\ell+1}$ (if it exists) and so on. For tile τ_k, here $\tau_{k,0}$ is the image input to the pipeline at stage S_0 and $\tau_{k,2c}$ is the final output of the pipeline at the stage S_{2c} for tile τ_k.

Consider any chip C_i, with internal stages S_{2i}, S_{2i+1}, S_{2i+2}. Here the stages S_{2i} and S_{2i+2} are communication stages (input and output stages) and S_{2i+1} is the compute stage. Stage S_{2i} brings in all the data needed for S_{2i+1} to start its computation on any given tile $\tau_{k,2i+1}$ and S_{2i+2} carries out all the data produced by S_{2i+1}. The data needed to perform the computation on tile $\tau_{k,2i+1}$ at Stage S_{2i+1} is not necessarily the same as $\tau_{k,2i+1}$ and may have been brought in entirely or partially at an earlier time. We formalize this notion below.

To process tile τ_k, the compute stage S_{2i+1} of chip C_i requires some additional data related to the tile $\tau_{k,2i+1}$. These bits have to be present in the chip C_i before stage S_{2i+1} can process $\tau_{k,2i+1}$. However the bits could have been brought in earlier to Chip C_i (wholly or in part). The time when these bits arrive at Chip C_i is decided by the input protocol which is discussed in Section 5.3.

5.1.1 Stage Start Times

In this section we derive an expression for the earliest starting times for each stage S_i in processing tile τ_k. This allows us to derive an expression for the overall time of the algorithm. For any stage S_ℓ, $(0 \leq \ell < 2c)$ and any tile τ_k, let $T_{k,\ell}$ be the earliest time when

stage S_ℓ can work on $\tau_{k,\ell}$. Let $t_{k,\ell}$ denote the time needed for stage S_ℓ to perform its action on tile $\tau_{k,\ell}$.

The time for stage S_ℓ to complete processing tile $\tau_{k,\ell}$ depends on two quantities.

1. The time for stage S_ℓ to finish processing the previous tile τ_{k-1}.
2. The time for stage $S_{\ell-1}$ to complete processing the current tile τ_k

The following theorem captures the above observation.

Theorem 3. If ℓ is any stage in the pipeline where $0 \leq \ell < 2c$, then the earliest time when stage S_ℓ can start on tile τ_k is

$$T_{k,\ell} = \max\{T_{k,\ell-1} + t_{k,\ell-1}, \quad T_{k-1,\ell} + t_{k-1,\ell}\}$$

■

The following lemma representing a well known standard result finds use later in this chapter.

Lemma 4. The recurrence, $a_0 = b_0$ and $a_n = a_{n-1} + b_n$, for $n > 0$ has the solution $a_n = b_0 + b_1 + b_2 + \cdots + b_n$.

Proof: We proceed by induction on n. For $n = 0$, $a_0 = b_0$. Assuming the lemma to hold for $n = n + 1$. consider a_{n+1}. $a_{n+1} = a_n + b_{n+1} = (b_0 + b_1 + \cdots + b_n) + b_{n+1}$. ■

Lemma 5. For $0 < k < \xi^2$, the earliest time when Stage S_0 can start on tile τ_k is $T_{k,0} = \sum_{u=0}^{k-1} t_{u,0}$

Proof: Since there is no stage before stage S_0, it does not have to wait on any other stages to receive the data of tile. Therefore from Theorem 3, $T_{k,0} = T_{k-1,0} + t_{k-1,0}$. With $T_{0,0} = 0$ (the count of time begins here), this recurrence has the solution $T_{k,0} = \sum_{u=0}^{k-1} t_{u,0}$ (from Lemma 4). ■

Lemma 6. For any $0 \leq \ell < 2c$, the earliest time when stage S_ℓ can start on tile τ_0 is $T_{0,\ell} = \sum_{v=0}^{\ell-1} t_{0,v}$

Proof: Since there is no tile before tile τ_0, a stage S_ℓ is free when $\tau_{0,\ell}$ arrives at S_ℓ. Then from Theorem 3, $T_{0,\ell} = T_{0,\ell-1} + t_{0,\ell-1}$. With $T_{0,0} = 0$ (the count of time begins here), this recurrence has the solution $T_{0,\ell} = \sum_{v=0}^{\ell-1} t_{0,v}$ (from Lemma 4). ■

We now discuss the total time needed to process all tiles.

Total Time: Recall that the given image of size $N \times N$ is decomposed into ξ rows and ξ columns of $n \times n$ tiles. The total time to process a given image is equal to the time taken by the pipeline to process all tiles in the $\xi \times \xi$ tile array. Denote the ξ^2 tiles in the array, by $\tau_0, \tau_1, \cdots, \tau_{\xi^2-1}$ where the indices $0, 1, \cdots, \xi^2 - 1$ reflect the number of tile in tile ordering (see Sections 4.2,4.3). This, coupled with the fact that a stage S_ℓ cannot process tile $\tau_{k+1,\ell}$ until it has finished with tile $\tau_{k,\ell}$, gives the following expression for the overall time T for processing all tiles through all stages is $T = T_{\xi^2-1,2c} + t_{\xi^2-1,2c}$

Definition 1. Consider a $(2c + 1)$-stage pipeline. For any fixed ℓ, where $0 \leq \ell \leq 2c$, Stage S_ℓ is a *comparable stage*, iff for all $0 \leq k < \xi^2$ and for every $0 \leq \ell' \leq 2c$, either $t_{k,\ell} \leq t_{k-\ell'+\ell,\ell'}$ or $t_{k,\ell} \geq t_{k-\ell'+l,\ell'}$.

In a stage that is not comparable, for some ℓ' we could have $t_{k,\ell} < t_{k-\ell'+\ell,\ell'}$ while for a different ℓ', we have $t_{k,\ell'} > t_{k-\ell'+\ell,\ell'}$

Definition 2. A $(2c + 1)$-stage pipeline, is a *totally ordered pipeline*, iff for all $0 \leq \ell \leq 2c$, Stage S_ℓ is a comparable stage.

In our application, each stage performs a specific activity on different tiles. The time for a stage is generally a function of the task it performs, as all tiles are of same size. Therefore we may expect our pipeline to be totally ordered.

In a totally ordered pipeline, For all $0 \leq k < \xi^2$, each pair of stages S_ℓ and $S_{\ell'}$ satisfies either $t_{k,\ell} \leq t_{k-\ell'+\ell,\ell'}$ or $t_{k,\ell} \geq t_{k-\ell'+\ell,\ell'}$. We use the notation $S_\ell \preceq S_{\ell'}$ to denote for all ℓ if $t_{k,\ell} \leq t_{k-\ell'+\ell,\ell'}$ and the notation $S_\ell \succeq S_{\ell'}$ to denote for all ℓ' if $t_{k,\ell} \geq t_{k-\ell'+\ell,\ell'}$. Intuitively, $S_\ell \preceq S_{\ell'}$, implies that S_ℓ will not hold $S_{\ell'}$ up due to the time it takes to process a tile.

Lemma 7. If $S_0 \preceq S_1$, then $T_{k+1,0} \leq T_{k,1}$

Proof: We proceed by induction on $k \geq 0$. For $k = 0$, we consider $T_{1,0}$ and $T_{0,1}$. From Theorem 3 and Lemma 5 we know that $T_{1,0} = T_{0,1} = t_{0,0}$. This implies $T_{1,0} \leq T_{0,1}$. This

proves the base case.
Assuming the lemma to hold for all $0 \leq k \leq \xi^2 - 2$, consider $T_{k+2,0}$ and $T_{k+1,1}$.

$$\begin{array}{rcll} T_{k+1,1} & = & \max\{T_{k+1,0} + t_{k+1,0},\ \ T_{k,1} + t_{k,1}\} & \text{from Theorem 3} \\ & = & \max\{T_{k+2,0},\ \ T_{k,1} + t_{k,1}\} & \text{from Lemma 5} \\ & \geq & T_{k+2,0} & \end{array}$$

■

We now generalize the above lemma to stages that are not necessarily neighbors.

Lemma 8. For any $0 \leq \ell \leq 2c$, if, $S_0, S_1, \cdots, S_{\ell-1} \preceq S_\ell$ then, for all $0 \leq k \leq \xi^2 - 1$, $T_{k+1,\ell} \leq T_{k,\ell+1}$.

Proof: We proceed by induction on $\ell \geq 0$. For $\ell = 0$ consider $T_{k+1,0}$ and $T_{k,1}$. We know that $T_{k+1,0} \leq T_{k,1}$. (from Lemma 7)
Assuming the current lemma to hold for any $\ell' < \ell$, consider $T_{k+1,\ell}$ and $T_{k,\ell+1}$.

$$\begin{array}{rcll} T_{k+1,\ell} & = & \max\{T_{k+1,\ell-1} + t_{k+1,\ell-1},\ \ T_{k,\ell} + t_{k,\ell}\} & \text{from Theorem 3} \\ & = & T_{k,\ell} + t_{k,\ell} & \text{as } T_{k,\ell} \geq T_{k+1,l-1} \text{ (induction hypothesis)} \\ & & & \text{and } t_{k,\ell} \geq t_{k+1,\ell-1} \text{ as } S_{\ell-1} \preceq S_\ell \end{array}$$

That is,

$$T_{k+1,\ell} = T_{k,\ell} + t_{k,\ell} \tag{5.1}$$

Now,

$$\begin{array}{rcll} T_{k,\ell+1} & = & \max\{T_{k,\ell} + t_{k,\ell},\ \ T_{k-1,\ell+1} + t_{k-1,\ell+1}\} & \text{from Theorem 3} \\ & = & \max\{T_{k+1,\ell},\ \ T_{k-1,\ell+1} + t_{k-1,\ell+1}\} & \text{from Equation (5.1)} \\ & \geq & T_{k+1,\ell} & \end{array}$$

■

Although the following lemma seems obvious from above Lemma, it does not follow directly. So we prove it below.

Lemma 9. For any $0 \leq \ell \leq 2c$, if, $S_0, S_1, \cdots, S_{\ell-1} \succeq S_\ell$ then, for all $0 \leq k \leq \xi^2 - 1$, $T_{k+1,\ell} \geq T_{k,\ell+1}$.

Proof: We proceed by induction on $\ell \geq 0$. For $\ell = 0$ consider $T_{k+1,0}$ and $T_{k,1}$. We know that $T_{k+1,0} \geq T_{k,1}$. (from Lemma 7)

Assuming the lemma to hold for any $\ell' < \ell$, consider $T_{k+1,\ell}$ and $T_{k,\ell+1}$.

$$\begin{array}{rcll} T_{k,\ell+1} & = & \max\{T_{k,\ell} + t_{k,\ell}, \quad T_{k-1,\ell+1} + t_{k-1,\ell+1}\} & \text{from Theorem 3} \\ & = & T_{k,\ell} + t_{k,\ell} & T_{k,\ell} \geq T_{k-1,\ell+1} \text{ by induction hypothesis} \\ & & & \text{as given is lemma } t_{k,\ell} \geq t_{k-1,\ell+1} \\ & & & \text{as } S_\ell \preceq S_{\ell-1} \end{array}$$

That is

$$T_{k,\ell+1} = T_{k,\ell} + t_{k,\ell} \tag{5.2}$$

Now,

$$\begin{array}{rcll} T_{k+1,\ell} & = & \max\{T_{k+1,\ell-1} + t_{k+1,\ell-1}, \quad T_{k,\ell} + t_{k,\ell}\} & \text{equation (3)} \\ & = & \max\{T_{k+1,\ell} + t_{k+1,\ell}, \quad T_{k,\ell+1}\} & \text{from Equation 5.2} \\ & \geq & T_{k,\ell+1} & \end{array}$$

■

In is a $(2c+1)$-stage totally ordered pipeline then there exists a stage S_m such that $S_m \succeq S_\ell$ for all $0 \leq \ell \leq 2c$. This stage is called *the maximal stage*. It will never have to wait on any of its previous stages. The following lemma formalizes this statement.

Lemma 10. If S_m is a maximal stage, then for all $0 \leq k < \xi^2$, $T_{k,m} = T_{k-1,m} + t_{k-1,m}$.

Proof: We know that

$$\begin{array}{rcll} T_{k,m} & = & \max\{T_{k-1,m} + t_{k-1,m}, \quad T_{k,m-1} + t_{k,m-1}\} & \text{from Theorem 3} \\ & = & T_{k-1,m} + t_{k-1,m} & \text{as } T_{k-1,m} \succeq T_{k,m-1} \text{ and Lemma 8} \end{array}$$

■

Lemma 11. If stage S_m is a maximal stage of a $(2c+1)$-stage totally ordered pipeline, then $T_{k,m} = \sum_{v=0}^{m-1} t_{0,v} + \sum_{u=0}^{k-1} t_{u,m}$.

Proof: From Lemma 10, $T_{k,m} = T_{k-1,m} + t_{k-1,m}$. Solving this recurrence, we get $T_{k,m} = T_{0,m} + \sum_{u=0}^{k-1} t_{u,m}$ (from Lemma 4). We know that $T_{0,m} = \sum_{v=0}^{m-1} t_{0,v}$ (from Lemma 6). Therefore $T_{k,m} = \sum_{v=0}^{m-1} t_{0,v} + \sum_{u=0}^{k-1} t_{u,m}$. ■

Let T_M denotes the time for stage S_m to process all ξ^2 tiles. Let T denotes the total time for the pipeline to process all tiles.

Theorem 12. In a $(2c+1)$-stage totally ordered pipeline the total time to run all ξ^2 tiles is $T = T_M + T_E$, where T_M is the time for the maximal stage to process all ξ^2 tiles and T_E is the time taken by the stages $S_{m+1}, S_{m+2}, \cdots, S_{2c}$ to process the last tile τ_{ξ^2-1}.

Proof: After the time T_M, all ξ^2 tiles have been processed by the stage S_m. For any stage S_ℓ, processing tile $\tau_{\xi^2-1,\ell}$ implies that all tiles $\tau_{0,\ell}, \tau_{1,\ell}, \cdots, \tau\xi^2 - 1, \ell$ have been processed. Therefore if stages $S_{m+1}, S_{m+2}, \cdots S_{2c}$ each process tile τ_{ξ^2-1}, then all tiles would have been processed. Therefore $T = T_M + T_E$. ∎

Time T_E depends on the times taken by the stages $S_{m+1}, S_{m+2}, \cdots S_{2c}$. Since all these stages have to process the last tile τ_{ξ^2-1} clearly, $T_E \geq \sum_{v=m+1}^{2c} t_{\xi^2-1,v}$. The theorem below identifies a case where this lower bound on T_E is achieved.

Lemma 13. If $S_\ell \succeq S_{\ell+1} \succeq \cdots \succeq S_{\ell'}$, then $T_{k,\ell'} = \sum_{v=\ell}^{\ell'-1} t_{k,v} + T_{k,\ell}$

Proof: We proceed by induction on $\ell \leq \ell'$. For $\ell = \ell'$, consider $T_{k,\ell'} = T_{k,l}$. This proves the base case. Assuming the lemma to hold for $0 < \ell < \ell'$ consider $\ell - 1$. so $S_{\ell-1} \succeq S_\ell$. By Theorem 3, $T_{k,\ell} = \max\{T_{k,\ell-1} + t_{k,\ell-1}, \quad T_{k-1,\ell} + t_{k-1,\ell}\}$. Since $S_{\ell-1} \succeq S_\ell$, we have $t_{k,\ell-1} \geq t_{k-1,\ell}$ and by Lemma 9, $T_{k,\ell-1} \geq T_{k-1,\ell}$.
Therefore

$$T_{k,\ell} = T_{k,\ell-1} + t_{k,\ell-1}. \tag{5.3}$$

By the induction hypothesis, $T_{k,\ell'} = \sum_{v=\ell}^{\ell'-1} t_{k,v} + T_{k,\ell}$. Substituting for $T_{k,\ell}$ (from Equation (5.3)) we have,

$$\begin{aligned} T_{k,\ell'} &= \sum_{v=\ell}^{\ell'-1} t_{k,v} + t_{k,\ell-1} + T_{k,\ell} \\ &= \sum_{v=\ell-1}^{\ell'-1} t_{k,v} + T_{k,\ell} \end{aligned}$$

∎

Theorem 14. In a $(2c+1)$-stage totally ordered pipeline with maximal stage S_m and for which $S_{m+1} \succeq S_{m+2} \succeq \cdots \succeq S_{2c}$, the total time to process all tiles is

$$T = \sum_{v=0}^{m-1} t_{0,v} + \sum_{u=0}^{\xi^2-1} t_{u,m} + \sum_{v=m+1}^{2c} t_{\xi^2-1,v}$$

Proof: From Lemma 13, $T = T_{\xi^2-1,2c} + t_{\xi^2-1,2c} = \sum_{v=m+1}^{2c} t_{\xi^2-1,v} + T_{\xi^2-1,m+1}$

$T_{\xi^2-1,m+1} = T_{\xi^2-1,m} + t_{\xi^2-1,m}$ (again by Lemma 13). substituting this value in the above expression. We have,

$$\begin{aligned} T &= \sum_{v=m+1}^{2c} t_{\xi^2-1,v} + T_{\xi^2-1,m} + t_{\xi^2-1,m} \\ &= T_{\xi^2-1,m} + \sum_{v=m}^{2c} t_{\xi^2-1,v} \\ &= \sum_{v=0}^{m-1} t_{0,v} + \sum_{u=0}^{\xi^2-1} t_{u,m} + \sum_{v=m}^{2c} t_{\xi^2-1,v} \quad \text{from Lemma 11} \end{aligned}$$

■

5.2 Memory

In this section we study space requirement for running the SIFT on the pipeline. Recall that a chip has three stages input, compute and output stage (see Figure 5.2). The input stage receives the data from the output stage of the previous chip (if any) and holds this information until the compute stage is ready for it. The compute stage processes this information employing, possibly some scratch-pad memory. However after computing, it passes the result to the output stage and moves on to the next tile. Ignoring the scratch pad memory, the memory requirement of the chip can be estimated from the requirements of the input and output stages.

Let $\phi_{k,\ell}$ be the set of bits received when tile $\tau_{k,\ell}$ is being processed by Stage S_ℓ. Let $\eta_{k,\ell}$ be the set of bits required to process the tile $\tau_{k,\ell}$. Clearly we can say that $\eta_{k,\ell} \subseteq \bigcup_{i=0}^{k} \phi_{i,\ell}$. After computing tile $\tau_{k,\ell}$, the Stage S_ℓ discards some bits say $\psi_{k,\ell}$ which are not required further. Set of bits that are left in Stage S_ℓ after processing tile $\tau_{k,\ell}$ are $\bigcup_{i=0}^{k} \phi_{i,\ell} - \bigcup_{i=0}^{k} \psi_{i,\ell} = M_{k,\ell}$.

Memory requirement of Stage $S_\ell = \max\ \{M_{k,\ell}\}$ where $0 \leq k < \xi^2 - 1$

5.3 Input Protocol

In this section we describe the following two input protocols considered in this thesis.

Tile-plus-neighborhood: As mentioned earlier, in order to process a tile in the computation stage S_1 some additional data is needed in the form of an x-neighborhood of the tile. That is a $(2x + n) \times (2x + n)$ array of pixels with an extra width of x around the $n \times n$ tile. In this tile-plus-neighborhood protocol the additional neighborhood information is also sent to the Stage S_0 of the pipeline model.

Tile-only: In this protocol only the tiles are sent to the Stage S_0 of the pipeline model. However the amount of data that needs to be sent to the Stage S_0 for the very first tile in the pipeline will be sum of all the tiles that have to present in the chip to start its computation on first tile. So S_1 may not be able to start on the tile immediately as the neighborhood data has not been known.

5.4 Architecture

In this section we briefly describe the two architectures we considered for the compute stage of each chip.

Uniprocessor: For a Uniprocessor model of computation, the standard RAM (Random Access Machine) model is assumed. In this model a single processor on a chip executes one instruction at a time.

Multiprocessor: For the Multiprocess model of computation, The Hierarchical Multilevel Multicore (HM model) 8.1 is assumed. Here model admits a memory hierarchy and thus abstracts away small details without ignoring expensive memory access.

5.5 Pipelining Multiple Images

In this section we extend the ideas discussed so far to the processing of multiple images in the pipeline model. So far we have considered pipelining of tiles of a single image. As we will see in Chapter 6, some stages finish their tasks before others. We therefore consider the processing of a stream of m images, each of size $N \times N$ images. This is not significantly different from that of tiling and feeding a single image except that memory requirement do

not cross the image boundaries. If there are M number of images are going through the pipeline one after the other then the overall time taken by the pipeline to process these images completely is given by $T_M = T_{M(\xi)^2-1,2c} + t_{M(\xi)^2-1,2c}$.

Chapter 6

Input Data Flow Requirements

In this section/chapter we discuss the flow of the input data required for chip C_0 (the first chip) to start its work on the tile. Subsequent flow of data of the pipeline will be governed by this. In order to process a tile of size $n \times n$, the algorithm requires some additional data related to the tile being processed. In this section, we will discuss the number of bits of data (tile + neighborhood) that must be delivered at stage S_0 of the pipeline model at each round of the algorithm. As noted in Section 5.3, we consider two protocols for the flow of data between the two stages tile-plus neighborhood and tile-only protocols.In Section 6.1, we discuss the input data requirements for the tile-plus neighborhood protocol and in Section 6.2, we deal with the input data of the tile-only protocol.

As noted earlier, the tile has n^2 pixels and the image is of size $N \times N$. Th neighborhood of the tile is defined in terms of the quantity x as follows.

Let τ be the set of pixels in the tile. For any pixel $\pi \in \tau$, let $nbr(\pi)$ be the set of pixels located at a distance x from π in all directions. Then for pixel (i, j), The quantity

$$nbr(i,j) = \{(k,\ell) \text{ where } i - x \ \leq \ k \ \leq \ i + x \text{ and } j - x \ \leq \ \ell \ \leq \ j + x\} - \{(i,j)\}\,.$$

For the entire tile τ, the neighborhood is

$$nbr(\tau) = \left(\bigcup_{\pi \in \tau} nbr(\pi) \right) - \tau$$

The neighborhood $nbr(\tau)$ represents the extra pixels (over and beyond τ) needed for processing tile τ. For any tile τ, $nbr(\tau)$ is small square band of width x around τ (see Fig-

ure 6.1.) We will refer this as the x-neighborhood of the tile.

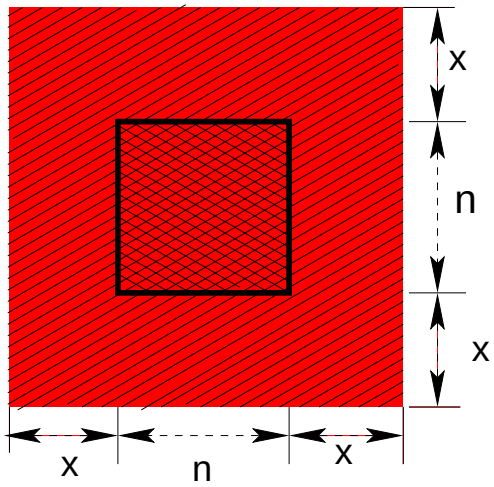


Figure 6.1: A tile and its neighborhood

Let the quantity x defined above satisfy $x = qn + r$ where $q = \lceil \frac{x}{n} \rceil$ and $r = x \bmod n$

6.1 Tile-Plus-Neighborhood Protocol

In this section we discuss the input data required by Stage S_0 of the pipeline to process the tile τ_k for the tile-plus-neighborhood protocol (see Section 5.3). In this protocol the data brought in is the tile and the neighborhood required to process the tile.

In the following sections we describe the input data required by the two tile orderings mentioned in Sections 4.2, 4.3.

Before we proceed to the different tile orderings, we illustrate a typical setting in the algorithm. This will allows us to define a convention that will be followed. If τ_k is to be processed then all the tiles τ_ℓ where $0 \leq \ell < k$ have already been processed, and to do that their neighborhoods must have been brought in (see Figure 6.2) red or darkly shaded area. This leaves just the portion shaded in yellow to be brought in during current round k. The singly hatched portion (region $\langle C \rangle$) represents the neighborhood of tile τ_k (region $\langle A \rangle$), which is doubly hatched. The previous tile in the image and its neighbourhood which had been brought in already is shown in dotted line (region $\langle B \rangle$). Regions $\langle F \rangle$ and $\langle E \rangle$ show the neighborhood of previous tile. This basic theme has many variations depending on the position of the tile in the image and the value of x. For example if $x \gg n$, then

the neighbourhood of several tiles at the right and bottom ends of the image would have been brought in several rounds in advance.

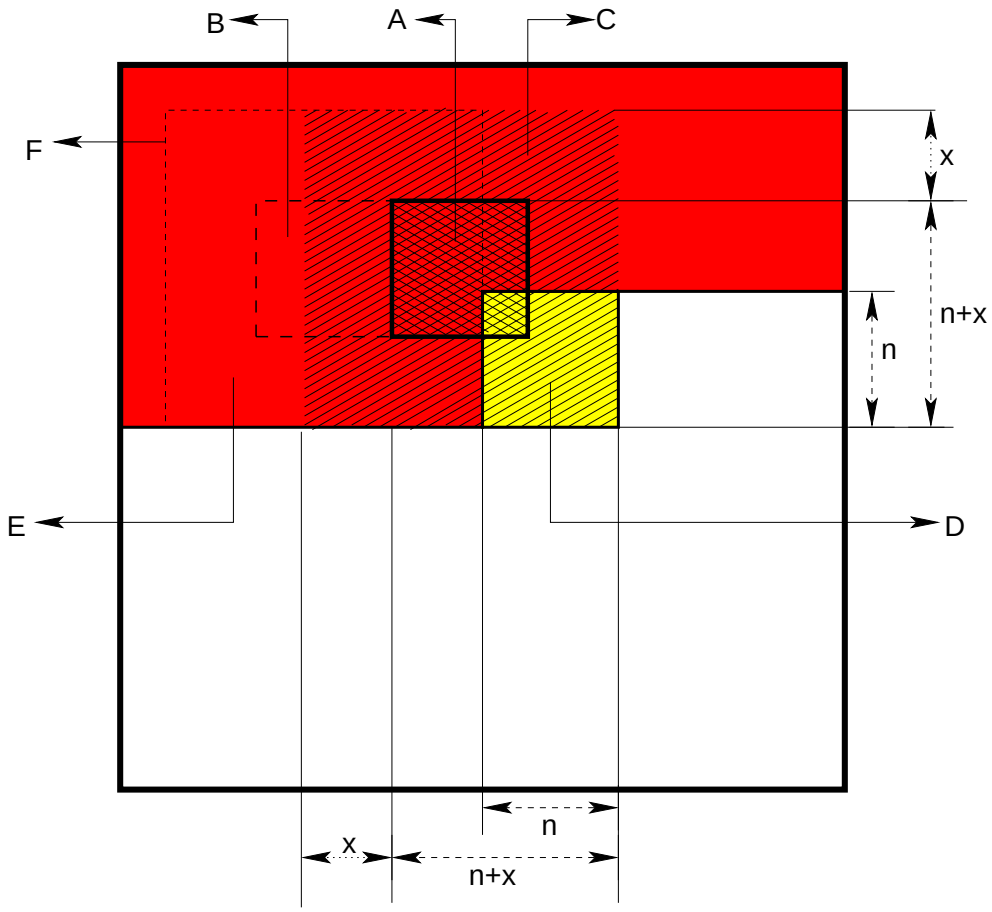


Figure 6.2: Tile and its neighborhood in the context of entire image

Note that (almost) every tile and its neighborhood consists of $(n + 2x)^2$ pixels. The information we address below is not about the number of pixels needed to process a tile (which is $(n + 2x)^2$ pixels) rather it is the number of pixels that need to be brought in at the start of the round that process a tile. We use this term pixels "input" (rather than "needed") for a tile to be processed. Here for a given tile ordering $\circledcirc$, if the tile in row i and column j of the tile array is processed k^{th} in order (that is $\tau_{i,j}^{\circledcirc} = \tau_k^{\circledcirc}$). We will refer to the bits input for the tile $\tau_{i,j}$ with the ordering $\circledcirc$ and hence k implied by the context.

6.1.1 Row Major Tile Ordering

In this section, we describe the data required at the stage S_0 of the pipeline model for the row major tile ordering described in Section 4.2. This is explained through a series of lemmas that address the different cases depending on the position (i, j) of tile $\tau_{i,j}$.

Lemma 15. For $i = 0$ and $j = 0$, the number of pixels input for tile $\tau_{i,j}$ is $(n + x)^2$.

Proof: See Figure 6.3.This is the first tile of the image. Clearly no bits have been received yet. So all the $(n + x)^2$ bits in the tile and its neighbourhood have to be brought in. ■

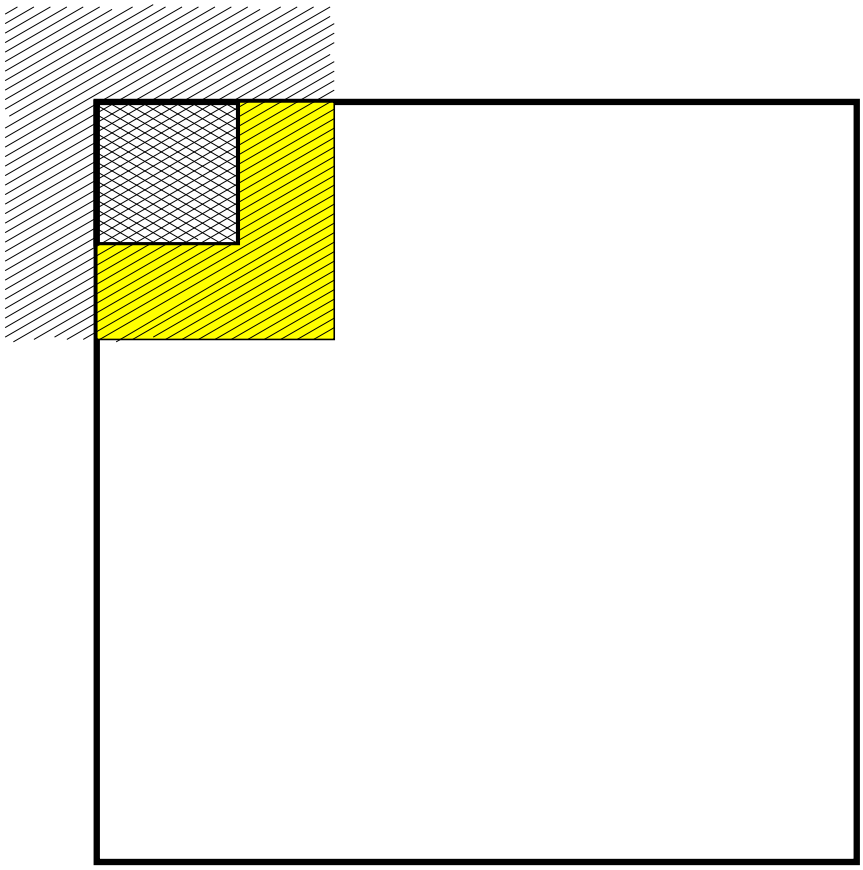


Figure 6.3: First tile of the row-major tile ordering

Note : Let $z = \lceil \frac{N-x}{n} \rceil$. Then consider tile $\tau_(i, z)$. The last column of pixels of this tile is column $zn - 1$ of the $N \times N$ image. The last column of the neighbourhood of $\tau_(i, z)$ is

$zn - 1 + x = \lceil \frac{N-x}{n} \rceil n - 1 + x > \left(\frac{N-x}{n}\right) n - 1 + x > N - 1$. Also $zn - 1 = \lceil \frac{N-x}{n} \rceil n < \left(\frac{N-x+1}{n}\right) n < N - x + 1 < N$.

This means that while tile $\tau_{i,z}$ is within the image, it neighbourhood includes the last column of pixels in the image. That is, $\tau_{i,z}$ is the last tile in row i for which an input is required. The input requirement for the tiles after tile $\tau_{i,z}$ ($\tau_{i,z+1}$, $\tau_{i,z+2}$, $\cdots$) is zero.

Lemma 16. For $r = 0$ and $0 < c < z$, the number of pixels input for the tile $\tau_{r,c}$ is $n(n + x)$.

Proof: From Figure 6.4, The number of pixels that should be inputed is shown by yellow portion and is equal to $n(n + x)$. ■

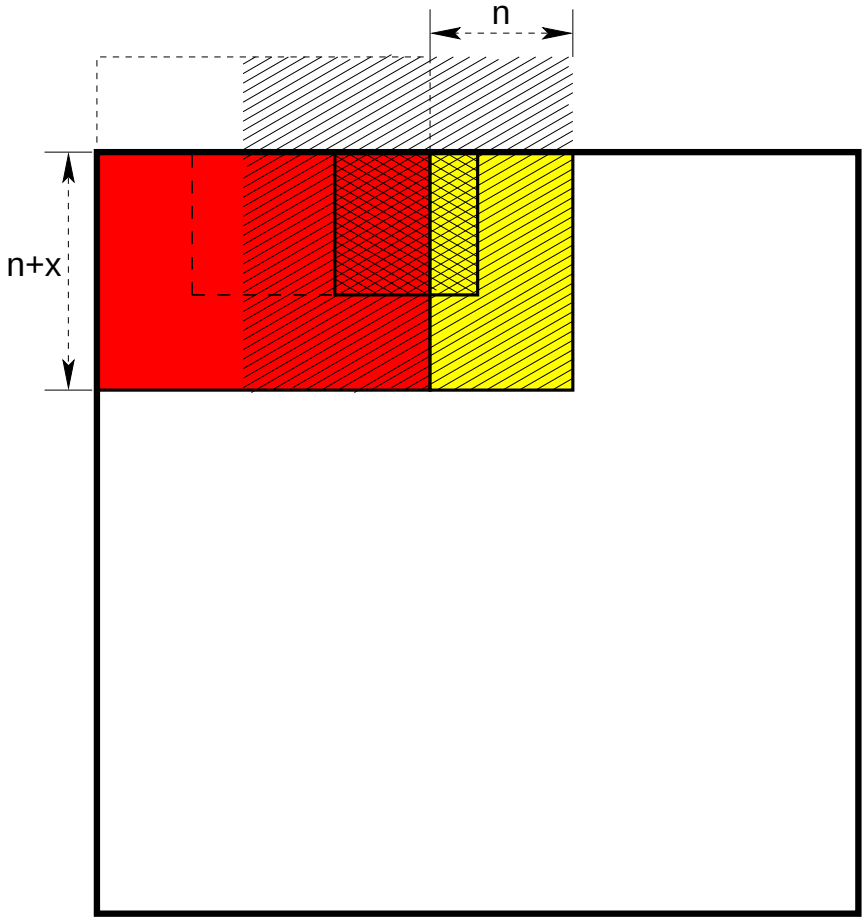


Figure 6.4: Tile in row 0 and column $0 < c < z$

Lemma 17. For $0 < r < z$ and $c = 0$, the number of pixels input for the tile $\tau_{r,c}$ is $n(n + x)$.

Proof: From Figure 6.5, The number of pixels that should be inputed is shown by yellow portion and is equal to $n(n+x)$. ■

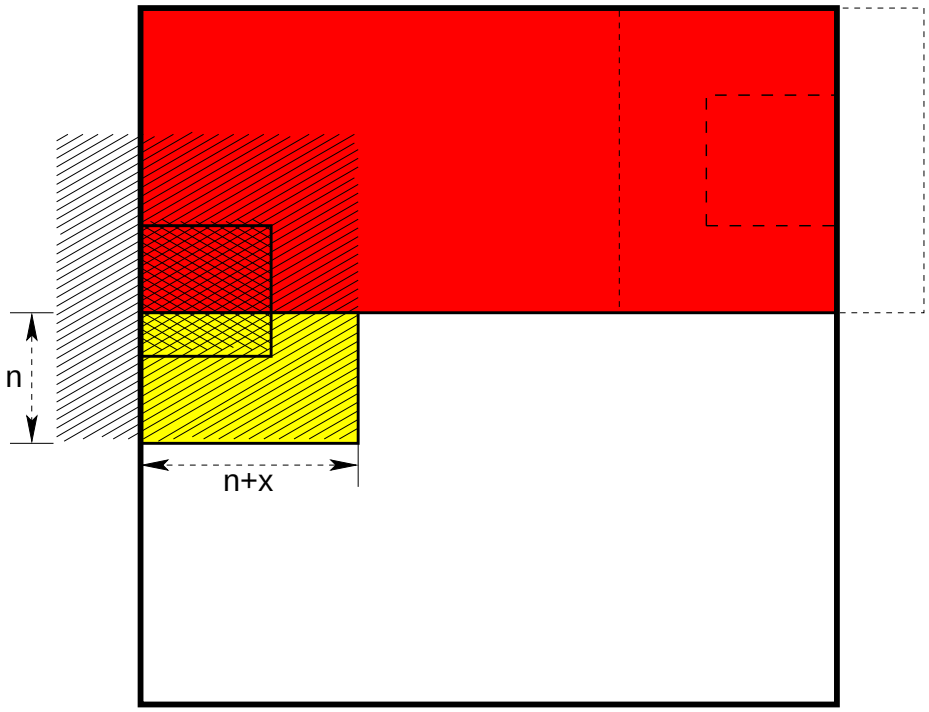


Figure 6.5: Tile in column 0 and in row $0 < r < z$ rows

Lemma 18. For $r = 0$ and $c = z$, the number of pixels input for the tile $\tau_{r,c}$ is $(n-r)(n+x)$.

Proof: From Figure 6.6, the neighbourhood of the file which lies outside the image does not exist. The data that needs to be brought in shown by the region in yellow.

The area of yellow region = length of $\langle 1,3\rangle$ * length of $\langle 3,4\rangle$. We have length of $\langle 3,4\rangle = n+x$ and length of $\langle 1,3\rangle$ = [length of $\langle 1,5\rangle$ - length of $\langle 3,5\rangle$] $= n-(nz+x-N) = n-n\ \left(\left\lceil \frac{N-x}{n} \right\rceil\right) + x - N = n-n\ \left(\left\lceil \frac{N}{n} - q \right\rceil\right) + x - N = n-(N-q+x-N)) = n-(x-q) = n-r$.

Thus the area of the yellow region $= (n-r)(n+x)$ ■

Lemma 19. For $r = z$ and $c = 0$, the number of pixels input for the tile $\tau_{r,c}$ is $(n-r)(n+x)$.

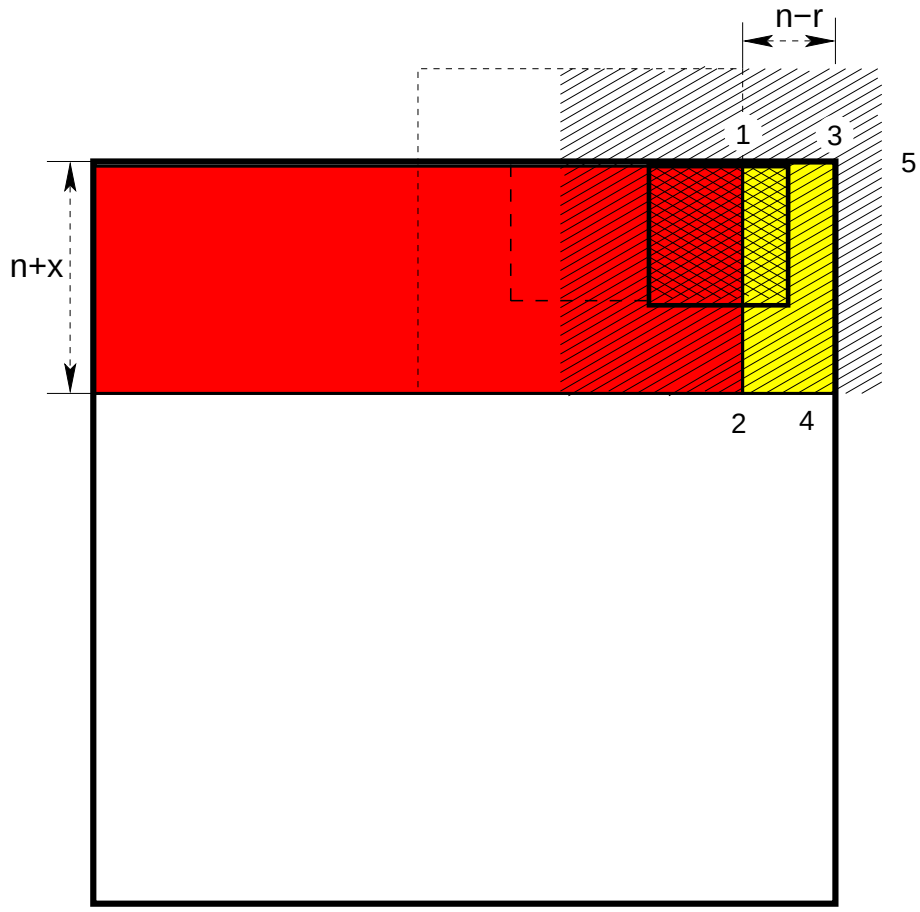


Figure 6.6: Tile in row 0 and column z

Proof: From Figure 6.7, along the lines of the proof of Lemma 18. ■

Lemma 20. For $0 < r, c < z$, the number of pixels input for the tile $\tau_{r,c}$ is n^2.

Proof: From Figure 6.8, the number of pixels that should be inputed is shown as the yellow portion and is equal to n^2. ■

Lemma 21. For $0 < r < z$ and $c = z$, the number of pixels input for the tile $\tau_{r,c}$ is $n(n-r)$.

Proof: From Figure 6.9, the number of pixels that should be inputed is shown in yellow which equals $n\ (n-r)$. ■

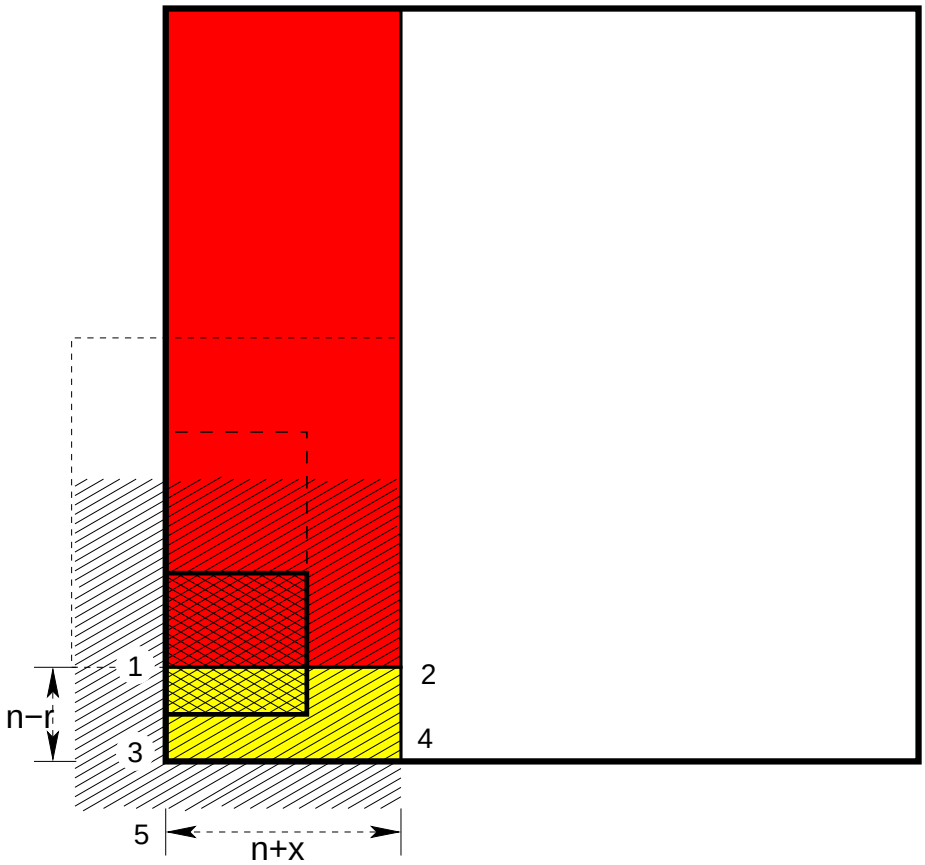


Figure 6.7: Tile in column 0 and row z

Lemma 22. For $0 < c < z$ and $r = z$, the number of pixels input for the tile $\tau_{r,c}$ is $n(n-r)$.

Proof: From Figure 6.10, the data need to be brought in is represented by the region shown in yellow color and it is equal to $n\ (n-r)$. ■

Lemma 23. For $r, c = z$, the number of pixels input for the tile $\tau_{r,c}$ is $(n-r)^2$.

Proof: From Figure 6.11

The number of pixels that should be inputed is shown by yellow portion and is equal to $(n-r)^2$. s ■

Lemma 24. For $r, c > z$, the number of pixels input for the tile $\tau_{r,c}$ is 0.

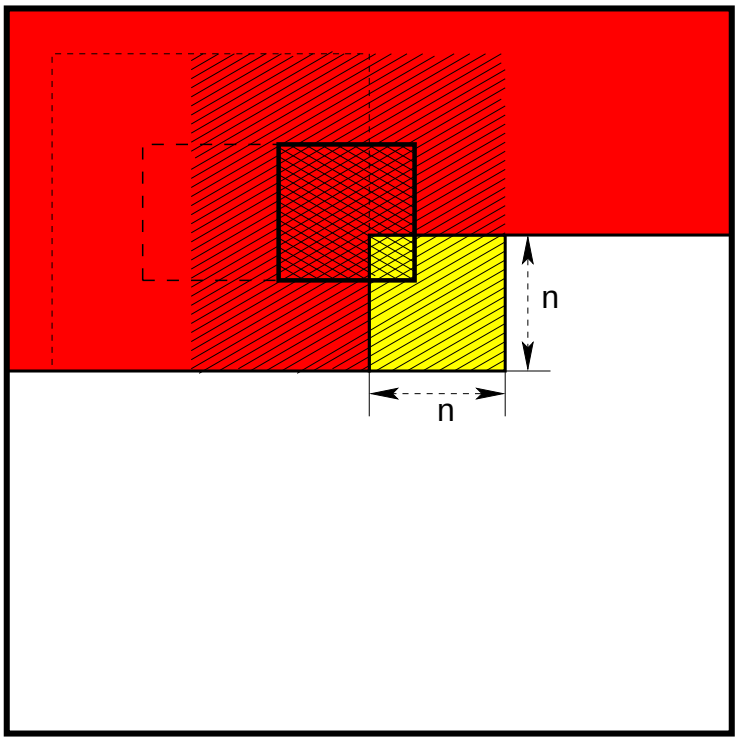


Figure 6.8: Tile in row $0 < r < z$ and column $0 < c < z$

Proof:

Here the number of pixels that are needed for $\tau_{r,c}$ have already been brought in at an earlier time. ■

The following theorem summarizes the above results

Theorem 25. The number of pixels input for the tile $\tau_{r,c}$ at Stage S_0 of the pipeline in

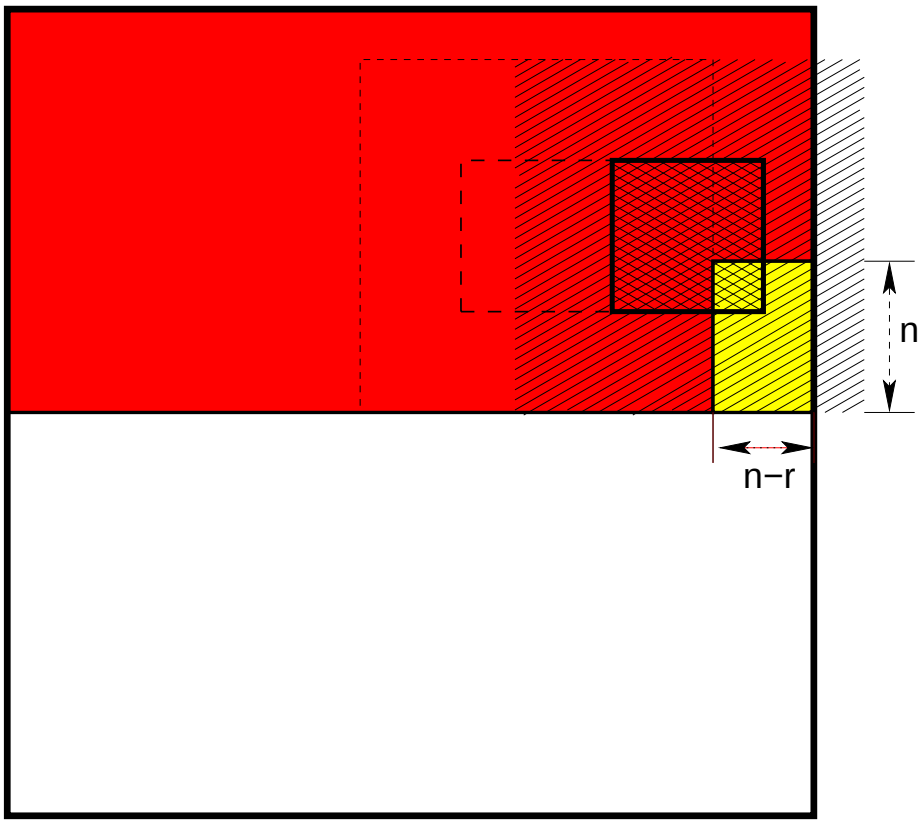


Figure 6.9: Tile in column z and row $0 < r < z$

row major tile ordering is

$$\eta_k^R = \begin{cases} (n+x)^2 & \text{if } r = 0 \text{ and } c = 0 \text{ ;} \\ n \cdot (n+x) & \text{if } r = 0 \text{ and } c < z \text{ OR } c = 0 \text{ and } r < z \text{ ;} \\ n^2 & \text{if } 0 < r, c < z \text{ ;} \\ (n-r)(n+x) & \text{if } r = 0 \text{ and } c = z \text{ OR } r = z \text{ and } c = 0; \\ n \cdot (n-r) & \text{if } r = z \text{ and } c < z \text{ OR } r < z \text{ and } c = z; \\ (n-r)^2 & \text{if } r = z \text{ and } c = z \text{ ;} \\ 0 & \text{if } z < r, c < \xi \end{cases}$$

■

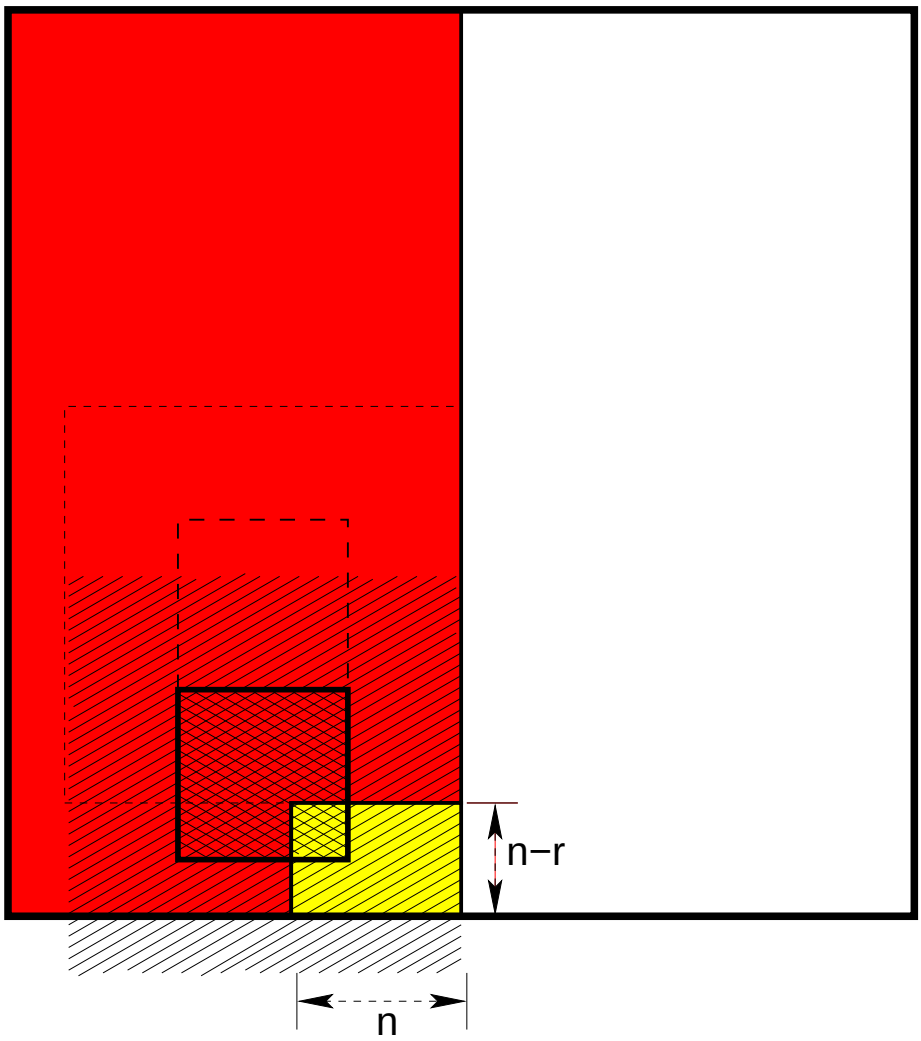


Figure 6.10: Tile row z and in column $0 < c < z$

6.1.2 Diagonal Method Tile Ordering

In this section we describe the data required at the Stage S_0 of the pipeline model for diagonal tile ordering as discussed in Section 4.3. In this tile ordering, the tiles are ordered by the value of $(i + j)$. For two tiles τ_{i_1,j_1} and τ_{i_2,j_2} with $(i_1 + j_1) = (i_2 + j_2)$. In this case the tile with the lower r value is enumerated first. That is, if $i_1 < i_2$ then rank of $\tau_{i_1,j_1} <$ rank of τ_{i_2,j_2}.

The proofs of most of the following lemmas follow along the lines of the proofs of Lemmas 15

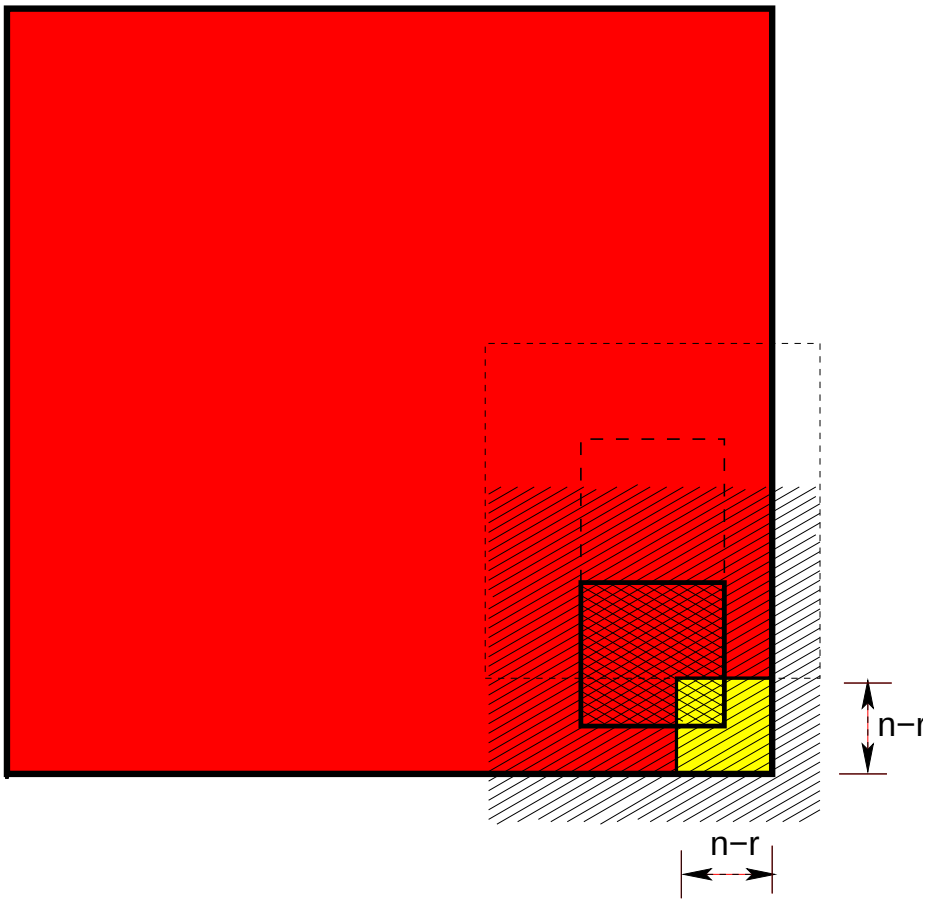


Figure 6.11: Tile in row z and column z

–24.

Lemma 26. For $r = 0$ and $c = 0$, the number of pixels input for tile $\tau_{r,c}$ is $(n + x)^2$.

Proof: This is the first tile of the image. Clearly no bits have been received yet. So all the $(n + x)^2$ bits in the tile and its neighbourhood have to be brought in. See Figure 6.12. ■

Lemma 27. For $r = 0$ and $0 < c < z$, the number of pixels input for the tile $\tau_{r,c}$ is $n(n + x)$.

Proof: See Figure 6.13 ■

Lemma 28. For $0 < r < z$ and $c = 0$, the number of pixels input for the tile $\tau_{r,c}$ is $n(n + x)$.

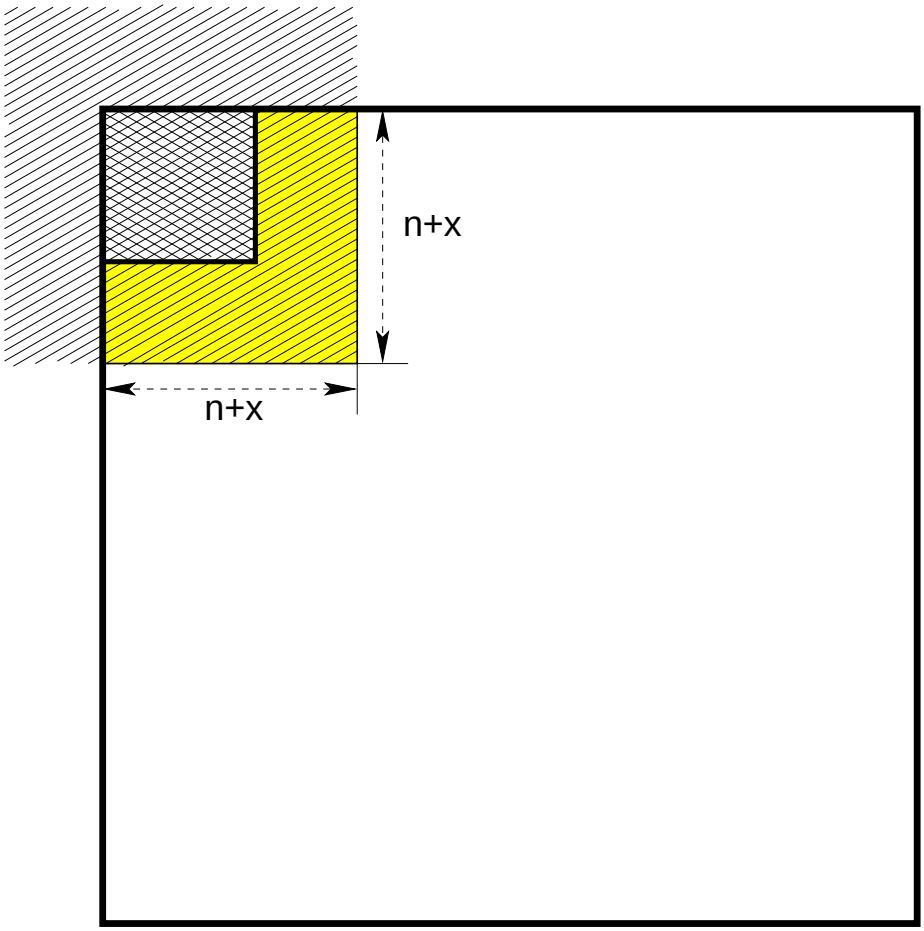


Figure 6.12: First tile of the diagonal tile ordering

Proof: See Figure 6.14 ■

Lemma 29. For $r = 0$ and $c = z$, the number of pixels input for the tile $\tau_{r,c}$ is $(n-r)(n+x)$.

Proof: From Figure 6.15, The area of yellow region = length of $\langle 1,3\rangle$ * length of $\langle 3,4\rangle$. We have length of $\langle 3,4\rangle = n+x$. The length of $\langle 1,3\rangle$ = [length of $\langle 1,5\rangle$ - length of $\langle 3,5\rangle$] $= n-(nz+x-N) = n-n\ \left(\left\lceil \frac{N-x}{n} \right\rceil\right) + x - N = n-n\ \left(\left\lceil \frac{N}{n} - q \right\rceil\right) + x - N = n-(N-q+x-N)) = n-(x-q) = n-r$.

The area of the yellow region is therefore $(n-r)(n+x)$. ■

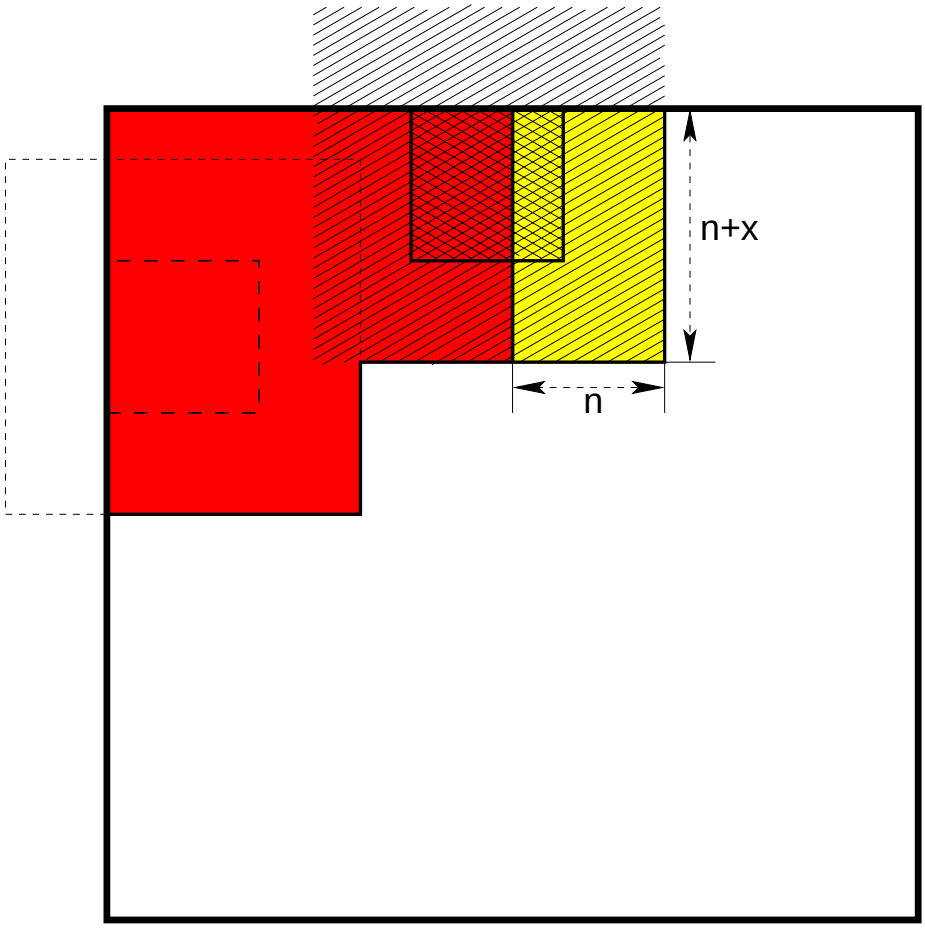


Figure 6.13: Tiles in row 0 and column $0 < c < z$

Lemma 30. For $r = z$ and $c = 0$, the number of pixels input for the tile $\tau_{r,c}$ is $(n - r)(n + x)$.

Proof: See Figure 6.16. ■

Lemma 31. For $0 < r, c < z$, the number of pixels input for the tile $\tau_{r,c}$ is n^2.

Proof: See Figure 6.17. ■

Lemma 32. For $0 < r < z$ and $c = z$, the number of pixels input for the tile $\tau_{r,c}$ is $n(n - r)$.

Proof: See Figure 6.18. ■

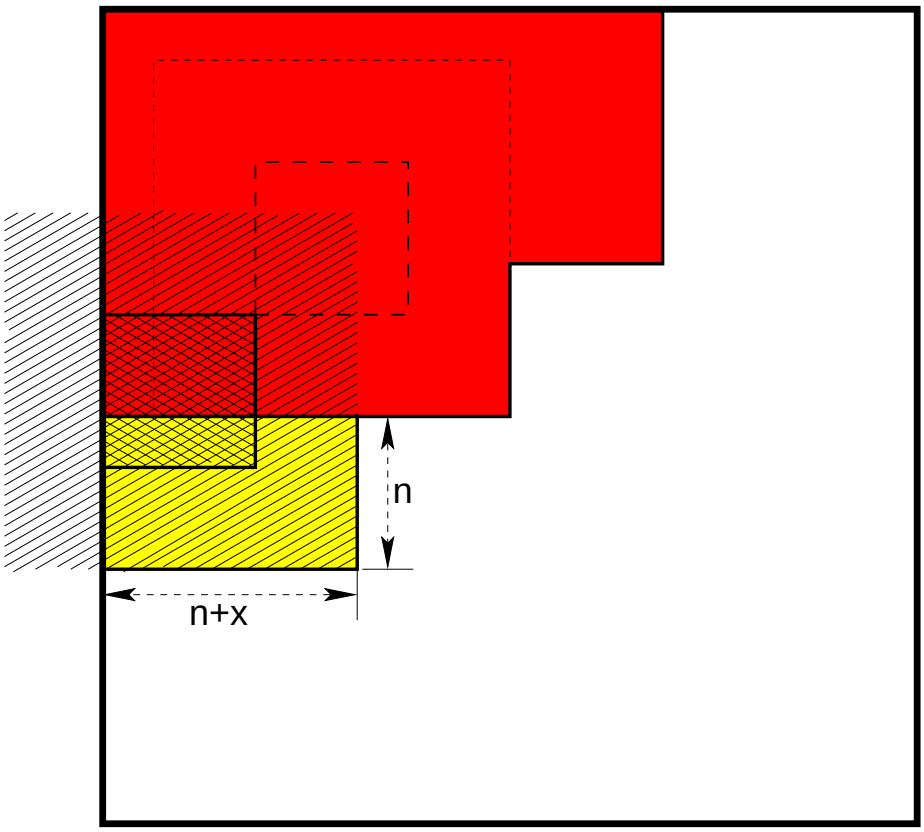


Figure 6.14: Tiles in column 0 and row $0 < r < z$

Lemma 33. For $0 < c < z$ and $r = z$, the number of pixels input for the tile $\tau_{r,c}$ is $n(n-r)$.

Proof: From Figure 6.19. ■

Lemma 34. For $r, c = z$, the number of pixels input for the tile $\tau_{r,c}$ is $(n-r)^2$.

Proof: See Figure 6.20. ■

Lemma 35. For $r, c > z$, the number of pixels input for the tile $\tau_{r,c}$ is 0.

Proof: In this case the number of pixels that should be sent as input the current round have been already brought in earlier time. So the number of pixels inputed in this round is zero. ■

From the proceeding of Lemmas we have the following theorem.

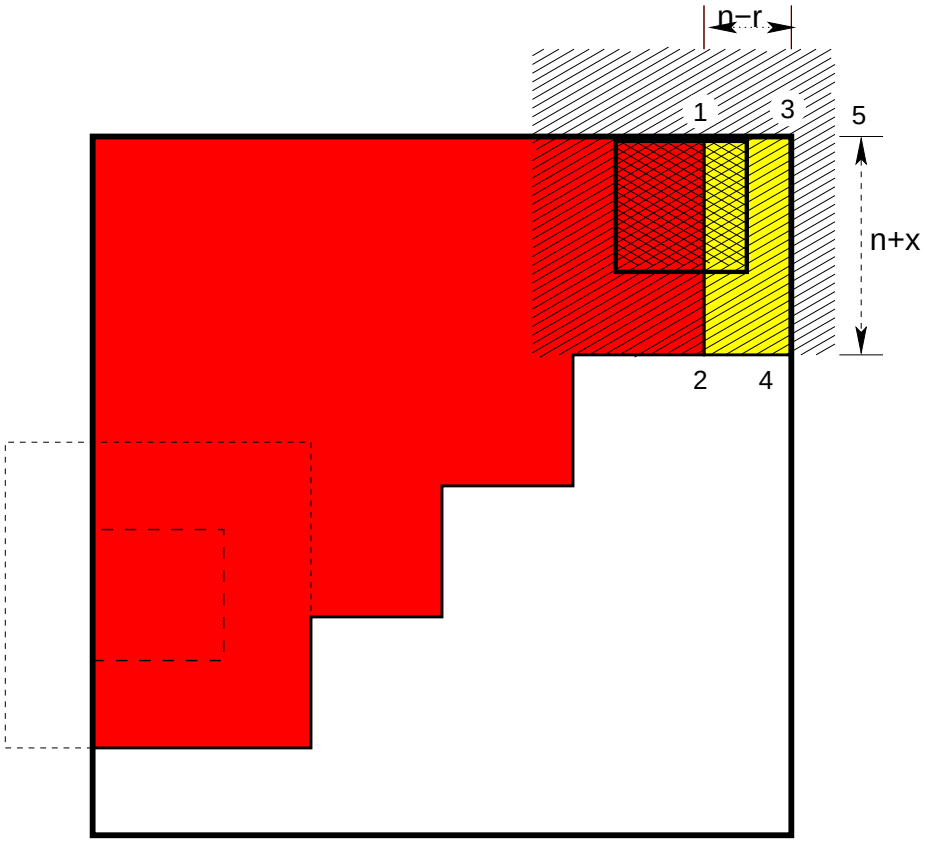


Figure 6.15: Tile in row 0 and column z

Theorem 36. the number of pixels input for the tile $\tau_{i,j}$ at stage S_0 of the pipeline in Diagonal Tile Ordering is

$$= \begin{cases} (n+x)^2 & \text{if } r = 0 \text{ and } c = 0 \ ; \\ n \cdot (n+x) & \text{if } r = 0 \text{ and } c < z \text{ OR } c = 0 \text{ and } r < z \ ; \\ n^2 & \text{if } 0 < r, c < z \ ; \\ (n-r)(n+x) & \text{if } r = 0 \text{ and } c = z \text{ OR } r = z \text{ and } c = 0; \\ n \cdot (n-r) & \text{if } r = z \text{ and } c < z \text{ OR } r < z \text{ and } c = z; \\ (n-r)^2 & \text{if } r = z \text{ and } c = z \ ; \\ 0 & \text{if } z \ < \ r, c \ < \ \xi. \end{cases}$$

■

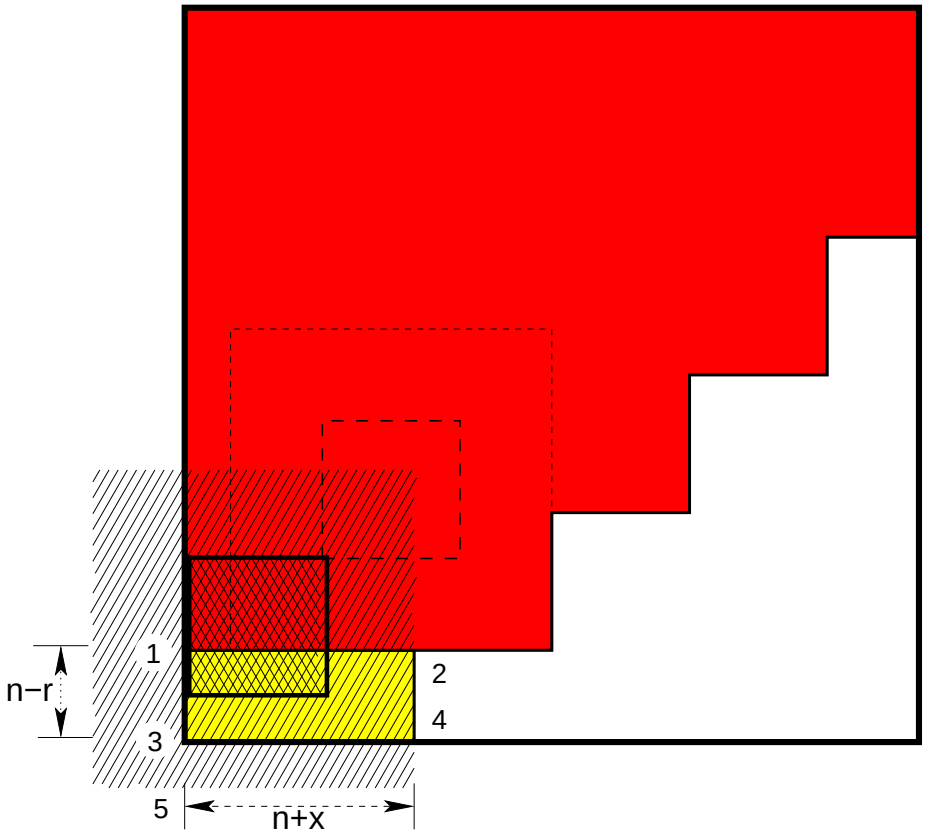


Figure 6.16: Tile in column 0 and row z

6.2 Tile-Only Protocol

So far we assumed that the input to Stage S_0 is in the form of an $n \times n$ tile and its neighborhood of width x forming an $(2x+n) \times (2x+n)$ tile and neighborhood. Without this neighborhood a tile cannot be processed. What would happen if the input received was only the tile (without its neighborhood)? This situation could happen when n is very small relative to x. We call such an input a "tile-only" input, as opposed to the "tile-plus-neighborhood" input.

Now we consider the time $t_{k,0}$ if tile only inputs are used. Before we proceed, we clarify the meanings of the quantities $t_{k,\ell}$ and $T_{k,\ell}$. We defined $T_{k,\ell}$ as the earliest starting time of Stage S_ℓ to work on tile τ_k and $t_{k,\ell}$ to be the time taken by Stage S_ℓ to process tile τ_k. We also derived the equation $T_{k,\ell} = \max\{T_{k,\ell-1} + t_{k,\ell-1},\ \ T_{k-1,\ell} + t_{k-1,\ell}\}$. In the context of a tile-plus-neighborhood input, the above meanings of $t_{k,\ell}$ and $T_{k,\ell}$ are consistent with

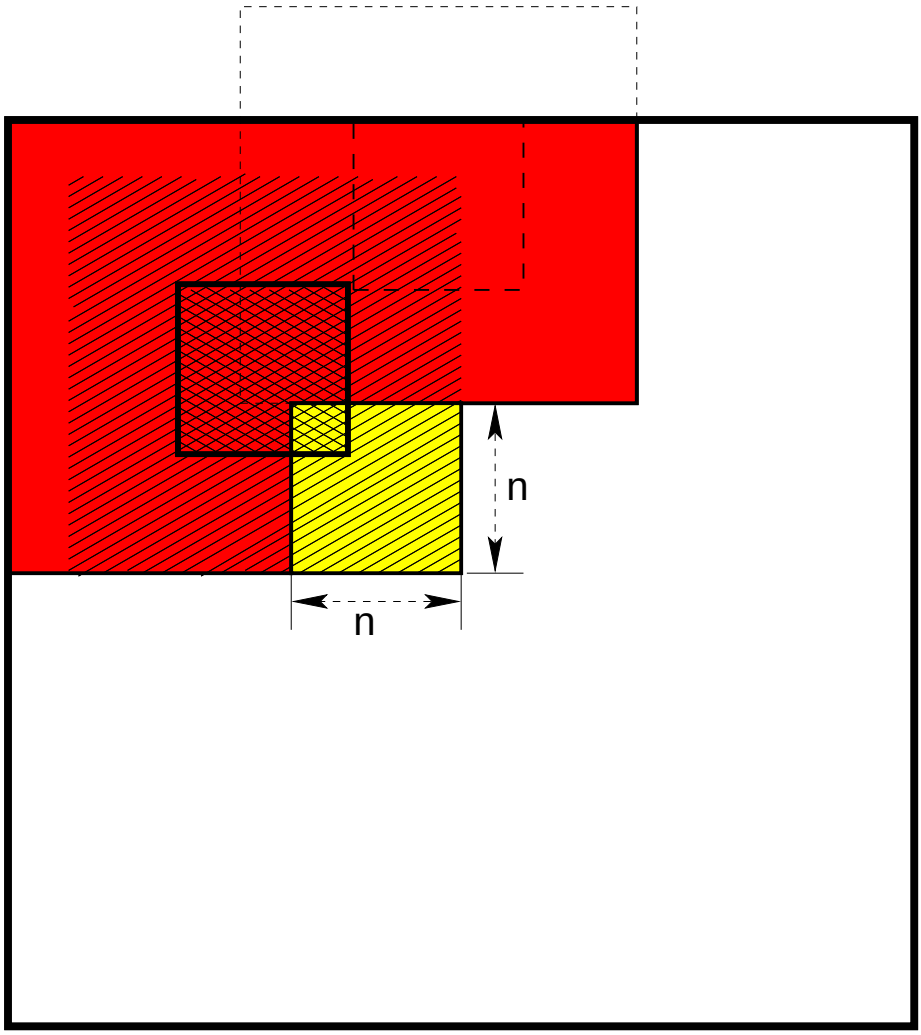


Figure 6.17: Tile in row $0 < r < z$ and column $0 < c < z$

the equation. The equation implies that if S_ℓ is free, then S_ℓ should be able to start on τ_k immediately after $T_{k,\ell-1} + t_{k,\ell-1}$ time. For the tile-only input if $t_{k,\ell}$ is the time taken to get any tile τ_k to Stage S_0, then the above meaning of $T_{k,1}$ would not hold. That is, S_ℓ may not be able to start on τ_k just because S_0 has received τ_k. Stage S_0 must also receive all tiles that includes the neighborhood of τ_k, before S_1 can start on τ_k. Thus $t_{k,\ell}$ is interpreted as the time needed for Stage S_ℓ to bring all the information needed for Stage $S_{\ell+1}$ to start on τ_k. We now examine Row major and Diagonal tile orderings with respect to Tile-only input protocol.

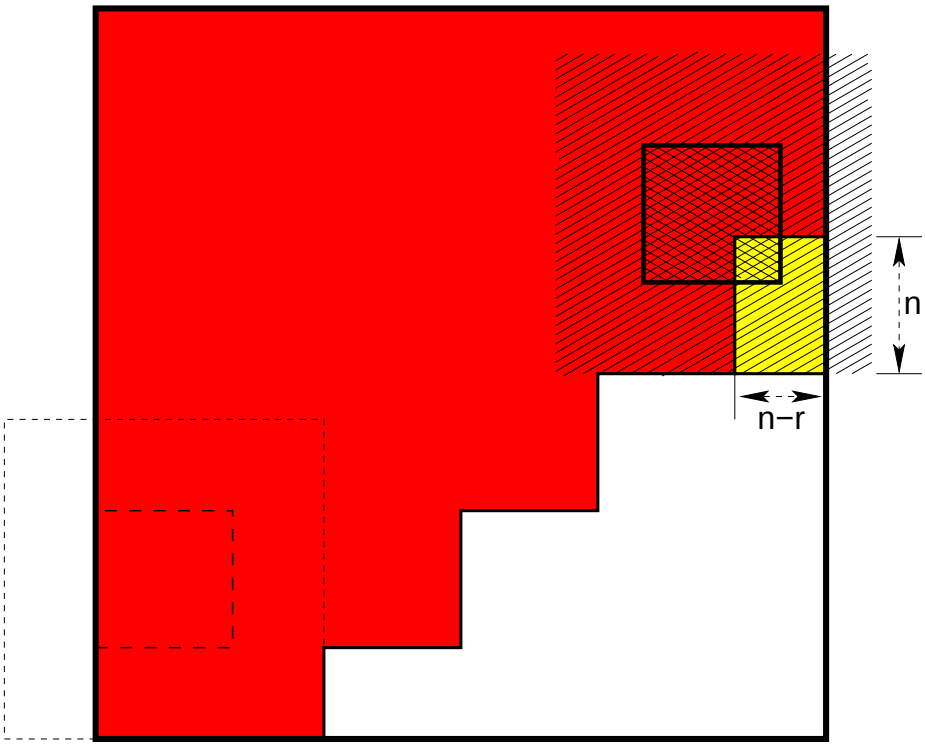


Figure 6.18: Tile in column z and row $0 < r < z$

6.2.1 Row Major Tile Ordering - Tile Only

Figure 6.21, shows tile τ_0 (or tile $\tau_{0,0}$), and a neighbourhood x around it. All pixels of the neighborhood will be received only after tile τ_{12} has been received at Stage S_0. So the the Stage S_1 would not start on tile τ_0 until tile τ_{12} has been received completely. In general tile $\tau_{i,j}$'s computation will not start on Stage S_1 until tile $\tau_{i+\lceil \frac{x}{n} \rceil, j+\lceil \frac{x}{n} \rceil} = \tau_{i+\delta, j+\delta}$ (where $\delta = \lceil \frac{x}{n} \rceil$) has been received completely. This means that $\tau_{0,0}$'s computation will not start until tile $\tau_{\lceil \frac{x}{n} \rceil, \lceil \frac{x}{n} \rceil} = \tau_{\delta,\delta} = \tau_{\delta(\xi+1)}$ is received. Until this time Stage S_1 idles. So the amount of data that needs to be received by Stage S_0 for Stage S_1 to process tile τ_0, equals the size of all tiles from τ_0 to $\tau_{\delta(\xi+1)}$. As each tile is of size n^2 pixels, the amount of data needed to process tile τ_0 is $[\delta(\xi+1)+1]\, n^2$. However the mount of data needed to be received by Stage S_0 to process the tile τ_k for all $k > 0$ is only n^2 pixels. This is because the neighborhood required to process these tiles has already been brought in, in the previous iterations. In the example of Figure 6.21, 15 tiles (shaded) need to be brought in before S_1 can process tile τ_0.

Theorem 37. The number of pixels input for the tile $\tau_{r,c} = \tau_k$ at Stage S_0 of the pipeline

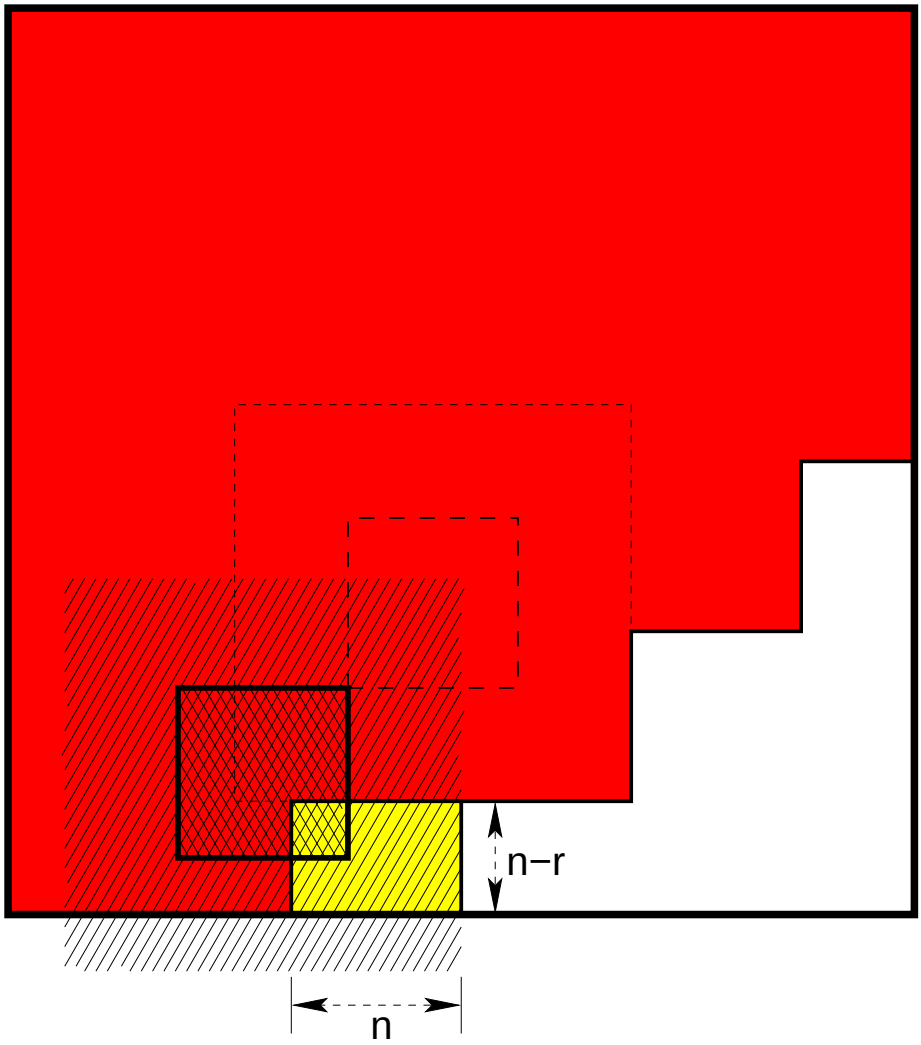


Figure 6.19: Tile in row z and column $0 < c < z$

in Row Major tile ordering for the tile only input format is

$$\eta_k^R = \begin{cases} [\delta(\xi+1)+1]\, n^2 & \text{if } r, c = 0 \text{ or } k = 0; \\ n^2 & \text{if } r, c > 0 \text{ or } k > 0\ ; \\ 0 & \text{if } r, c > \xi - 1 - \delta \text{ or } k > (\xi^2 - 1) - \delta(\xi+1) \end{cases}$$

■

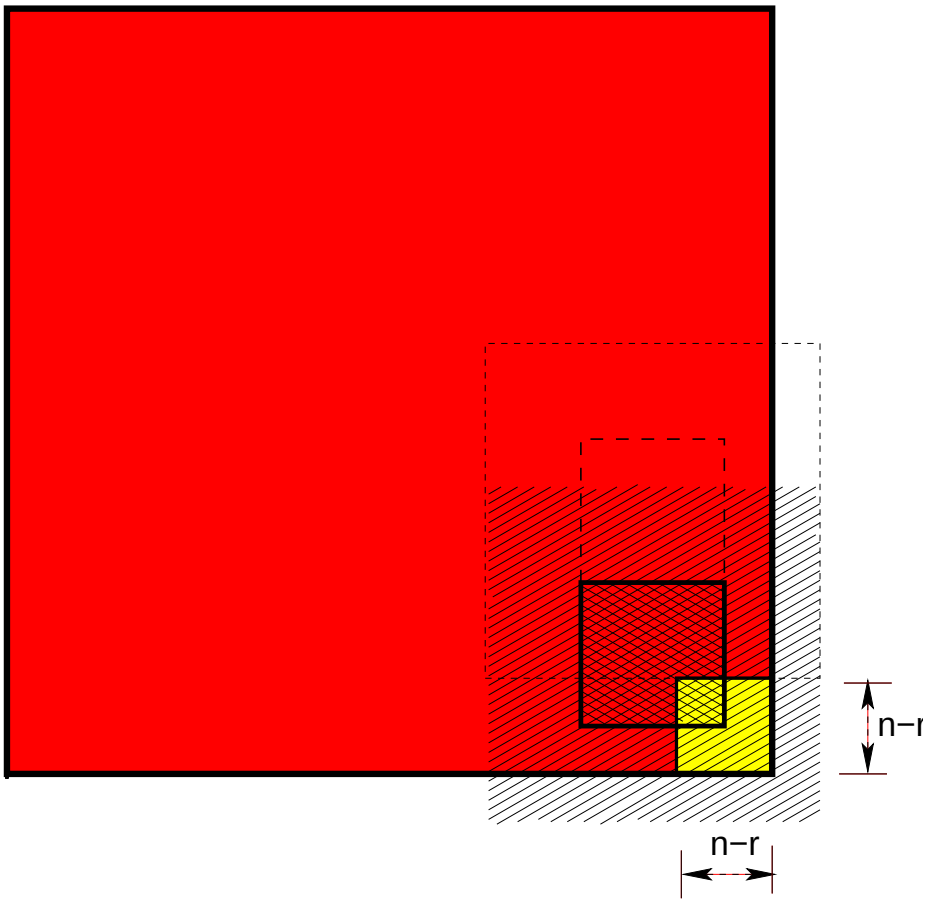


Figure 6.20: Tile in row z and column z

6.2.2 Diagonal Tile Ordering - Tile Only

Figure 6.21, shows tile τ_0 (or tile $\tau_{0,0}$), and a neighbourhood x around it. All pixels of the neighborhood will be received only after the tile τ_{12} has been receive at Stage S_0. So the the Stage S_1 can not work on tile τ_0 would not start until tile τ_{12} has been received completely. In general tile $\tau_{i,j}$'s computation will not start on Stage S_1 until tile $\tau_{\lceil \frac{x}{n} \rceil, \lceil \frac{x}{n} \rceil} = \tau_{\delta,\delta} = \tau_{[\delta(2\delta+1)+\delta]}$ (where $\delta = \lceil \frac{x}{n} \rceil$) has been received completely. As each tile is of size n^2 pixels, the amount of data needed to process tile τ_0 is $[\delta(2\delta+1)+\delta+1]n^2$. For the example shown in Figure 6.22 the 12 tiles (shaded) tiles need to be received by S_1 before it can process tile τ_0. However the amount of data needed to be received by Stage S_0 to process the tile $\tau_{i,j}$ for all $i = 0$ and $j > 0$ is only $(2\delta+1)n^2$ pixels. The mount of data needed to be received by Stage S_0 to process the remaining tiles is n^2. This is because the neighborhood

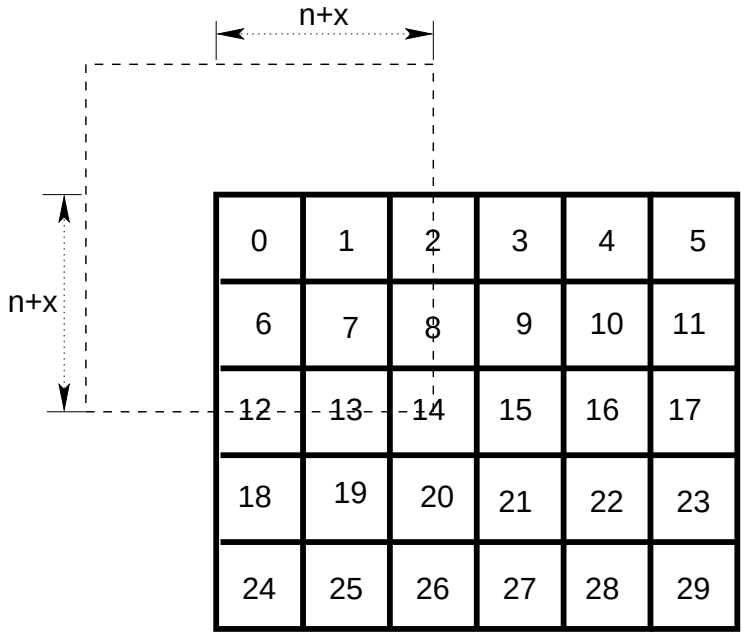


Figure 6.21: Tile only input for row major ordering

required to process these tiles has already been brought in, in the previous iterations.

Theorem 38. The number of pixels input for the tile $\tau_{r,c} = \tau_k$ at Stage S_0 of the pipeline in Row Major tile ordering for the tile only input format is

$$\eta_k^D = \begin{cases} [\delta(2\delta+1)+\delta+1]\,n^2 & \text{if } r,c=0 \text{ or } k=0; \\ (2\delta+1)\,n^2 & \text{if } r,c>0 \text{ or } k>0\ ; \\ n^2 & \text{if } r,c>0; \\ 0 & \text{if } k>(\xi^2-1)-[\delta(2\delta+1)+\delta]. \end{cases}$$

■

Notice that $td_{0,0}$ here is independent of N. In contrast, $t_{0,0} \geq xN$ for the row major ordering. Considering that among the times $t_{k,0}$, $t_{0,0}$ is the only one that matter for computationally bottlenecked case (see Chapter 8). The diagonal ordering makes a big difference to the performance.

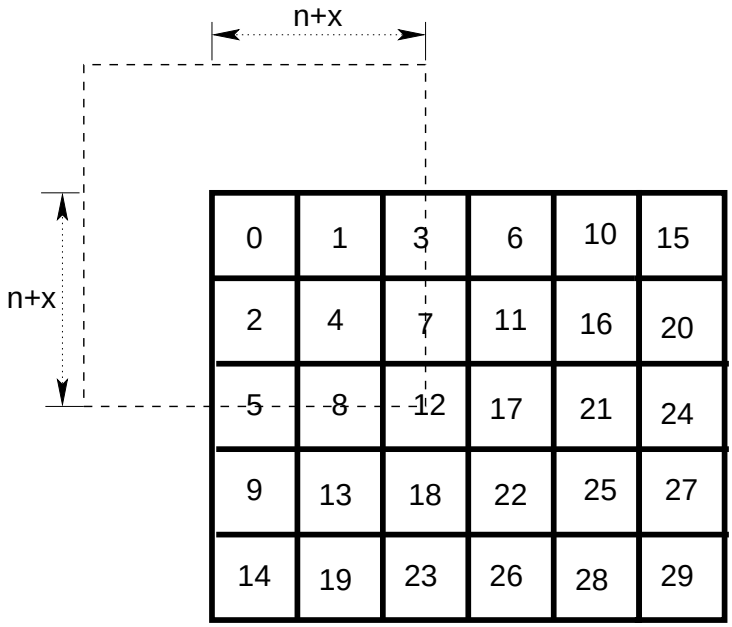


Figure 6.22: Tile only input for diagonal ordering

Chapter 7

Single-Chip Uniprocessor

In this chapter we study the performance of the SIFT algorithm in terms of its time complexity and memory requirement on a single chip processing pipeline (see Figure 7.1), where the computing platform consists of a single CPU. This is the simplest case and its study lays the foundation for the single-chip multicore case (Chapter 8) and the two-chip pipeline (Chapter 9). Its also provides a basis to deal with an n-chip pipeline. As discussed in Chapter 4, the order in which image tiles are input to the chip is significant for the performance of SIFT.

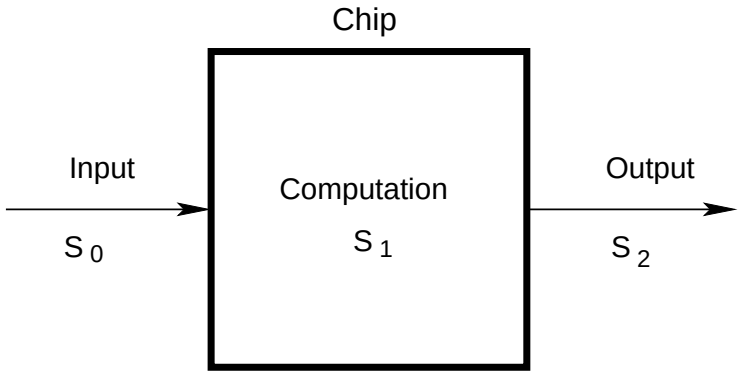


Figure 7.1: The 3-stage pipeline model

As discussed in Chapter 5, the single-chip model results in a 3-stage pipeline consisting of the *input stage*, *compute stage* and *output stage*. These stages are denoted by S_0, S_1 and S_2 according to the notation discussed in Section 5.1. The tiles enter the pipeline at Stage S_0, the processor performs its computation in Stage S_1 and the output is delivered

through Stage S_2.

7.1 Running Time on a 3-Stage Pipeline

Recall that the given image of size $N \times N$ is decomposed into a $\xi \times \xi$tile array (where $\xi = \frac{N}{n}$), so there are ξ^2 tiles numbered as $0, 1, \cdots, \xi^2 - 1$ in the image. In Section 5.1.1, we stated that in a $(2c+1)$-stage totally ordered pipeline with maximal stage S_m and for which $S_{m+1} \succeq S_{m+2} \succeq \cdots \succeq S_{2c}$, the total time to process all tiles is

$$T = \sum_{v=0}^{m-1} t_{0,v} + \sum_{u=0}^{\xi^2-1} t_{u,m} + \sum_{v=m+1}^{2c} t_{\xi^2-1,v} \tag{7.1}$$

Now we use this to derive the expression for the overall time of the pipeline. For $0 \leq k < \xi^2$, the time to process tile τ_k in stage S_ℓ is $t_{k,\ell}$ ($t_{k,0}, t_{k,1}$ and $t_{k,2}$ in the 3-stage pipeline). The overall running time depends on where the maximal stage is located in the pipeline.

Case 1 $S_0 \preceq S_1 \preceq S_2$ **:** In this case the maximal stage is stage S_2. From Equation (7.1)

$$T = \sum_{v=0}^{1} t_{0,v} + \sum_{u=0}^{\xi^2-1} t_{u,2} = t_{0,0} + t_{0,1} + \sum_{u=0}^{\xi^2-1} t_{u,2}$$

Case 2 $S_0 \succeq S_1 \succeq S_2$**:** In this case the maximal stage is stage S_0. Again from Equation (7.1)

$$T = \sum_{u=0}^{\xi^2-1} t_{u,0} + \sum_{v=1}^{2} t_{\xi^2-1,v} = \left(\sum_{u=0}^{\xi^2-1} t_{u,0} \right) + t_{\xi^2-1,1} + t_{\xi^2-1,2}$$

Case 3 $S_0 \preceq S_1 \succeq S_2$**:** In this case the maximal stage is stage S_1. Again from Equation (7.1)

$$T = t_{0,0} + \left(\sum_{u=0}^{\xi^2-1} t_{u,1} \right) + t_{\xi^2-1,2}$$

Case 4 $S_0 \succeq S_1 \preceq S_2$**:** Here either S_0 or S_2 is maximal. So either Case 1 or Case 2 applies.

We now consider the time for the stages.

7.2 Time Complexity of Stages

In this section we derive the time complexities of the stages in the 3-stage pipeline. namely for the *input*, *compute* and *output* stages. As mentioned in Section 5.1.1, the time taken by Stage S_ℓ to complete its process on tile τ_k is denoted by $t_{k,\ell}$.

Input Stage Time Complexity: The input stage time complexity is the time ($t_{k,0}$) taken by Stage S_0 to make the received data for tile τ_k available to the next stage S_1. In Chapter 6, we discussed details of the amount of input data received at stage S_0 for the two tile orderings. Let p_i be the number of input pins to the chip at stage S_0, let b be the number of bits in each pixel of the image and let Γ_0 be the clock rate for Stage S_0. Let tile ordering $\circledcirc$ be the used (where $\circledcirc \in \{\text{R,D}\}$) and let there be the $\eta_k^{\circledcirc}$ pixels coming into S_0 at iteration k. Then, the time taken by Stage S_0 to receive the input data and make it available to the next stage is

$$t_{k,0} = \left\lceil \frac{\eta_k^{\circledcirc} \cdot b}{p_i} \right\rceil \Gamma_0 \tag{7.2}$$

For the worst case $\eta_k^{\circledcirc} = (n+x)^2$ (see Theorem 25). So $t_{k,0} \leq \left\lceil \dfrac{(n+x)^2 \cdot b}{p_i} \right\rceil \Gamma_0$

Compute Stage Time Complexity The compute stage complexity is the time taken by Stage S_1 to extract features from the tile received from the previous stage and deliver features to the next output stage. This time for tile τ_k is denoted by $t_{k,1}$. The computations performed by Stage S_1 includes the phases Gaussian blurring, difference of Gaussians, extrema detection, potential keypoints detection and generation of keypoint descriptors (explained in Sections 2.2 – 2.5). Table 2.1 shows the time complexities and the number of operations required by each phase of the SIFT algorithm for an N^2-pixel image. Applying these to an n^2-pixel tile and then substituting the values of $\alpha = 0.6\%, \beta = 35\%$ and $\gamma = 0.04\%$ values (from Section 3.3) gives the time complexity (number of operations) of the computation stage as shown in Table 7.1.

Therefore with Γ_1 as the clock rate for the computation stage,
$t_{k,1} = (4n^2w^2s + 4n^2s + 104sn^2 + 0.6sn^2 + 48sn^2 + 12.03x^2n^2) \cdot b \cdot \Gamma_1$
$= (4w^2s + 157s + 12.03x^2)\, n^2 \cdot b \cdot \Gamma_1$. The $12.03x^2n^2$ term is the dominating term.

Table 7.1: The number of SIFT operations for an n^2 pixel tiles

Phase	Number of operations
Gaussian Blurring	$4n^2w^2s$
Difference of Gaussian	$4n^2s$
Scale-space Extrema Detection	$104sn^2$
Keypoint Detection	$0.6sn^2$
Orientation Assignment	$48sn^2$
Keypoint Descriptor Generation	$12.03x^2n^2$

Output Stage Time Complexity The output stage time complexity, $t_{k,2}$ (for τ_k) is the time taken by Stage S_2 to output the tile features. The size of each feature is $(2\log x + 1)$ bits. If Γ_2 represents clock rate of the output stage, p_o denotes the number of output pins of the chip, then the time taken by stage S_2 is nominally $\left\lceil \frac{(2\log x+1)}{p_o} \right\rceil \cdot \Gamma_2 \cdot \left[(\alpha\beta+\gamma)\, n^2\right]$.

In general $b = 32$ bits , $x \cong 8, w \cong 3, s \cong 2$ as used in Lowe's [13] algorithm. Here we assume $p_i = p_o = 320$. Modern chips such as FPGA can have hundreds of input output pins. So

$$t_{k,0} = \left\lceil \frac{(n+x)^2 \cdot b}{p_i} \right\rceil \cdot \ \Gamma_0 = \left\lceil \frac{(n+8)^2 \cdot 32}{320} \right\rceil \cdot \Gamma_0 = \frac{(n^2+64+16n)}{10} \cdot \Gamma_0$$

$$t_{k,1} = \left(4w^2s + 157s + 0.084x^2\right) n^2 \cdot b \cdot \Gamma_1 \cong (400)\, n^2 \cdot 32 \cdot \Gamma_1 = 36928n^2 \cdot \Gamma_1$$

$$t_{k,2} = \left\lceil \frac{(2\log x+1)}{320} \right\rceil \cdot \Gamma_2 \cdot \left[(\alpha\beta+\gamma)\, n^2\right]. = 1 \cdot \left[(\alpha\beta+\gamma)\, n^2\right] = 0.0015n^2\Gamma_2$$

So the case of $S_0 \preceq S_1 \succeq S_2$ applies. The time complexity of running the SIFT algorithm on a 1-chip (3-stage) uniprocessor pipeline is (from Equation (7.1))

$$T = t_{0,0} + \left(\sum_{u=0}^{\xi^2-1} t_{u,1} \right) + t_{\xi^2-1,2}.$$

Therefore $T = \Theta\left(\frac{(n+x)^2 \cdot b \cdot \Gamma_0}{p_i} + \frac{\log x \cdot \Gamma_2}{p_o} + N^2 \cdot \Gamma_1\right)$. The last term is independent of n so to reduce the overall time n can be selected to be as small as possible. While maintaining the relationship $S_0 \preceq S_1 \succeq S_2$, we could also adjust p_i, Γ_0 and p_o, Γ_2. Clearly $p_i, p_o \geq 1$. They are likely to be much larger. Modern chips are available with around 1000 pins.

For example suppose $N^2 = 10^6$ for a 1000×1000 image. The total time the algorithm spends

on the stages is about $S_0 \cong \frac{10^6 \times 32 \cdot \Gamma_0}{320} = 100000\ \Gamma_0$ for Stage S_0; about $S_1 \cong 1154 \times 10^6\ \Gamma_1$ for Stage S_1 and about $S_2 \cong 1500\ \Gamma_2$ for Stage S_2. By slowing Γ_0 and Γ_2 proportionally to Γ_1 power savings are possible. So one could make Γ_0 11540 times slower than Γ_1 and Γ_2 769×10^3 times slower than Γ_1 without changing the maximal stage. Of course this would increase the times for $t_{0,0}$ and $t_{\xi^2-1,2}$.

This may be significant if the input and output stages work over a noisy (say wireless) channel. The clock rates Γ_0, Γ_2 can be adjusted to be substantially smaller than Γ_1. This decreased clock rate reduces the bit error rate (BER) of the input. The reduced clock also reduces power. One could increase the redundancy in the input to again reduce the BER. Many intermediate choices are possible as well whose benefits do not come at the purse of speed.

In Section 5.5, we discuss the processing of multiple images in pipeline model. Even if X images are processed one after the other (that is $X\xi^2$ tiles), S_1 is still the maximal stage and $T = t_{0,0} + X\left(\sum_{u=0}^{\xi^2-1} t_{u,1}\right) + t_{\xi^2-1,2}$

Theorem 39. In a 3-stage totally ordered pipeline, for all $0 \leq k \leq \xi^2 - 1$, S_1 is being the maximal stage, the time taken by the pipeline to process all $X \geq 1$ images one after the other is

$$T = \Theta\left(\frac{(n+x)^2 \cdot b \cdot \Gamma_0}{p_i} + \frac{\log x \cdot \Gamma_2}{p_o} + XN^2 \cdot b \cdot \Gamma_1\right)$$

7.3 Memory Requirement on a 3-Stage Pipeline

The amount of memory required in the chip is discussed in Section 5.2. Since the output of SIFT is considerably smaller than the input, the memory requirement is driven by the input stage. So here the memory requirement is the difference between the amount of data received so far and the data that is not required further to process the remaining tiles. We use the notation M_k for memory requirement, U_K for data that has been received so far, and V_k for the data that is not needed further (see Section 5.2). Now we consider the two input orderings to determine the memory requirement.

7.3.1 Row Major Ordering

Figure 7.2 shows a tile (bold square) and its neighborhood (hatched). The total area of the image that has been brought in so far is shaded (pink or red). Of this, the portion shaded light (pink) (Area B) is required for the current tile and future tiles. The portion shaded dark (red) (Area A) may be discarded. The memory requirement at this point of the algorithm is Area B = total shaded area − Area A. The determination of the size of this area is done with the help of Figures 7.3, 7.4.

Figure 7.3 shows the total amount of data that has been received so far while processing

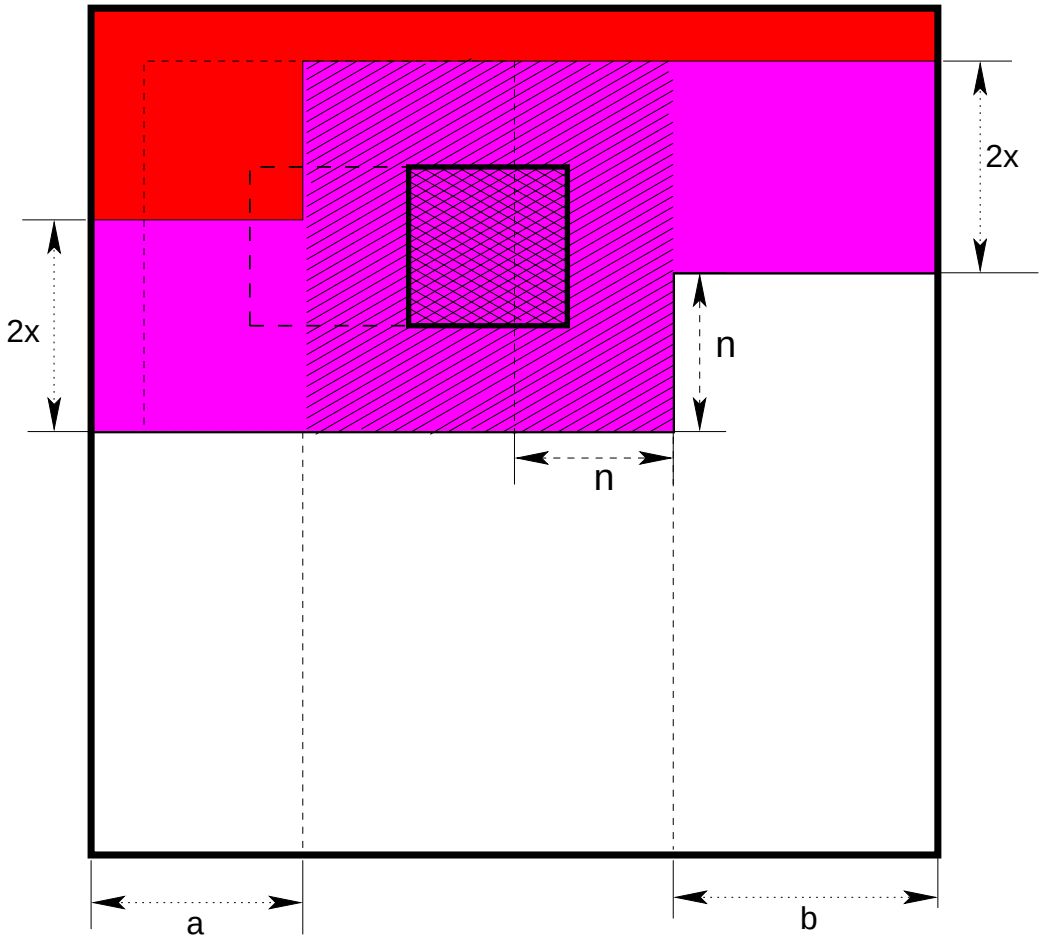


Figure 7.2: Total data received

the current tile $\tau_k = \tau_{i,j}$ (on tile array). The figure shows the four regions labeled $1, 2, 3, 4$. Let $\langle \ell_k \rangle$ denote the area of the region ℓ for tile k, where the context is clear, we will omit k and simply call this area as $\langle \ell \rangle$. The total amount of data received so far before $\tau_{i,j}$ is

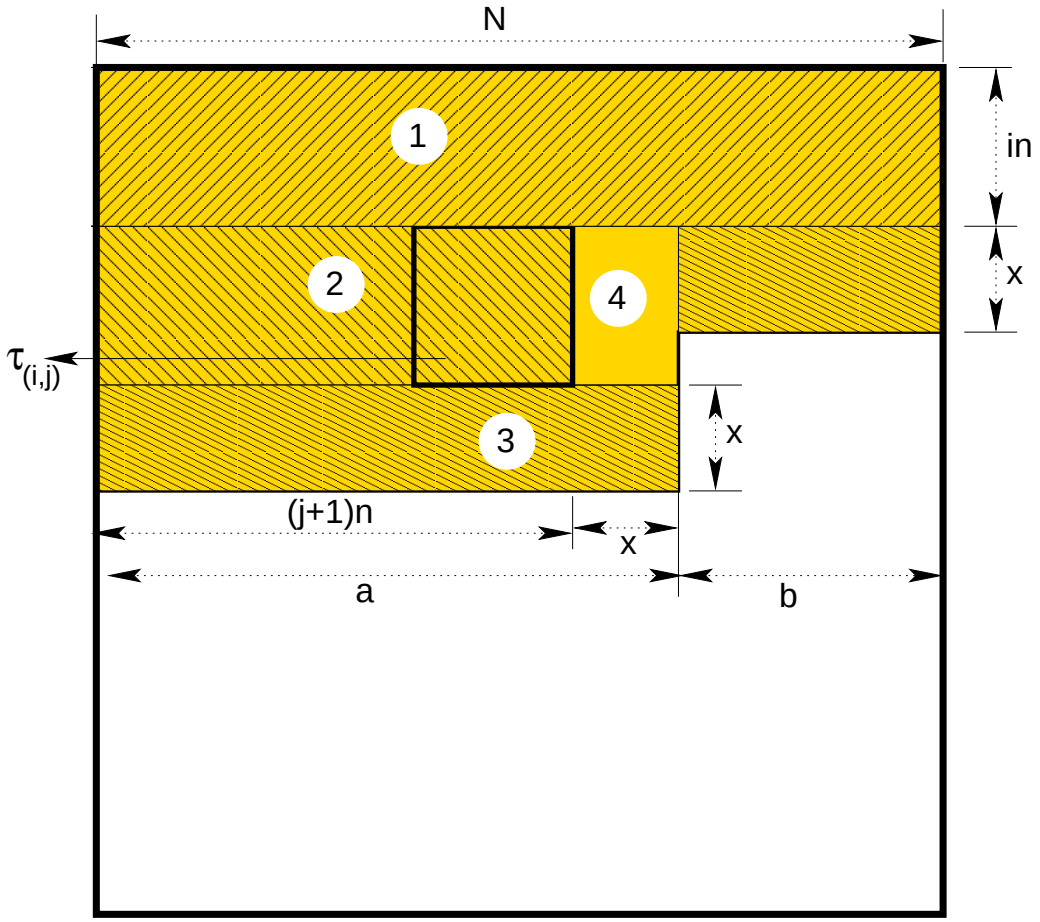


Figure 7.3: Regions of total data received

processed, U_k, is the sum of the areas of regions $1, 2, 3, 4$.That is

$$\begin{aligned} U_k &= \langle 1\rangle + \langle 2\rangle + \langle 3\rangle + \langle 4\rangle = inN + (j+1)n \cdot n + (ax + bx) + nx \\ &= inN + jn^2 + n^2 + Nx + nx \quad \text{since } (a + b = N) \end{aligned}$$

So we have,

$$U_k = inN + jn^2 + n^2 + Nx + nx \tag{7.3}$$

Figure 7.4 shows the amount of data that is not needed further to process the remaining tiles after the tile $\tau_k = \tau_{i,j}$. This is equal to the sum of the areas of regions $\langle 1\rangle$, $\langle 2\rangle$.

$$\begin{aligned} V_k &= \langle 1\rangle + \langle 2\rangle = (in - x)\, N + (jn - x)\, n \\ &= inN + Nx + jn^2 + nx \end{aligned}$$

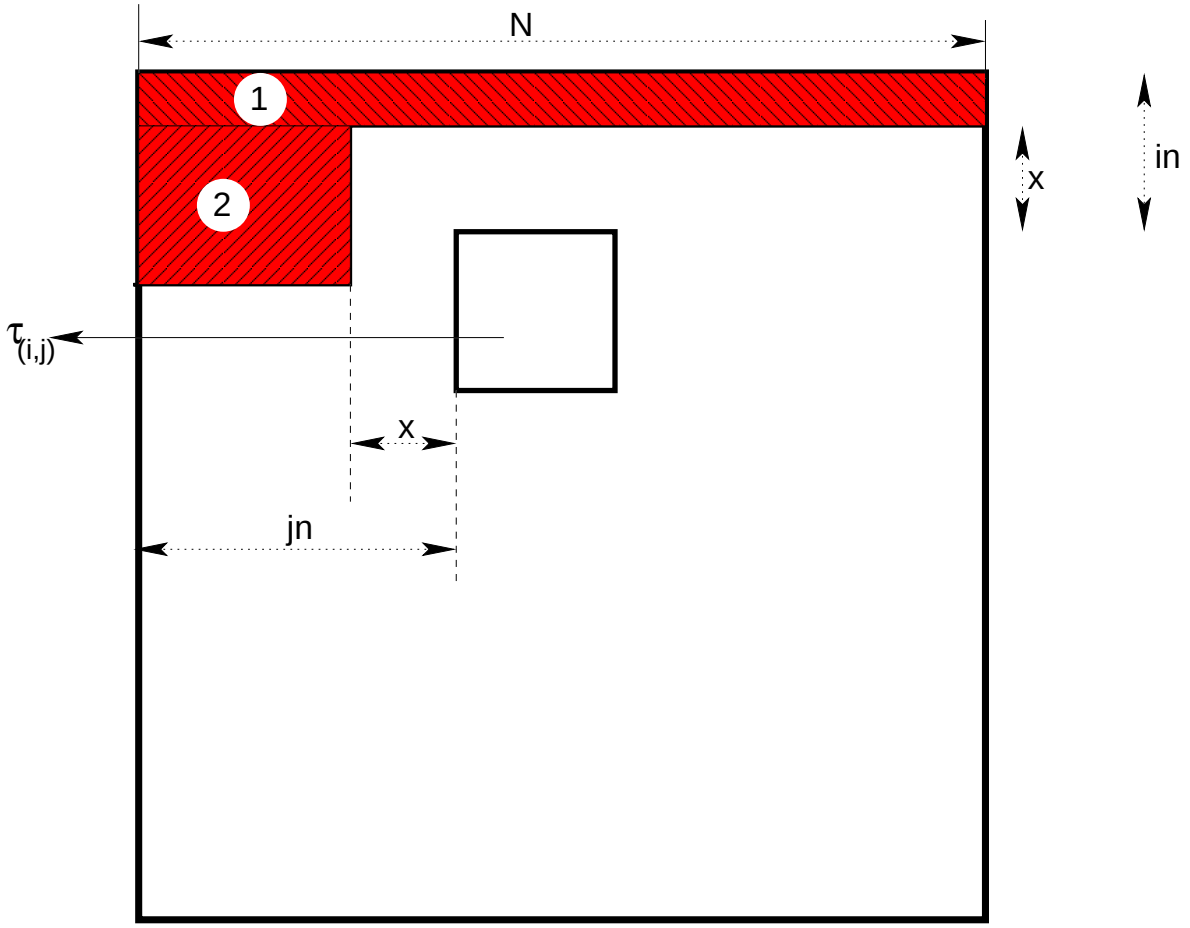


Figure 7.4: Data that is not needed further

Thus we have

$$V_k = inN + Nx + jn^2 + nx \tag{7.4}$$

Now, The amount of memory needed to process the remaining tiles is given by Equation (7.3) − Equation (7.4). Thus

$$\begin{aligned} \text{Amount of memory needed} &= U_k - V_K \\ M_k &= n^2 + 2nx + 2Nx. \end{aligned}$$

Notice that M_k is independent of k. Except for the very beginning and end of the image the algorithm will require $M_k = n^2 + 2nx + 2Nx$ of memory to store pixels to be used at a later time.

Theorem 40. The amount of memory required to run the SIFT algorithm on a 1-chip (3-stage) uniprocessor pipeline for row major ordering is $n^2 + 2nx + 2Nx$.

7.4 Diagonal Ordering

In this section we study the amount of memory required in the chip. It is the difference between the amount of data received so far and the data that is not required further to process the remaining tiles. In Figure 7.5, the total area of the image that has been brought in so far is shaded (pink or red). Of this the portion shaded light (pink) (Area B) is required for the current tile and future tiles. The portion shaded dark (red) (Area A)may be discarded. The memory requirement at this point of the algorithm Area B = total shaded area − Area A. The evaluation of the size of this memory needed is calculated with the help of Figures 7.9, 7.10.

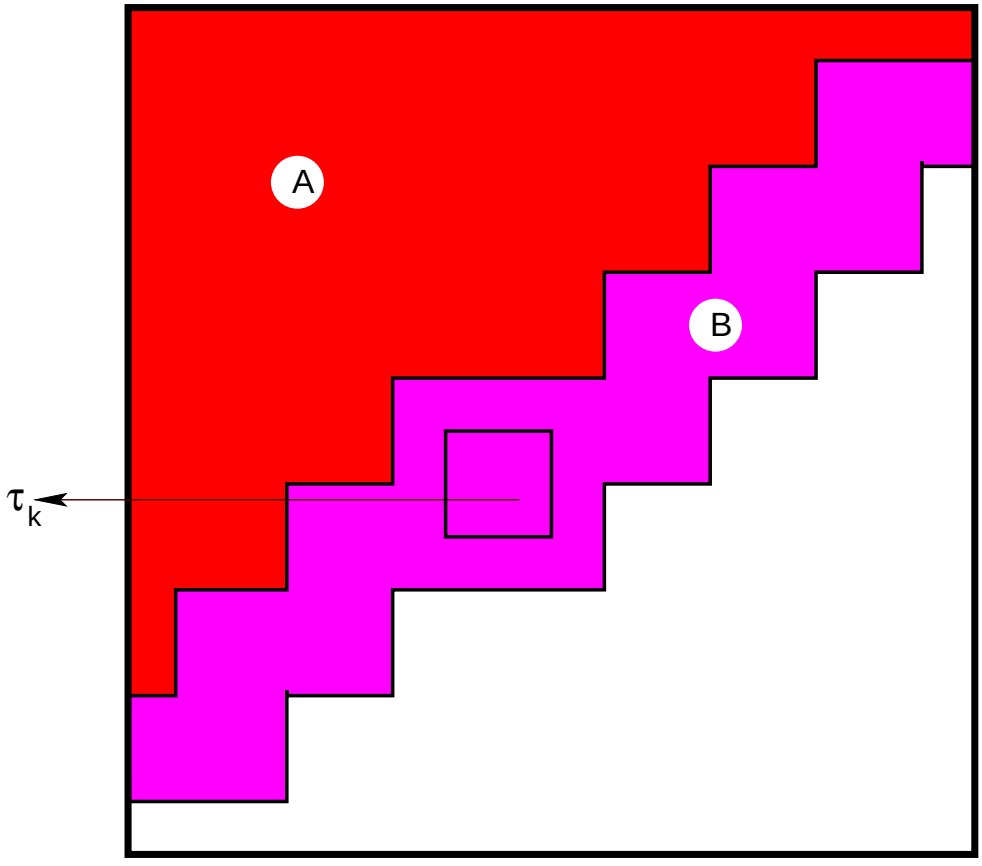


Figure 7.5: Memory requirement for Diagonal Ordering

Figure 7.9 shows the total amount of data that has been received so far while processing the current tile $\tau_k = \tau_{i,j}$ (on tile array). The figure shows the two regions labelled $1, 2$. Let ℓ_k denote the area of the region ℓ for tile k, where the content is clear, we will omit k and simply call this area as $\langle \ell \rangle$.

We now define some additional quantities and develop some intermediate results.

Definition 3. A series in rectilinear space is a contiguous set of adjacent horizontal and vertical lines. If $L_0, L_1, L_2 \cdots L_{u-1}$ are adjacent lines rectilinear series formed by these lines is denoted by L, where $L = \langle L_0, L_1, L_2 \cdots L_{u-1} \rangle$.

For example, Figure 7.6 shows a five segment line.

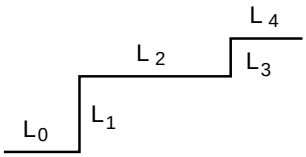


Figure 7.6: $L = \langle L_0, L_1, L_2, L_3, L_4 \rangle$

Definition 4. An x-border of a series L in rectilinear space is the shape defined by two other series $L+$ and $L-$ parallel to L at distances $\pm x$. Figure 7.7 illustrates this.

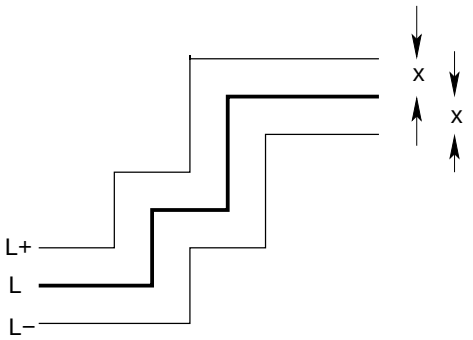


Figure 7.7: x-border of L

Lemma 41. Let $L = \langle L_0, L_1, L_2 \cdots L_{u-1} \rangle$ be a series in rectilinear space whose total length is ℓ. Then for sufficiently small $x > 0$, the area of an x-border of L is $2x\ell$.

Proof: We proceed by induction on u. For $u = 1$, there is only one line segment whose length is ℓ and the area of the x-border is $2x \times \ell = 2x\ell$. Assuming the lemma to hold for any $u \geq 1$, consider a $u + 1$ line segment. That is $L = \langle L_0, L_1, L_2 \cdots L_{u-1}, L_u \rangle$. Let series $\langle L_0, L_1, L_2 \cdots L_{u-1} \rangle$ have the length ℓ' and the series $\langle L_u \rangle$ have the length ℓ'', so the length $\ell = \ell' + \ell''$.

The Area of the x-border of series $\langle L_0, L_1, L_2 \cdots L_{u-1} \rangle = 2x\ell'$ (from the induction hypothesis).

Now consider the area of the x-border of line segment $\langle L_u \rangle = x(\ell'' + x) + x(\ell'' - x) =$

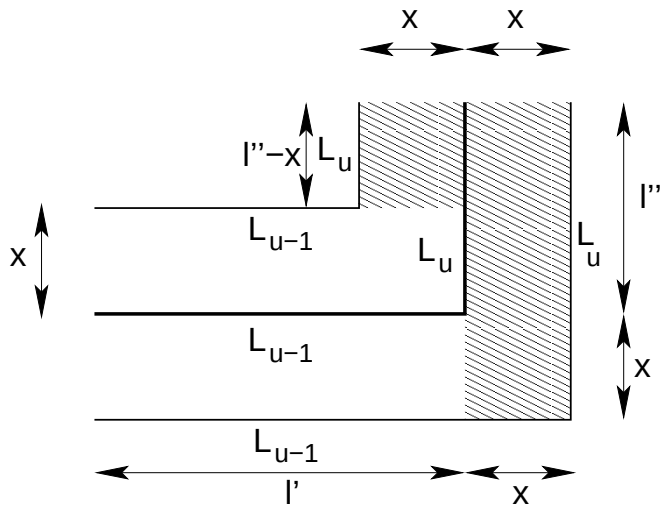


Figure 7.8: Area details of x-border of L

$2x\ell''$ (from Figure 7.8). Overall area of the x-border of line segment
$L = \langle L_0, L_1, L_2 \cdots L_{u-1}, L_u \rangle = 2x\ell' + 2x\ell'' = 2x(\ell' + \ell'') = 2x\ell$. ■

For tile $\tau_{i,j}$ (located anywhere in the tile array), memory requirement (that is proportional to the Area 2 in Figure 7.10) depends on $2(i + j)n$. The value of $(i + j)$ is maximum when the tile $\tau_{i,j}$ is located on the diagonal of the tile array and here $(i + j) = 2(\xi - 1)$. The amount memory required is also maximum for the tiles that are located in the primary diagonal of the tile array, its because that the amount of data brought in is high for these tiles when compared to the remaining tiles. Thus we use this value of $(i + j) = 2(\xi - 1)$ in the determination of memory requirement for diagonal tile ordering.

$$\begin{aligned} U_k &= \langle 1 \rangle + \langle 2 \rangle = (k+1)n^2 + 2\ell x && \text{from Lemma 41} \\ &= (k+1)n^2 + 2\left((i+j)n\right)x = (k+1)n^2 + 2\left(\xi - 1\right)nx && \text{here } i + j = \xi - 1 \end{aligned}$$

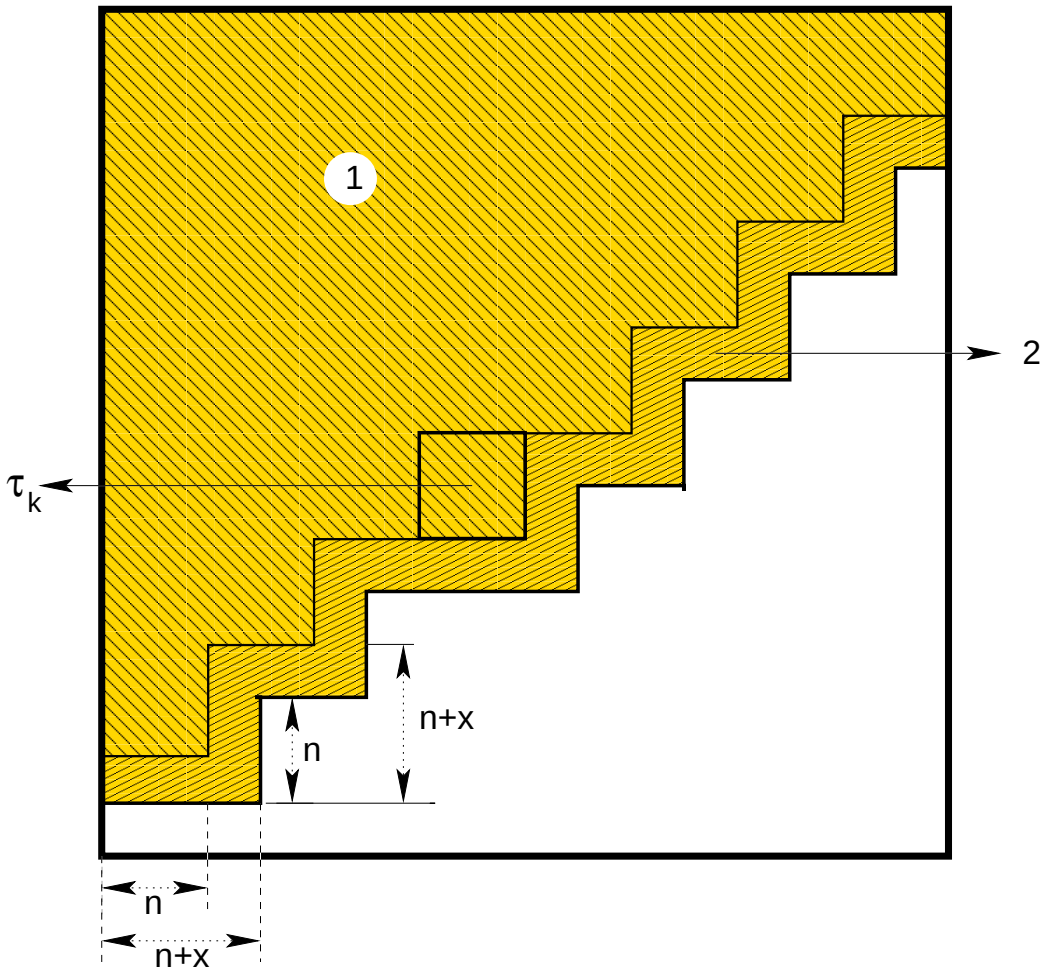


Figure 7.9: The memory requirement for Diagonal Ordering

Thus

$$\text{Total amount of data received} = (k+1)n^2 + 2\,(\xi - 1)\,nx \tag{7.5}$$

In Figure 7.10, the amount of data that is not needed further V_k is shown by the region $\langle 1 \rangle$.

$$\begin{aligned} V_k &= \{\langle 1 \rangle + \langle 2 \rangle + \langle 3 \rangle\} - \{\langle 2 \rangle + \langle 3 \rangle\} = (k+1)n^2 - \{2\,(\xi - 1)\,nx + n^2\} \\ V_k &= (k+1)n^2 - 2\,(\xi - 1)\,nx - n^2 \end{aligned}$$

Thus

$$V_k = (k+1)n^2 s - 2\,(\xi - 1)\,nx - n^2 \tag{7.6}$$

Now, The amount of memory needed M_k is given by the difference of Equation (7.3) and

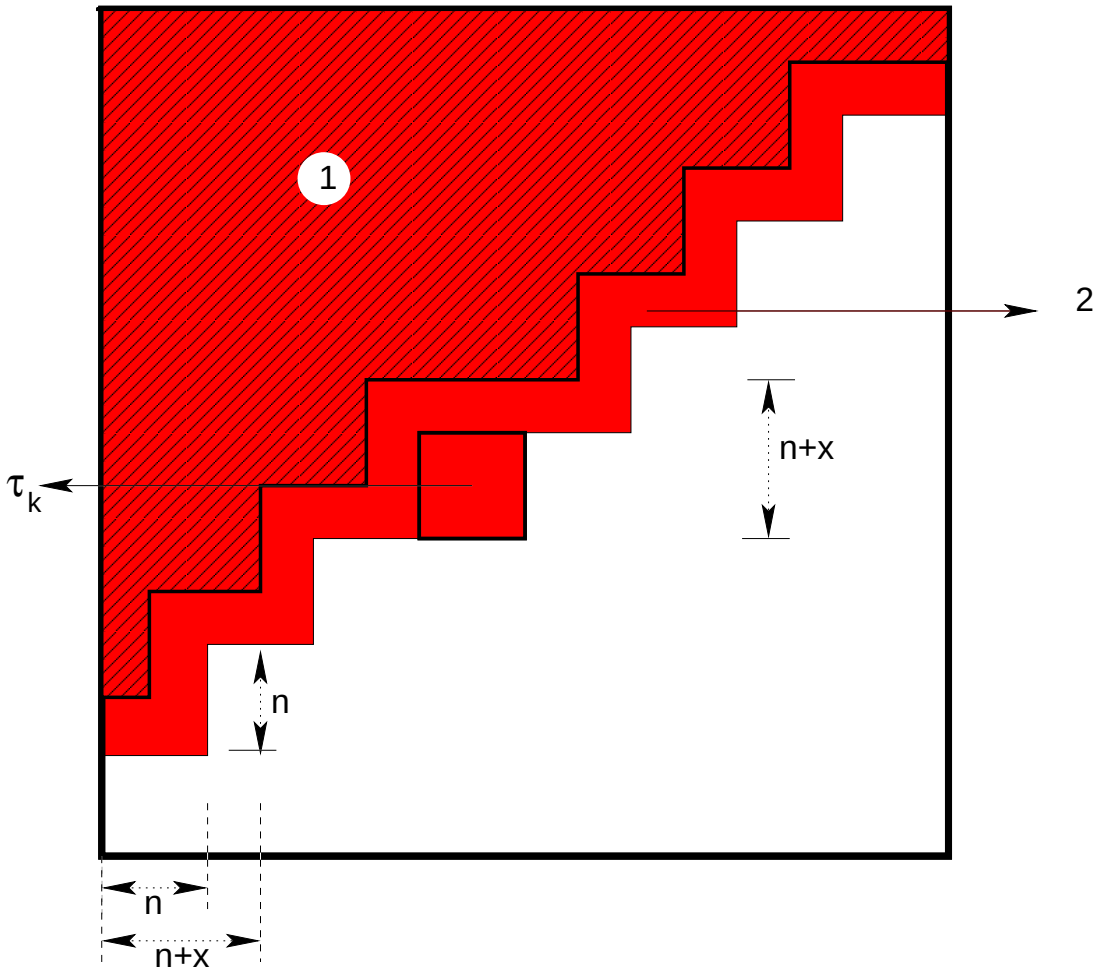


Figure 7.10: The memory requirement for Diagonal Ordering

Equation (7.4).

$$\begin{aligned} M_K &= U_k - V_k = (k+1)n^2 + 2(\xi-1)nx - ((k+1)n^2 - 2(\xi-1)nx) \\ &= 4(\xi-1)nx + n^2 = 4\left(\tfrac{N}{n} - 1\right)nx + n^2 = 4Nx - 4nx + n^2 \end{aligned}$$

Theorem 42. The amount of memory required to run the SIFT algorithm on a 1-chip (3-stage) uniprocessor pipeline for the diagonal ordering is $4Nx - 4nx + n^2$. ■

7.5 Tile Only Input

In this chapter we so far considered the tile-plus-neighborhood input protocol, where the input sends the tile and its neighborhood one after the other. We now examine the time T to run the SIFT on an image on a 1-chip uniprocessor pipeline for tile-only input protocol.

7.5.1 Row Major Ordering

As mentioned earlier in Section 6.2.1,the number of input pixels that are coming in to Stage S_0 of the pipeline for τ_0 is $\eta_k^R = [\delta(\xi+1)+1]\, n^2$. From Section 7.2, the input time complexity for tile $t_{0,0} = \frac{\eta_k^R\ b\ \Gamma_0}{p_i} = \frac{[\delta(\xi+1)+1]n^2 \cdot b \cdot \Gamma_0}{p_i} = \Theta(xN)$. Clearly, the value of $t_{0,0}$ depends on the value of N (as $\xi = \frac{N}{n}$) and the compute stage complexity is proportional to the value of n^2. So $t_{0,0} \geq t_{k,1}$.

Even though $t_{0,0} \geq t_{k,1}$, it is clear that once S_1 start processing of τ_0, there is no stopping of S_1. So the complexity of $t_{0,0} = \Theta(xN)$. This could be significantly larger than $t_{k,1} = \Theta(n^2)$. However from Theorem 37, for tile-only input $t_{k,0} \leq t_{k,1}$ for all $k > 0$. The condition for $S_0 \preceq S_1$ requires that for all $k \geq 0$ $T_{k,1} + t_{k,1} \geq T_{k+1,0} + t_{k+1,0}$. Thus $t_{0,0}$ is not required to be $\leq t_{k,1}$. That is $S_0 \preceq S_1$ and Theorem 14 holds. Therefore we have

Theorem 43. For all $0 \leq k \leq \xi^2 - 1$, the total time to run the SIFT algorithm on a 1-chip (3-stage) pipeline on an image of size $N \times N$ using row major ordering and tile-only protocol is: $T = \Theta(xN) + \left(\sum_{u=0}^{\xi^2-1} t_{u,1}\right) + t_{\xi^2-1,2}$ ∎

Remark: Notice that the tile-only contribution of time of Stage S_0 is due to $t_{0,0}$. Because $t_{0,0}$ in this case is large there is a significant degradation in time even though $t_{k,0}$ is small for all $k > 0$.

7.5.2 Diagonal Ordering

As mentioned earlier in Section 6.2.2, the number of input pixels that are coming into Stage S_0 of the pipeline for τ_0 is $\eta_k^D = [\delta\,(2\delta+1) + \delta + 1]\, n^2$. From Section 7.2, the input time complexity for tile $t_{0,0} = \frac{\eta_k^D\ b\ \Gamma_0}{p_i} = \frac{[\delta(2\delta+1)+\delta+1]n^2 \cdot b \cdot \Gamma_0}{p_i}$. Here the value of $t_{0,0}$does not depends on the value of N and $t_{0,0} = \Theta(x^2)$.

Here $S_0 \preceq S_1$, so the time complexity of the entire algorithm is better than for the row major ordering case.

Theorem 44. For all $0 \leq k \leq \xi^2 - 1$, the total time to run the SIFT algorithm on a 1-chip (3-stage) pipeline on an image of size $N \times N$ using the diagonal ordering and the tile-only protocol is: $T = \Theta(x^2) + \left(\sum_{u=0}^{\xi^2-1} t_{u,1} \right) + t_{\xi^2-1,2}$. ■

In the next chapter we extend these ideas to study a single-chip multicore pipeline model.

Chapter 8

Single-Chip, Multicore Processor

In this chapter we study the performance of the SIFT algorithm, again on a 1-chip (3-stage) pipeline, but this time with a multicore computing platform. We model the multicore platform using the hierarchical multi-level-caching model (HM model) [4].

In the next section we briefly describe the HM model. In section 8.2 we discuss the mapping of tiles to the cores and in Section 8.3 we derive the expressions for time to run SIFT on a multicore platform.

8.1 The Hierarchical Multi-Level-Caching (HM) Model

Modern multicore architectures use multiple processor cores that share a memory hierarchy (for example, the Intel Xeon Multiprocessor [9]). The Hierarchical Multi-Level-Memory model (HM model) [4] captures this structure in a general architecture that abstracts away details of particular chips and interconnects, but without ignoring memory access costs. Let the chip here be $P > 1$ processor cores numbered $0, 1, 2, \cdots, P-1$. Let the chip contain h levels of caches, numbered $1, 2, \cdots, h$ (see Figure 8.1). For $1 \leq 0 \leq h$, an L_i-cache refers to a Level-i cache. While a full description of the cache structure appears in Chowdhury *et al.* [4], we detail only relevant parts here. The caches are arranged in a tree-like hierarchy. Let the number of L_{i-1}-caches connected to an L_i-cache be s_i; if $i = 1$ then the L_{i-1}-cache is replaced by a processor core. As in Chowdhury *et al.*, we assume that $s_1 = 1$; that is, each processor core has its own private L_1-cache.

It is assumed that there is only one module of the highest level cache (L_h-cache). The L_h-

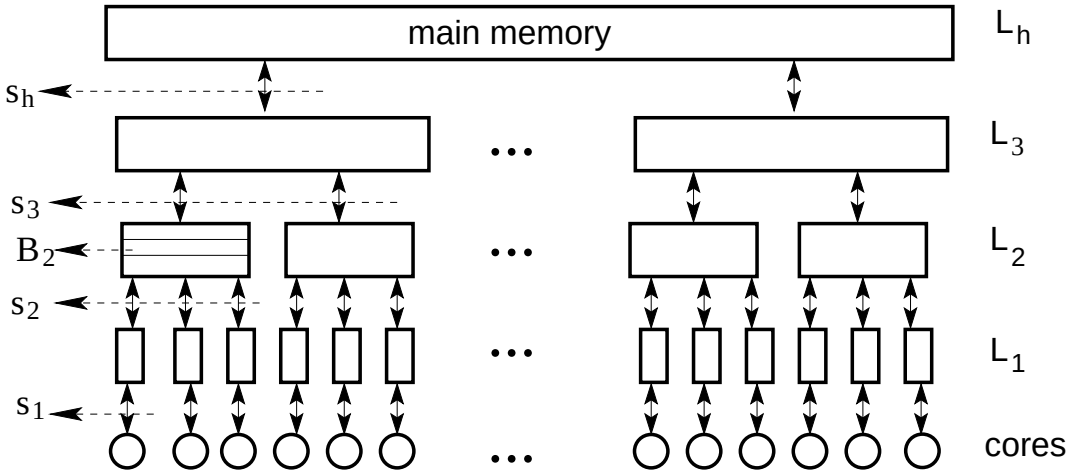


Figure 8.1: Hierarchical Multi-Level-Cache (HM) Model

cache interfaces to the input/output system outside the chip. For any $i < h$, the number of L_i-cache modules is $\mathcal{R}_i = s_h, s_{h-1}, \cdots, s_{i+1}$. The shadow of a cache is the number of processor cores under it in the hierarchy. There is $s_1 = 1$ core under the shadow of each L_1-cache, $s_1 s_2$ cores under the shadow of each L_2-cache, $s_1 s_2 s_3$ cores under the shadow of each L_3-cache, and so on. Let $\mathcal{Q}_i = s_1 s_2 \cdots s_i$ denote the size of shadow of L_i-cache. In this notation $\mathcal{R}_i \mathcal{Q}_i = P$, the number of cores. Let each L_1-cache have a line size of B_i pixels; we express this in pixels to facilitate our analysis. The actual line size can be obtained by multiplying B_i by the size of a pixel. If for all levels i, $s_i \leq \frac{B_i}{B_{i-1}}$, we will call it an *inclusive large-line cache*, that is, al line from each of $_i$ subcache at the L_{i-1} level, will fit in a single L_i-cache line. If $s_i = s$, for all $i$we will call the hierarchy *uniform*. In subsequent discussion we will consider hierarchies that are inclusive large-line and uniform.

Each processor core u $(0 \leq u < P)$ accesses data as follows. It first looks for the data in its L_1-cache. If the data is found, then there is no penalty on the access. Otherwise, there is an L_1 miss. This causes an L_2 access. An L_2 miss causes an L_3 access and so on. For our application (as we will show later), we consider accesses with misses all the way to the L_h-cache. When an L_i miss triggers an L_{i+1} access, this access competes with all other L_i misses under that L_{i+1} module. These accesses to the L_{i+1} module are all sequential, so the complexity of these accesses is the number of accesses to the L_{i+1} module. However different L_{i+1}-cache modules can be accessed in parallel. Thus the cache complexity can be viewed as the maximum number of accesses to any L_i-cache module, summed over all levels. More specifically, if A_i is the maximum number of accesses to an L_i-cache module, then the cache access complexity is $O\left(\sum_{i=0}^{h} A_i\right)$. We also note that the access to an L_i-cache

equals the number of misses at all s_i subcaches at level $i-1$.

8.2 Mapping Tile Data to Cores

As described in Section 8.1, the last level L_h-cache interfaces to outside the chip. Thus in our 3-stage pipeline, the input stage brings in pixels to the L_h-cache. These pixels are ultimately mapped to particular processor cores. In this section we describe this mapping.

Let $\frac{n}{P} \geq 1$ be an integer, where $n \times n$ is the tile size. Divide the $n \times n$ tile into P subtiles each of size $n \times \frac{n}{P}$. The u^{th} subtile is mapped to core u (where $0 \leq u < P$) (see Figures 8.2, 8.3). In this scheme, column j (where $0 \leq j \leq N$) of the image is mapped to core u iff $\lceil \frac{jP}{n} \rceil = u$. Since $\frac{n}{P}$ is an integer, column j (where $0 \leq j \leq n$) of a tile is also mapped to core u iff $\lceil \frac{jP}{n} \rceil = u$. We will refer to data mapped to core u as *local data*. As data arrives in the L_h-caches, they are all first moved to the L_1 caches of the appropriate cores (recall that each core has a private L_1-cache). Subsequent references to data that is local to a different core will cause cache misses.

In the SIFT algorithm, the amount of data that comes into the cache, the times when it comes in and the computation for which its needed are all known in advance. Therefore, we assume that data that is no longer required for the algorithm is automatically swapped out to make room for other data. Thus, each cache has a miss only for the first time a piece of data is needed.

8.3 Computation Stage S_1 in the HM Model

As each $n \times n$ tile is input to the multicore chip, it enters at the highest level cache, the L_h-cache. The first step of the computation stage is for each core to access its local pixels. Core u accesses all columns j of the tile such that $u = \lceil \frac{jP}{n} \rceil$ (see Figure 8.2).

At this point each L_1-cache holds the local data of a subtile corresponding to a core. But before the core can apply SIFT on its local data, it needs the x neighborhood of its local data and some of this may be local to other cores. The second step is to get this neighborhood data. After this step each core has all data needed to independently process its local subtile.

We now describe these phases below.

8.3.1 Accessing Local Data

We examine how each of P processors accesses its local data from the $n \times n$ tile located in the L_h-cache. See Figure 8.2 for the mapping of a subtile to a processor. Each core needs to access $\frac{n^2}{P}$ pixels of local data. Let the L_1-cache have a cache-line length of B_1. Then the local data will require $\left\lceil \frac{n^2}{B_1 P} \right\rceil$ cache lines of storage at the L_1 level. Since we do not have any constraints on how cache lines are organized in the hierarchy, we may assume that each L_i-cache line contains $R_i = \left\lceil \frac{B_i}{B_{i-1}} \right\rceil$, L_{i-1} lines. That is, data for lower level caches are compactly placed within the upper level cache lines as needed. This structure is possible as local data is exclusive and the hierarchy is a tree. The total number of accesses due to any one processor at level L_1 is $A_1 = \left\lceil \frac{n^2}{B_1 P} \right\rceil$. The number of accesses at level L_2 is $A_2 \leq s_2 \left\lceil \frac{A_1}{R_2} \right\rceil \leq s_2 \left(\left\lceil \frac{A_1 B_1}{B_2} \right\rceil \right) \leq s_2 \left(\frac{A_1 B_1}{B_2} + 1 \right)$. Similarly for any level L_i-cache,

$$A_i \leq s_i \left\lceil \frac{A_{i-1}}{R_i} \right\rceil \leq s_i \left(\left\lceil \frac{A_{i-1} B_{i-1}}{B_i} \right\rceil \right) \leq s_i \left(\frac{A_{i-1} B_{i-1}}{B_i} + 1 \right) \tag{8.1}$$

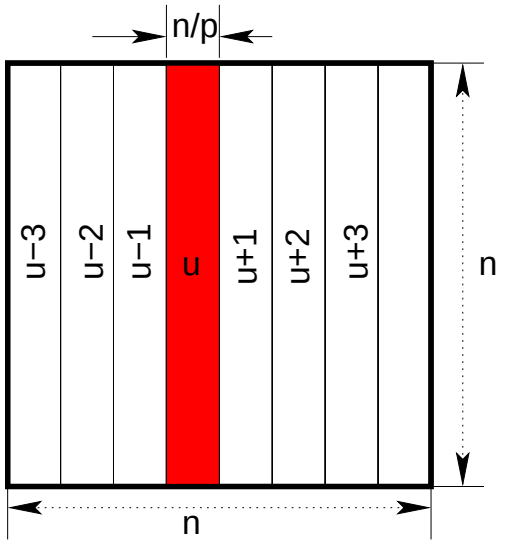


Figure 8.2: Accessing local data

Lemma 45. For $i \geq 2$, the number of accesses required at the level-i in the hierarchy is $A_i \leq \frac{\mathcal{Q}_i A_1 B_1}{B_i} + \frac{\mathcal{Q}_i}{B_i} \sum_{j=2}^{i} \frac{B_j}{\mathcal{Q}_{j-1}}$.

Proof: Recall that $\mathcal{Q}_i = s_1 s_2 s_3 \cdots s_i$. We proceed by induction on $i \geq 2$. For $i = 2$, $A_2 \leq s_2 \left(\left\lceil \frac{A_1 B_1}{B_2} \right\rceil \right) \leq s_2 \left(\frac{A_1 B_1}{B_2} + 1 \right)$. Since $s_1 = 1$ and $\mathcal{Q}_1 = 1$, $A_2 \leq \frac{s_2 A_1 B_1}{B_2} + s_2 \leq \frac{s_1 s_2 A_1 B_1}{B_2} + \frac{s_1 s_2}{s_1} \leq \frac{Q_2 A_1 B_1}{B_2} + \frac{Q_2}{Q_1}\frac{B_2}{B_2}$. This is in the form stated in the Lemma.
Assume the lemma to hold for any $i \geq 2$ and consider A_{i+1}.
$A_{i+1} \leq s_{i+1} \left(\frac{A_i B_i}{B_{i+1}} + 1 \right) \leq s_{i+1} \left(\frac{B_i}{B_{i+1}} \left[\frac{\mathcal{Q}_i A_1 B_1}{B_i} + \frac{\mathcal{Q}_i}{B_i} \sum_{j=2}^{i} \frac{B_j}{\mathcal{Q}_{j-2}} \right] + 1 \right)$ (from the induction hypothesis).
Therefore, $A_{i+1} \leq s_{i+1} \frac{\mathcal{Q}_i A_1 B_1}{B_{i+1}} + \frac{s_{i+1} \mathcal{Q}_i}{B_{i+1}} \sum_{j=2}^{i} \frac{B_j}{\mathcal{Q}_{j-1}} + s_{i+1} \leq \frac{\mathcal{Q}_{i+1} A_1 B_1}{B_{i+1}} + \frac{s_{i+1} \mathcal{Q}_i}{B_{i+1}} \sum_{j=2}^{i} \frac{B_j}{\mathcal{Q}_{j-1}} + s_{i+1}$.
Notice for $j = i+1$, $\frac{s_{i+1} \mathcal{Q}_i}{B_{i+1}} \frac{B_j}{\mathcal{Q}_{j-1}} = s_{i+1}$. Therefore $A_i \leq \frac{\mathcal{Q}_{i+1} A_1 B_1}{B_{i+1}} + \frac{s_{i+1} \mathcal{Q}_i}{B_{i+1}} \sum_{j=2}^{i+1} \frac{B_j}{\mathcal{Q}_{j-1}}$.

This completes the proof. ■

We now have $A_i \leq \frac{\mathcal{Q}_i A_1 B_1}{B_i} + \frac{\mathcal{Q}}{B_i} \sum_{j=2}^{i} \frac{B_j}{\mathcal{Q}_{j-1}}$. Recall that $P = s_1 s_2 \cdots s_h$, $\mathcal{Q} = s_1 s_2 s_3 \cdots s_i$, $\mathcal{R}_i = s_{i+1} s_{i+2} \cdots s_h$ and $\mathcal{Q}_i \mathcal{R}_i = P$.

Then from Lemma 45 consider the term $\frac{\mathcal{Q}_i}{B_i} \sum_{j=2}^{i} \frac{B_j}{\mathcal{Q}_{j-1}} = \sum_{j=2}^{i} \frac{\mathcal{Q}_i}{\mathcal{Q}_{j-1}} \frac{B_j}{B_i} = \sum_{j=2}^{i} s_j \frac{\mathcal{Q}_i}{\mathcal{Q}_j} \frac{B_j}{B_i}$.

Suppose the hierarchy is inclusive large-line. Then the ratio of cache lines in successive caches levels is much higher than the ration of caches. That is, $\frac{B_i}{B_{i-1}} \geq \frac{\mathcal{Q}_i}{\mathcal{Q}_{i-1}} = s_i$. Based on this assumption we have $\frac{\mathcal{Q}_i}{\mathcal{Q}_j} \frac{B_j}{B_i} \leq 1$ and $\frac{\mathcal{Q}_i}{B_i} \leq 1$. If $s_i = s$, a constant for all i, then $\sum_{j=2}^{h} s_j = is$ and $s^h = P$ so $h = \log_s P$. So Now we have

$$A_i \leq A_1 B_1 + is \tag{8.2}$$

We now derive the time T_L to access local data. Let an L_i-cache require α_i access time. Then the time required for A_i accesses is $T_{L_i} = \alpha_i A_i$. Let $\alpha_m = \max\{\alpha_i : 1 \leq i \leq h\}$; usually $m = h$. The time to access all the local data is $T_L = \sum_{i=1}^{h} T_{Li} = \sum_{i=1}^{h} \alpha_i A_i$

$= \sum_{i=1}^{\ell-1} \alpha_i A_i \leq (A_1 B_1)\alpha_m h + \alpha_m s \sum_{j=2}^{h} i \leq \alpha_m A_1 B_1 \log P + \alpha_m \frac{s}{2}\left(\log^2 P + \log P\right)$. Substituting the value of $A_1 = \left\lceil \frac{n^2}{PB_1} \right\rceil$, $T_L = \alpha_m \left(\frac{n^2}{PB_1} + 1 \right) B_1 \log P + \frac{s}{2}\left(\log^2 P\right) + \alpha_m \frac{s}{2} \log P = \alpha_m \left(\frac{n^2}{P} \log P + B_1 \log P + \frac{s}{2}\left(\log^2 P\right) + \frac{s}{2} \log P \right)$.

$$T_L = O\left(\frac{n^2}{P} \log P\right) \tag{8.3}$$

Notice that since each subtile is of size $\frac{n^2}{P}$, the overhead is only $O(\log P)$

8.3.2 Accessing Neighborhood Pixels

Consider core u again. In processing its subtile (shaded dark (red) in Figure 8.3) core u requires the neighborhood data (shaded light (yellow) in Figure 8.3). In the worst case the entire neighborhood has to be accessed. However the pixels in the core u shaded dark (red) are local and already available to core. The core needs to bring in the pixels shaded in light color (yellow). Clearly, the access is symmetric on either side of u (that is, access is from core $u \pm v$ for some v). So we consider only one side. The total data to be accessed is $x(n + 2x)$ pixels. Let x span σ processors. So, $\sigma = \left\lceil \frac{x}{nP} \right\rceil = \left\lceil \frac{xP}{n} \right\rceil$. That is for one half of the lightly shaded (yellow) region in Figure 8.3, core u gets $\frac{n^2}{P}$ pixels from each of cores $u \pm 1, u \pm 2, \cdots u \pm (\sigma - 1)$ and $\rho = (x \bmod \frac{\mathrm{n}}{\mathrm{P}})$ pixels from core $u \pm \sigma$. This data is in the L_1-cache of these cores. Recall that B_1 is the size of each L_1-cache line. Then core u needs to access a total of $A_1 = (\sigma - 1)\left\lceil \frac{n(n+2x)}{PB_1} \right\rceil + \left\lceil \frac{\rho(n+2x)}{B_1} \right\rceil = (\sigma - 1)D_1 + E_1$ (say) lines. Notice that if x is large, then the E_1 term can be replaced by D_1 without significant loss of accuracy. So $A_1 = \sigma \left\lceil \frac{n(n+2x)}{PB_1} \right\rceil$. Clearly this is the number of misses is well. So these accesses go to an L_2-cache. Each L_2-cache has s_2 number of L_1-caches attached to it and each L_2-cache line holds R_2 lines of L_1-caches. These A_1 accesses are to data in σ, L_1-cache. In fact, these accesses mirror those accesses of Section 8.3.1, except for the fact that A_1 is different.

From Lemma 45, the number of accesses required at the level-i is
$A_i \leq \frac{\mathcal{Q}_i A_1 B_1}{B_i} + \frac{\mathcal{Q}_i}{B_i} \sum_{j=2}^{i} \frac{B_j}{\mathcal{Q}_{j-1}}$. Simplifying this with the assumption of inclusive large-line caches and a uniform hierarchy we got $A_i \leq A_1 B_1 + si$ (From Equation (8.2)). Thus,

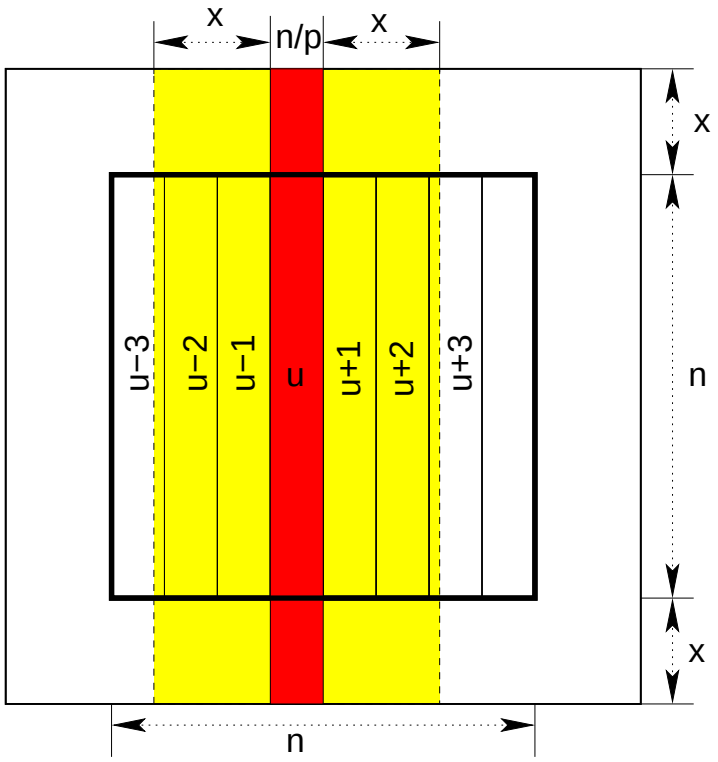


Figure 8.3: Accessing the neighborhood data

$A_i \leq \sigma \left\lceil \frac{n(n+2x)}{PB_1} \right\rceil B_1 + si \leq \sigma \left(\frac{n(n+2x)}{PB_1} + 1 \right) B_1 + si \leq \left(\sigma \frac{n(n+2x)}{P} + \sigma B_1 \right) + si.$

Substituting $\sigma = \left\lceil \frac{xP}{n} \right\rceil$ in above, we have, $A_i \leq \left(\frac{xP}{n} + 1 \right) \frac{n(n+2x)}{PB_1} + \left(\frac{xP}{n} + 1 \right) B_1 + si$

The number of accesses required at the level-i is given by

$$A_i \leq \frac{x(n+2x)}{B_1} + \frac{n(n+2x)}{PB_1} + \frac{xPB_1}{n} + B_1 + si \tag{8.4}$$

We now derive the time T_N to access local data. Recall that we have only considered the subtiles on one side of core u. Let an L_i-cache require α_i access time. Let $\alpha_m = \max\{\alpha_i : 1 \leq i \leq h\}$, then the time required for A_i accesses is $T_{Ni} = 2\alpha_i A_i$. The time to access all

the local data is

$$\begin{aligned}
T_N &= 2\sum_{i=1}^{h} T_{Ni} = 2\sum_{i=1}^{h} \alpha_i A_i \\
&\leq \left(\tfrac{x(n+2x)}{B_1} + \tfrac{n(n+2x)}{PB_1} + \tfrac{xPB_1}{n} + B_1\right)\sum_{i=1}^{h} 1 + 2s\sum_{i=1}^{h} i \\
&\leq 2\alpha_m\left(\tfrac{x(n+2x)}{B_1} + \tfrac{n(n+2x)}{PB_1} + \tfrac{xPB_1}{n} + B_1\right)h + s\alpha_m\tfrac{h(h+1)}{2} \\
&\leq 2\alpha_m\left(\tfrac{x(n+2x)}{B_1} + \tfrac{n(n+2x)}{PB_1} + \tfrac{xPB_1}{n} + B_1\right)\log P + \alpha_m s\left(\log^2 P + \log P\right) \\
T_N &\leq 2\alpha_m\left(\tfrac{nx}{B_1} + \tfrac{2x^2}{B_1} + \tfrac{n^2}{PB_1} + \tfrac{2xn}{P} + \tfrac{xPB_1}{n}\right)\log P + B_1\log P + \alpha_m s\left(\log^2 P + \log P\right)
\end{aligned}$$

Again since s is small and $P \leq n$, the terms outside the parenthesis may be ignored. $T_N = O\Big(\frac{x(n+x)}{B_1} + \frac{n}{P}\left(\frac{n}{B_1} + 2x\right)\Big)\log P$.

That is

$$T_N = O\left(\frac{x(n+x)}{B_1} + \frac{n}{P}\left(\frac{n}{B_1} + 2x\right)\right)\log P \tag{8.5}$$

Again the overhead is $O(\log P)$.

8.3.3 Running SIFT on the Subtile

The compute stage complexity is the time taken by the stage S_1 to extract features from the tile received from the previous stage and deliver features to the next output stage. This time for tile τ_k is denoted by T_C. Table 2.1 shows the time complexities and the number of operations required by each phase of the SIFT algorithm for N^2 pixels. Applying these to the $n \times \frac{n}{P}$ pixel tiles and then substituting the values of $\alpha = 0.6\%, \beta = 35\%$ and $\gamma = 0.04\%$ values (from Section 3.3) gives the time complexity (number of operations) of the computation stage as shown in Table 8.1. The fact that a subtile is not a square does not affect the time complexity.

Table 8.1: The number of operations required by phases of SIFT for $n \times \frac{n}{P}$ pixel tiles

Phase	Number of operations
Gaussian Blurring	$4\frac{n^2}{P}w^2 s$
Difference of Gaussian	$4\frac{n^2}{P}s$
Scale-space Extrema Detection	$104s\frac{n^2}{P}$
Keypoint Detection	$0.6s\frac{n^2}{P}$
Orientation Assignment	$48s\frac{n^2}{P}$
Keypoint Descriptor Generation	$12.03x^2\frac{n^2}{P}$

Therefore with Γ_1 as the clock rate for the compute stage,

$$\begin{aligned} T_C &= \left(4\frac{n^2}{P}w^2 s + 4\frac{n^2}{P}s + 104s\frac{n^2}{P} + 0.6s\frac{n^2}{P} + 48s\frac{n^2}{P} + 12.03x^2\frac{n^2}{P}\right)\cdot b \cdot \Gamma_1 \\ &= \left(4w^2 s + 157s + 12.03x^2\right)\frac{n^2}{P}\cdot b \cdot \Gamma_1 \qquad (8.6) \\ &= O\left(\frac{n^2x^2}{P}\right) \qquad (8.7) \end{aligned}$$

The total time taken by Stage S_1 of the pipeline to process tile τ_k is $t_{k,1} = T_L + T_N + T_C$. Plugging in the values of T_L, T_N, T_C from Equations (8.3, 8.5, 8.7), We have

$$\begin{aligned} t_{k,1} &= \Theta\left(\frac{n^2}{P}\log P\right) + O\left(\frac{n^2}{P} + \frac{x(n+x)}{B_1} + \frac{n}{P}\left(\frac{n}{B_1} + 2x\right)\right)\log P + \frac{n^2x^2}{P} \\ &= O\left(\left(\frac{n^2}{P} + nx + x^2\right)\log P + \frac{n^2x^2}{P}\right) \end{aligned}$$

The quantity x is fixed by the algorithm and P is fixed by the multicore chip. Select n such that $n = \max\{x, P\}$. Then

$$t_{k,1} = O\left(\left(\frac{n^2}{P} + nx\right)\log P + \frac{n^2x^2}{P}\right) \qquad (8.8)$$

Theorem 46. For all $1 < i \leq h$, and for an inclusive large-line and uniform hierarchy, the time required by the P-core computation stage is $t_{k,1} = O\left(\left(\frac{n^2}{P} + nx\right)\log P + \frac{n^2x^2}{P}\right)$. ∎

We now consider two cases.

Case 1 ($P \leq x$): Here $n = \max\{x, P\} = x$ and $O\Big(\frac{n^2}{P} + nx\Big) = O\Big(\frac{x^2}{P} + x^3\Big) = O(x^3) = O(nx)$.

Case 2 ($P > x$) : Here $n = \max\{x, P\} = P$ and $O\Big(\frac{n^2}{P} + nx\Big) = O(P + Px) = O(Px) = O(nx)$.

In either case $O\Big(\frac{n^2}{P} + nx\Big) \log P = O(nx)$. So with $n = \max\{x, P\}$,
$t_{k,1} = O\Big(nx \log P + \frac{n^2x^2}{P}\Big)$.
For all value of P such that $P = O\Big(\frac{nx}{\log nx}\Big)$, $t_{k,1} = O\Big(\frac{n^2x^2}{P}\Big)$ which is optimal considering that a uniprocessor solution to the problem takes $O(n^2x^2)$ time.

Let us examine $\frac{x}{\log nx}$. In the algorithm of Lowe [13], $x = 8$. for higher feature accuracy x may increase. Here for $x = 8$, $\frac{x}{\log nx} \leq 1$, implies that $8 \leq 3 + \log n$ or $n \geq 2^5$. That is $P \leq 32 \leq n$ cores can be supported without loss of efficiency. If we increase $x = 9$, then $P \leq \frac{2^9}{9} \leq 56 \geq n$, so very modest increase in x allows for a much longer increase in P. If $x = 16$, the next logical higher value of x, then $P \leq 2^{12} \leq n$. Thus the method proposed can scale to quite large values of P.

8.3.4 Total Time

As we argued above, $t_{k,1} = O\Big(\frac{n^2x^2}{P}\Big)$, whereas $t_{k,0}$ (when non zero) is $\Omega(n^2)$. Even though Stage S_0 has parts where nothing need to be brought in, it can continue to bring in further data as it is the bottleneck. Thus $S_0 \succeq S_1 \succeq S_2$ holds. So Case 2 of Section 7.1 holds. We now have the following result for the total time.

Theorem 47. For any number of pictures $X \geq 1$, the time required to run the SIFT algorithm on a P-core chip is $T = T_0 + T_1 + T_2$ where T_0 is the time taken by the first stage to input all X pictures and T_1, T_2 are the times to process the last tile of the last picture in the compute and output stages. ■

This theorem clearly illustrates the importance of large input bandwidth, without which the parallelism of multicore environment cannot be fully used.

8.4 Memory Requirement

We assumed the multicore chip to swap out any pixel data that was not required for the future use. Therefore at level L_h the memory requirement will be M_k the same as in the uniprocessor case. Since each subsequent level can operate by holding no more than what its parent cache holds (as we have an inclusive cache), and since memory usage is fully predictable, a level L_i-cache can also swap out anything it does not need. Thus the total memory at any level L_i is M_k. Since there are $\mathcal{R}_i$ cache modules in each level i, each L_i cache module has size $\frac{M_k}{\mathcal{R}_i}$.

From Theorem 40, the amount of memory required to run the SIFT algorithm on a 1-chip (3-stage) uniprocessor pipeline for row major ordering is $M_k = \Theta(Nx)$. So the amount of memory required in each module of L_i-cache on a 1-chip multicore pipeline is $\Theta\left(\frac{Nx}{\mathcal{R}_i}\right)$.

Theorem 48. The size of each module of L_i-cache required to run the SIFT algorithm on a P-core chip $O\left(\frac{Nx}{P}\mathcal{Q}_i\right)$. The total memory needed is $O(hNx)$. ∎

Observe the memory requirement is the same as in the uniprocessor case assuming h to be a constant.

Chapter 9

Two-Chip, Multicore Processor

In this chapter we study the performance of the SIFT algorithm on a two-chip processing pipeline (see Figure 9.1), where each chip can be single core or multicore. We model the multicore platform using the hierarchical multi-level-caching model (HM model) [4], the same as in the case of single-chip multicore pipeline.

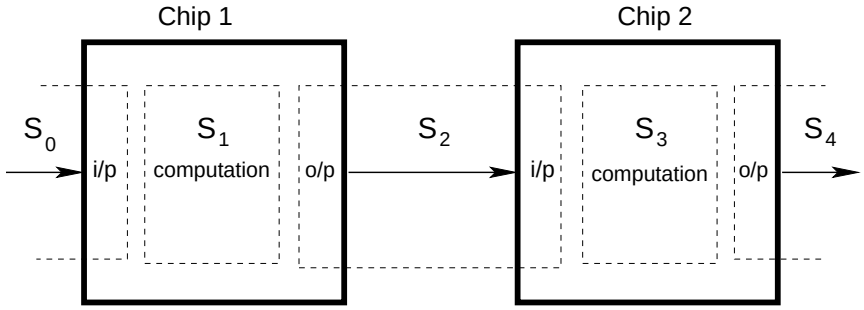


Figure 9.1: A two-chip pipeline

As discussed in Chapter 5, the two-chip model results in a 5-stage pipeline. These stages are denoted by S_0, S_1, S_2, S_3 and S_4 according to the notation discussed in Section 5.1. The tiles enter the pipeline at Stage S_0 and the final output is delivered through Stage S_4. The input and output stages are discussed in Section 7.2. The computation of SIFT on an image is divided among the two chips. The first chip computes the phases of Gaussian blurring, difference of Gaussians, scale-space extrema and keypoint detection in stage S_1. The Chip 2 computes the phases of orientation assignment and keypoint descriptor generation in Stage S_3. The parameters for the Stage S_2 is the same as that of Stage S_0. The relation between these five stages is assumed to be $S_0 \cong S_2$ and $S_0, S_1, S_2, S_3 \succeq S_4$.

Table 9.1: The number of SIFT operations required for an n^2 pixel tile on a Uniprocessor chip

Phase	Number of operations
Gaussian Blurring	$4n^2w^2s$
Difference of Gaussian	$4n^2s$
Scale-space Extrema Detection	$104sn^2$
Keypoint Detection	$0.6sn^2$
Orientation Assignment	$48sn^2$
Keypoint Descriptor Generation	$12.03x^2n^2$

Table 9.2: The number of SIFT operations required for an $n \times \frac{n}{P}$ pixel tile on a P-core chip

Phase	Number of operations
Gaussian Blurring	$4\frac{n^2}{P}w^2s$
Difference of Gaussian	$4\frac{n^2}{P}s$
Scale-space Extrema Detection	$104s\frac{n^2}{P}$
Keypoint Detection	$0.6s\frac{n^2}{P}$
Orientation Assignment	$48s\frac{n^2}{P}$
Keypoint Descriptor Generation	$12.03x^2\frac{n^2}{P}$

Tables 9.1 and 9.2 show the number of operations required by the different phases of SIFT to process an image of size $n \times n$ for the uniprocessor case and $n \times \frac{n}{P}$ for the multicore case. (See also Sections 2.5, 7.2, 8.3.3). Of the phases shown in the Tables, the Stage S_1 of Chip C_1 corresponds to Gaussian blurring, difference of Gaussians, Scale-Space extrema detection and keypoint detection, Stage S_3 of the Chip C_2 handless to the remaining phases.

9.1 Time Complexities of the Stages

The time complexities of the input and output stages for the 2-chip model pipeline are same as the complexities of the single-chip uniprocessor and single-chip multiprocessor pipelines discussed in Section 7.2.

The time taken by the compute stages S_1 and S_3 depends on whether they use uniprocessor or multicore chips. For $i \in \{1, 3\}$ and $\alpha \in \{U, M\}$ (for uniprocessor or multicore), Let $t^{\alpha}_{k,i}$ denote the time for Stage S_i to process tile τ_k. Observe that S_3 works on an $n \times n$ tile with an x neighborhood. In the same way S_1 works an an $n \times n$ tile with a w neighborhood. Thus if $t^{\alpha}_{k,3} = f(n, x)$ then $t^{\alpha}_{k,1} = f(n, w)$. We showed in Section 7.2, that $t^{U}_{k,3} = O(n^2x^2)$, so $t^{U}_{k,1} = O(n^2w^2)$. In Section 8.3, we showed that for an inclusive large-line and uniform hierarchy, with P_3 processor cores $t^{M}_{k,3} = O\Big(\frac{n^2x^2}{P}\Big)$, so with P_1 cores $t^{M}_{k,1} = O\Big(\frac{n^2w^2}{P}\Big)$.

Since P_3 and P_1 would be chosen to be as large as possible as argued in Section 8.3.4 for the multicore case, $t^{U/M}_{k,1} = O\Big(\frac{n^2w^2}{P}\Big)$ which is smaller than the input complexity $\Theta(n^2 + x^2)$. However the constants with the $t^{U/M}_{k,1}$ are quite small. If $x < n$, then most input iterations would be non-empty and $S_0 \succeq S_1, S_3$. So Stage S_0 would be the maximal stage. Therefore the time taken by the 2-chip pipeline to run the SIFT in the image is $T=$ Input time for all tiles + time to move the last tile through the rest of the pipeline.

Again this stresses the importance of an increased input bandwidth.

If Stage S_3 is a uniprocessor stage, then clearly its $O(n^2x^2)$ complexity would dominant. So the maximal stage in the pipeline would be Stage S_3. The time T taken by the pipeline to process an image is the time taken by Stage S_3 to process an entire image plus the time taken by the Stages S_0, S_1, S_2 for the first tile and time taken by the Stage S_4 for last tile.

Note that since Stage S_3 represents only part of the SIFT computation, the time is till an improvement over the single chip uniprocessor case.

If Chip 1 is a uniprocessor chip but Chip 2 is a multicore chip then $t^{U}_{k,1} = O(n^2w^2)$ and $t^{M}_{k,3} = O\Big(\frac{n^2x^2}{P_3}\Big)$. Assuming the times for the Stages S_1 and S_3 match, we will still be bottlenecked by the input stages S_0 and S_3.

In summary, for this split of computing among the two chips, it appears to be better to use less resources for Chip 1 than Chip 2, based on the times of these stages (see Table 9.2, 9.1). However, the bottleneck is still the input stage.

Chapter 10

Conclusion and Future Work

In this thesis, we developed a template for running SIFT in terms of tiles that facilitates its analysis without getting bogged down on input/output details. We developed a c-chip ($(2c+1)$-stage) pipeline model and derived general expressions for the time required to run SIFT on the model. We considered uniprocessor and multicore computing platforms and analyzed SIFT on a single-chip (3-stage) pipeline model and the two-chip (5-stage) pipeline model and for all combinations of single and multicore chips. Two tile orderings (row major and diagonal) were considered as well.

In the single-chip uniprocessor pipeline model (consisting of stages S_0, S_1, S_2) the time to run SIFT is essentially the time to perform the computation on the complete image on stage S_1. The times due to the input stage S_0 and output stage S_2 are less relevant, being restricted to that needed by the input stage S_0 to make data of the first tile available to the Stage S_1, and time needed by the output stage S_2 to output the features of last tile; these times are much smaller than the time to run SIFT on the entire image (that stage S_2 does). The overall time complexity for an $N \times N$ image with $n \times n$ tiles was shown to be $\Theta\left(\frac{(n+x)^2}{p_i}\Gamma_0 + \alpha\beta N^2 x^2 \Gamma_1 + \frac{(\alpha\beta+\gamma)\, n^2 \log x}{p_o}\Gamma_2\right)$; where Γ_ℓ is the clock rate of a stage S_ℓ and p_i, p_o are the number of input and output pins and α, β, γ are the feature fractions. Here x is a measure of the size of the neighborhood of a keypoint defined by SIFT. In this complexity, the middle term that depends N is the largest. The remaining terms depend on x and n that are much smaller ($\frac{N}{n}$ could be a around 100, so $\frac{N^2}{n^2}$ could be several thousands). We showed that the overall time complexity will not increase significantly as long as $\frac{(n+x)^2\Gamma_0}{p_i}, (\alpha\beta+\gamma)\, n^2\Gamma_2 = \Theta\left(N^2 x^2 \Gamma_1\right)$. This allows the possibility to reduce Γ_0, Γ_2 there by reducing the power and/or improving the data transmission quality for input/output.

As we move to a multicore computational platform (modeled using the HM model), the input stage of the pipeline becomes the bottleneck as the computation time at the Stage S_1 is decreased due to the use of multicores. We showed that, broadly speaking, about $P = O\left(\frac{nx}{\log nx}\right)$ cores can be employed fully, but increasing x slightly from 8 to 9 increase the range of cores to 56. With the parameter values used by Lowe [13] about 60 cores can be used. The time taken by the pipeline to run SIFT on an $N \times N$ image here equals the time taken by Stage S_0 to input the entire image plus the time taken by stages S_1, S_2 to perform computation and feature output for the last tile. The input is the bottleneck and without better input bandwidth, the power of multicore processing cannot be utilized fully.

In the two-chip (5-stage) model pipeline where each chip can be a single core or multicore, the exact relationship between the stages depends on the chip (multicore or uniprocessor) used. In any case, once again Stage S_0 is the bottleneck. In summary, we have established that without major improvements in input bandwidth, algorithms such as SIFT cannot fully utilize the power of multicore technology.

We also derived the expressions for the amount of memory needed to run SIFT. In general the amount of space is $\Theta(Nx)$ that is roughly $\sqrt{\text{image size}}$. There are many other directions for possible future research. In the derivation of the total time to run SIFT, the maximal stage is a key stage to identify. For the case where $S_m \succeq S_{m+1} \preceq S_{m+2} \succeq S_{m+3} \cdots \preceq S_{2c}$ we developed a time bound that we feel can be further tightened. However, we do not expect it to significantly alter the results and conclusions reached in this thesis. The 3-stage and 5-stage pipeline models are studied in this thesis. To analyze the $(2c + 1)$-stage pipeline, a detailed study of SIFT is needed to identify the computation at the stages themselves and to establish relationships between these stages. In the 5-stage pipeline, Stage S_3 needs

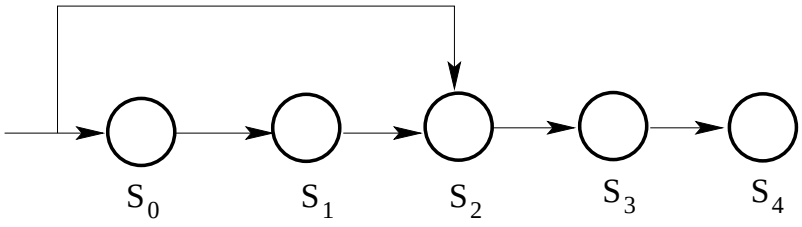


Figure 10.1: Splitting the data at Stage S_0 for Stage S_1 and Stage S_3

input data from Stage S_2 that is same as that of Stage S_0. But Stage S_1 does not require this entire data. Data transfer as shown in the Figure 10.1 could be employed.

The number of features of an image depends on the level of detail in the image and its contents. In applications such as surveillance, the camera points to a fixed point and all images are likely to have common features. This may also be the case for video data where

successive frames are correlated. One could consider mechanisms to exploit this, possibly on a model that configures itself to suit the common features.

Bibliography

[1] L. Arg, M. T. Goodrich, M. Nelson and N. Sitchinava, "Fundamental Parallel Algorithms for Private-Chip Multiprocessors," In *Proc. 20th ACM Symposium on Parallelism in Algorithms and Architectures,* pp. 107-206, 2008.

[2] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh1, D. McCauley, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb, "Die Stacking (3D) Microarchitecture," *Proc. 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06),* 2006.

[3] M. Brown, D. G. Lowe, "ecognising Panoramas," *Proc.* 9^{th} *IEEE International Conference on Computer Vision,* pp. 12181225.

[4] R. A. Chowdhury, F. Silvestri, B. Blakeley and V. Ramachandran, "Oblivious Algorithms for Multicores and Network Processors," Univeristy of Texas Computer Science Technical Report TR-09-19, July 2009.

[5] C. Seiculescu, S. Murali, L. Benini and G. De Micheli, "3D Network on Chip Topology Synthesis: Designing Custom Topologies for Chip Stacks," *3D Integration for NoC-based SoC Architectures, Integrated Circuits and Systems,* 2009.

[6] K. Chakraborthy, P. M. Wells and G. S. Sohi "Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly," *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS'06),* Oct. 2006.

[7] H. Nasir, V. Stankovic, and S. Marshall, "Image Registration For Super Resolution Using Scale Invariant Feature Transform, Belief Propagation and Random Sampling Consensus," *18th European Signal Processing Conference (EUSIPCO-2010),* Aug. 2010.

[8] S. Heymann, K. Mller, A. Smolic, B. Froehlich, and T. Wiegand, "SIFT Implementation and Optimization for General-Purpose GPU ," In *proc. of the WSCG'07,* Jan. 2000.

[9] Intel 64 and IA-32 Architectures Software Developers Manua, vol. 1: Basic Architecture.

[10] J.-Y. Kim, S. Oh, S. Lee, M. Kim, J. Oh, and H.-J. Yoo, "An Attention Controlled Multi-core Architecture for Energy Efficient Object Recognition," *Signal Process. Image Commun,* 2010.

[11] T. Ko, Z. M. Charbiwala, S. Ahmadian, M. Rahimi, M. B. Srivastava, S. Soatto, and D. Estrin, " Exploring Tradeoffs in Accuracy, Energy and Latency of Scale Invariant Feature Transform in Wireless Camera Networks," *IEEE*, 2007.

[12] M. C. Jordon, "A Configurable Decode for Pin-Limited Applications," MS Thesis, Electrical and Computer Engineering, Louisiana State University, Dec. 2006.

[13] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision,* vol. 60, no. 2, pp. 91–110, Nov. 2004.

[14] D. G. Lowe, "Object Recognition from local scale-invariant keypoints," In *proc. of International Conference on Computer Vision,* pp. 1150-1157, 1999.

[15] D. G. Lowe, "Local feature vire clustering for 3D object recognition," *IEEE Cconference on Computer Vision and Pattern Recognition,* pp. 682-688, 2001.

[16] D. G. Lowe, J. Little,"Vision-based mobile robot localization and mapping using scale-invariant features," *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA). 2,* pp. 2051, doi:10.1109/ROBOT.2001.932909. http://citeseer.ist.psu.edu/425735.html.

[17] K. Mikolajczyk, "Detection of local features invariant to affine tranformations," *Ph.D. thesis,Institut National Polytechnique de Grenoble,* France.

[18] "OpenCV 2.1 C++ Reference," http://opencv.willowgarage.com/documentation/cpp/index.htm

[19] Hess - SIFT Library, http://blogs.oregonstate.edu/hess/code/sift/.

[20] S. Cho, P.-C. Yew, and G. Lee, "A High-Bandwidth Memory Pipeline for Wide Issue Processors," *IEEECS*, 2001.

[21] B. H. Shekar, M. Sharmila Kumari, L. M. Mesteskiy and N. Dyshkant, "FLD-SIFT: Class Bases Scale Invariant Feature Transform for Accurate Classification of Faces," *CNC 2011,CCIS 142*, pp. 15–21, 2011.

[22] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc, "Feature Tracking and Matching in Video Using Programmable Graphics Hardware, *Machine Vision and Applications,* Mar. 2007

[23] T. Wen, X. Fan, W. H. Yuan, and Z. Bo, "Fast Scale Invariant Feature Transform Algorithm Based on CUDA," DOI CNKI:SUN:JSJC.0.2010-08-079.

[24] Z. Qu and Z. Wang, " The Improved Algorithm of Scale Invariant Feature Transform on Palmprint Recongnition," *Advanced Material Research,* vol. 186, pp. 565–569, Jan. 2011.

[25] A.P. Witkin, "Scale-Space Filtering," *International Joint Conference on Artificial Intelligence,* pp. 1019-1022, 1983.

[26] I. A. Young, E. Mohammed, J. T. S. Liao, A. M. Kern, S. Palermo, B. A. Block, M. R. Reshotko, and P. L.D. Chang, "Optical I/O Technology for Tera-Scale Computing," *Digital Object Identifier.*

[27] Y.-s. Lin, S.-M. Chang, J. C.Tsai, T. K.Shih and H.-H. Hsu, "Motion Analysis Via Feature Point Tracking Technology," *MMM'11 Proceedings of the 17th International Conference on Advances in Multimedia Modeling,* vol. Part II.

[28] Q. Zhang, Y. Chen, Y. Zhang and Y. Xu, "SIFT Implementation and Optimization for Multi-Core Systems," *Proc. of the IEEE,* 2008.

[29] Piper Seneca RP-C 2123.jpg
http://www.griffinaviation.in/Images/Piper%20Seneca%20I%20RP-C%202123.JPG
(last retrieved date, 04-24-2011).

[30] Exterior-Challenger-850.jpg
http://www.stratosjets.com/charter-aircraft/Bombardier/Challenger/Challenger-850/Exterior-Challenger-850.jpg
(last retrieved date, 04-24-2011).

[31] Old warden.jpg
http://upload.wikimedia.org/wikipedia/commons/c/c1/504_at_Old_Warden.jpg
(last retrieved date, 08-24-2010).

[32] AirOne Flight Academy Gallery
http://www.aironeflightacademy.com/files/gallery_images/IMG_0101.JPG
(last retrieved date, 08-11-2010).

[33] Orlando FL, EPCOT, sphere.jpg
http://www.pleaseyourself.ca/Orlano%20FL,%20EPCOT,%20sphere.jpg
(last retrieved date, 04-24-2011).

[34] Opera Sphere.jpg
http://www.cameronballoons.co.uk/images/gallery/7/Opera_Sphere.jpg
(last retrieved date, 08-11-2010).

[35] chrome-sphere.jpg
http://www.psdgraphics.com/file/chrome-sphere.jpg
(last retrieved date, 04-24-2011).

[36] orange-sphere.jpg
http://www.psdgraphics.com/file/orange-sphere.jpg
(last retrieved date, 04-24-2011).

[37] einstein2.jpg
http://elementaryteacher.files.wordpress.com/2010/01/einstein2.jpg
(last retrieved date, 08-11-2010)

[38] File:Albert_Einstein_1947a.jpg
http://commons.wikimedia.org/wiki/File:Albert_Einstein_1947a.jpg
(last retrieved date, 04-24-2011).

[39] Einstein and Thatcham Mindwork's Webblog
http://mindworksblog.com/2008/07/31/einstein-and-thatcham/
(last retrieved date, 04-24-2011).

[40] Einstein Potrait2.jpg
http://www.neoformix.com/2008/EinsteinWordPortrait.html
(last retrieved date, 08-11-2010)

[41] Mohandas Karamchand GANDHI cille85
http://cille85.wordpress.com/2009/04/16/mohandas-karamchand-gandhi/
(last retrieved date, 04-24-2011).

[42] index of ngo images 3362122.jpg
http://aryangroupofhospital.com/ngo/images/3362122.jpg
(last retrieved date, 04-24-2011).

[43] Mohandas Karamchand GANDHI cille85
http://cille85.wordpress.com/2009/04/16/mohandas-karamchand-gandhi/
(last retrieved date, 04-24-2011).

[44] Little known Facts About the Great Mahatma
http://www.randomthoughtz.com/archives/1007
(last retrieved date, 04-24-2011).

[45] Palm-trees1.jpg
http://haveanopinion.files.wordpress.com/2008/04/palm-trees1.jpg
(last retrieved date, 04-24-2011).

[46] armstrongpalms.jpg
http://0.tqn.com/d/goneworleans/1/0/X/A/armstrongpalms.JPG
(last retrieved date, 04-24-2011).

[47] palm-trees.jpg
http://withoutwords.files.wordpress.com/2007/11/palm-trees.jpg
(last retrieved date, 04-24-2011).

[48] frindswsunbehindreallife2.jpg
http://www.tikibarcentral.com/frondswsunbehindreallife2.jpg
(last retrieved date, 04-24-2011).

[49] 81552687.jpg
http://content.answcdn.com/main/content/img/ getty/8/7/81552687.jpg
(last retrieved date, 04-24-2011).

[50] http://1.bp.blogspot.com/
_7cOXlIkyaH0/TS9_yEMrLI/AAAAAAAAAIk/h7GkUHJsB7A/s1600/
OconnorPhoto2009.jpg
(last retrieved date, 08-11-2010).

[51] 74166268.jpg
http://content.answcdn.com/main/content/img/getty/6/8/74166268.jpg
(last retrieved date, 04-24-2011).

[52] Buffmotorsports Images http://buffmotorsports.com/images/
img_0290_dm7g.jpg
(last retrieved date, 08-11-2010)

[53] Index of /classifieds/uploadt/au/vehicles http://www.adpost.com/classifieds/
uploadt/au/vehicles/au_vehicles.7104.1.jpg
(last retrieved date, 08-11-2010).

[54] www.fleetowner.com http://blog.fleetowner.com/trucks_at_work/wpcontent/uploads/
2010/08/fo2011chevysilveradonm.jpg
(last retrieved date, 08-11-2010).

Vita

Phaneendra Vinukonda was born in April, 1987, in Warangal, Andhra Pradesh, India. He graduated with his Bachelor of Technology in Electrical and electronics engineering from Jawaharlal Nehru Technological University, Hyderabad, India, in the year 2007. He is presently enrolled in master's program in electrical and computer engineering at Louisiana State University and is expected to graduate in May 2011. His research interests include networking, interconnection networks and computer architecture.