An implementation of k-tiling algebra

Matthew Swan, Ken Hawick,
Paul Coddington
(maswan|khawick|paulc)@cs.adelaide.edu.au

Technical Report DHPC–098
27 October 2000

Department of Computer Science
Adelaide University,
South Australia, 5005

Abstract

A large amount of regular multi-dimensional data exists in the field of Computer Science. This multi-dimensional data comes from a variety of sources. Satellite data, data gathered from scanners in medical equipment and the Visible Human data set[Waldby, 2000] are just a few examples of multi-dimensional data sets. What is required is a general method to perform mappings of this multi-dimensional data onto other multi-dimensional data sets.

In 1993, the k-tile format for performing a number of different mappings from one multi-dimensional array onto another was described [Fletcher, 1993]. A number of extensions to the k-tile format were also described, adding to the types of mappings that could be performed. A complete implementation of the k-tile format however has not been developed, so much of this work remains untested.

A revision of both the k-tile format and the k-tile extensions from those previously described in [Fletcher, 1993] is described here. In forming the

revision, a complete implementation of the k-tile format and the k-tile extensions was developed for the first time, and the algorithms developed are also described here.

In order to perform mappings however, a language was required to specify the k-tile mapping; this led to the development of the k-script language. The k-script language uses XML to define a k-tile, and the data on which to perform the mapping. A simple compiler called kScript was built to parse the k-script language into code that could be executed by a k-tile implementation.

In addition to k-script, a number of other new additions have been made to the k-tile format. These include the k-blocking extension, the subsection extension and generic k-tiles.

A number of example mappings are presented here, demonstrating the working implementation developed.

Contents

List of Figures

List of Tables

Acknowlegement

This thesis would not have been possible without the help of a number of people. First I would like to thank my supervisors Ken Hawick and Paul Coddington. I would like to thank Ken especially, for allowing me the freedom to build my own project and giving many suggestions as to how I should proceed, supporting me all the way. I am very grateful for his support during honours and in the past allowing me to reach this point in my life.

Paul too deserves a big thank you for giving me many suggestion on improvements and new ideas to expand on what I had already done. Working through revision after revision of this thesis for me has provided valuable insight in areas that needed improvement and corrections that I would not have found.

Thanks also goes to Francis Vaughan who has been like a backup supervisor to me in the past 6 months. He provided a lot of much needed technical feed back on various aspects of my project and his assistance with my talk was incredible. His suggestions resulted in the unique and very well received end of year presentation that I gave for this project.

Also Heath James deserves thanks for providing me with the start I needed in XML and his many corrections in the early days of my thesis and the *k*-script language.

I would like to also like to thank the many friends I made in honours this year. Diana Howard who gave me feed back on my project, despite being snowed under with work herself, Katherine Mickan who kept me laughing through the worst of it and everyone else who I worked with in honours. It has been a privilege to work with all of you this year and I hope to keep seeing you all in later years.

Thanks also to my parents who supported me throughout the year while I finished my honours degree. Without them, this thesis would not have been possible.

Finally I would like give a massive thank you Angelina Holding for her understanding and love during my honours year, as well as the last minute corrections that she helped me make to my thesis.

Chapter 1

Introduction

In the field of Computer Science, multi-dimensional data represents a large amount of the data produced and collected. Typically it is necessary to convert this data into other, equivalent multi-dimensional data forms. The purpose behind the conversion of this data depends largely on how the data is to be processed by a computer or viewed by a user. A particular algorithm may require the data to be presented in a specific way, or in the interests of performance, the data may need to be split between a number of processors in a parallel machine. For example, very large images may need to be tiled into smaller images that will fit into the memory of a single processor in a parallel machine. The breaking up of this data into tiles may be performed either sequentially by a single processor in the parallel machine or by multiple processors.

A mapping may also be performed across a number of disks. These mapping are already done by disk systems that employ RAID [Wong] technology.

If the data is to be viewed by a computer user, the multi-dimensional data set may need to be manipulated into something that the user can understand. For instance, the Visible Human [Waldby, 2000] data set consists of a number of slices through the human form. A user may wish to view a small section or slice of this data set, requiring the computer to convert the multi-dimensional form of the data set into a new, reduced data set that the user requires.

1.1 Existing mapping technologies

A number of technologies already exist to perform different data mappings. One popular system which is used for the storage of large amounts of data, with high speed access is RAID[Wong]. Systems utilise a number of disks operating in parallel. RAID maps single dimensional arrays of bytes rep-

resenting files over an array of disks. RAID systems allow 6 different configurations, RAID-0 to RAID-5. Each one of these configurations specifies how data is split up over the disks. To summarise, the primary methods are striping across all disks, mirroring the data, or a combination of both of these mappings. Parity data can also be added so that the data on a disk is recoverable in a situation where it fails. In general, the mappings performed by a RAID system consider the storage of large amounts of data.

Another system which has been proposed for the mapping of data is Vesta [Dror G Feitelson and Prost]. This system was designed to provide file access to data from parallel processors [Baylor and Wu, 1999]. The control of data layout is provided so that anticipated access patterns of data can be taken advantage of. Single files can be broken up into a number of smaller files, then spread over the disks, improving random access to the files.

Both RAID and Vesta however are limited in the mappings they can perform. RAID only allows for 6 different configurations and Vesta has a similar limitation on the mappings it can perform. For instance, RAID and Vesta cannot apply operations like rotations, padding and replication to a data set. Neither RAID or Vesta can be applied to any other applications than distributing files over a number of disks.

What is required is a general technique to map from a multi-dimensional data set to another. This technique should ideally have a mathematical basis and be as general as possible in covering the many different multi-dimensional data sets and their subsequent mapping.

Such a general technique already exists and was described in 1993 Peter Fletcher [Fletcher, 1993]. Fletcher describes a mathematical technique to describe multi-dimensional mappings called the k-tile format. The k-tile format is based on Fraser's [Fraser, 1988] multi-dimensional tile format built during his work with the CSIRO. The multi-dimensional format hierarchically broke down the image dimensions into a higher-dimensional space.

Fletcher designed an implementation of the k-tile format for the MasPar MP-1 SIMD computer called PISTON which allowed the viewing of mappings. This implementation however, would only work when the size of the dimensions making up the multi-dimensional array were powers of 2.

1.2 Goal

The goal here is to describe a complete implementation of the k-tile format and the extensions proposed. This implementation is to be as general as possible, programmed in Java[Patrick, 1999] and not limit the size of the dimensions in the multi-dimensional array to powers of 2. The use of Java

ensures compatibility of the code with a large range of architectures, as platform independent code is a goal of the Java language.

Building an implementation of the basics for the k-tile format involves going through the format, making corrections where necessary and removing any limitations and potential implementation difficulties. Once the implementation of the basics is achieved, the extensions proposed to the k-tile format by Fletcher [Fletcher, 1993] need to be explored with corrections made where necessary. The extensions to the k-tile format can then be implemented for the first time.

An implementation alone however is not sufficient, and there needs to be a general way that k-tiles can be specified. This is to be performed in the form of the creation of a language called k-script which specifies a k-tile mapping and generic k-tiles which specify the mapping aspect of a k-tile and not the size of the data.

1.3 Structure

Chapter 2 describes the syntax used for multi-dimensional arrays. A small sample of a few multi-dimensional arrays is also given to further described the syntax.

Chapter 3 describes the k-tile format, using the definitions given in chapter 2. This chapter includes much of Fletcher's work on the k-tile format. Modifications made to the k-tile format will be noted, with a description of the purpose of the modification.

Chapter 4 describes the extensions to the k-tile format. These extensions add further functionally to a k-tile mapping. Operations such as rotations, padding, offsets and replication are added to the k-tile format. In addition to this, a new extension called subsections is described here.

Chapter 5 examines some of the problems of the k-tile format discovered while working through an implementation of the format. It was found that not all mapping are possible. k-blocking is an extension to the k-tile format which can be used to solve problems in performing mappings between two multi-dimensional arrays. This extension is also used to solve other implementation problems which occurred while developing the implementation of the k-tile format.

An introduction to the generic k-tile is given in chapter 6. This represents a new addition to the k-tile format, separating the size of the data from the mapping performed. In a normal k-tile, two aspects of the mapping are described. Firstly, the size of both the target and the source multi-dimensional data sets and secondly the mapping performed, such as a tiling

or a rotation. The aim of a generic k-tile is to capture the mapping aspect of a k-tile and keep the size of the target and source data sets non-specific.

The implementation developed is described in chapter 7, with the algorithms developed given in pseudo code. This implementation tests the k-tile format and examples of mappings are provided to verify the format.

Chapter 8 describes the k-script language, which provides a way to specify a k-tile mapping between two multi-dimensional spaces. In this way, k-tiles can be easily defined to perform mappings on data sets. The k-script language uses XML[Bosak and Bray, 1999], which makes the k-script language simple to follow, and easily expandable for future improvements.

Chapter 2

Multi-dimensional arrays

The only types of data sets that will be discussed here will be regular or rectangular data and not irregular data sets. Irregular data sets represent non-rectangular data and will not be addressed, since the algorithms and techniques to perform mappings on these data sets are different from those used on a regular data sets. The use of irregular data sets would complicate the k-tile mappings described here.

A regular data set can be described as a rectangularly grided multi-dimensional array with a fixed number of elements along each dimension.

Mapping a multi-dimensional array onto another multi-dimensional array can be either dependent on the data in the array or independent of the data in the array. Both techniques are useful in Computer Science, but only mappings which are independent of data contained in the array will be focused on here. Such data mappings are called *data-value independent mappings*.

2.1 Definitions

In order to describe a data mapping technique, a notation needs to be derived for multi-dimensional arrays. This notation is given by Fletcher [Fletcher, 1993].

2.2 Multi-dimensional arrays

A multi-dimensional array is called a *space* and is described using the following terms:

- The *data element type* specifies the data type of elements in a space

- The *dimensionality* is the number of dimensions in a space
- The *shape* defines the length of each dimension in a space
- An *address* describes the location of a single element in a space
- The *index space* is the set of all addresses in a space
- The *size* of a shape is the size of the index space set

The definition of a space will be given in terms of Z in the following definitions, the space A will serve as an example of a space. Thus, A represents a multi-dimensional array which is defined as:

$$A = \begin{pmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{pmatrix}$$

2.2.1 The data element type for a space

The *Data element type* defines the type of data stored in a space. The example space A contains the numbers $0 \ldots 5$. Typical data element types for spaces are numbers and letters. The data element type for all spaces discussed here will be a single byte. Bytes can represent a number from 0 to 255 or ASCII characters.

2.2.2 The dimensionality of a space

The *dimensionality* of a space is defined as the number of dimensions in the multi-dimensional array the space represents. The notation (Z) defines the number of dimensions in a space Z. Hence the dimensionality of the example space A is: $(A) = 2$.

2.2.3 The shape of a space

The *shape* of a space is the size of each dimension in the multi-dimensional array represented. The shape of the space Z is defined as:

$$Z[z_0, \ldots, z_{(Z)-1}]$$

Hence the shape of the example space A is:

$$A[2, 3]$$

2.2.4 An address in a space

An *address* of a single element in a space is defined as a unique vector pointing to a single element in the multi-dimensional array. The definition for an address of an element in the space Z:

$$Z(u_0, \ldots, u_{(Z)-1}) \text{ where } \forall u_i, 0 \leq u_i \leq z_i - 1 \mid 0 \leq i \leq (Z) - 1$$

The following lists all of the addresses in the example space A and the corresponding elements that the address points to:

$$\begin{aligned} A(0,0) &= 0 \\ A(1,0) &= 1 \\ A(0,1) &= 2 \\ A(1,1) &= 3 \\ A(0,2) &= 4 \\ A(1,2) &= 5 \end{aligned}$$

2.2.5 The index space of a space

The index space of a space is defined as the set of all addresses in the space Z. This set is given by the function *index* which is applied to the space Z.

$$index(Z) = \{(u_0, \ldots, u_{(Z)-1}) \mid u_i, 0 \leq i \leq (Z) - 1 \mid\leq u_i \leq a_{i-1}\}$$

The index space for A is:

$$index(A) = \{(0,0), (1,0), (0,1), (1,1), (0,2), (1,2)\}$$

2.2.6 The size of a space

The *size* of a space is defined as the number of elements in the space. The function $size(Z)$ returns the size of the set $index(Z)$ where:

$$size(Z) = \prod_{i=0}^{(Z)-1} z_i$$

The size of the space A is:

$$size(A) = 2 \times 3 = 6$$

2.3 Summary for the definition of a space

Listed here is a summary of all of the definitions of a space in terms of Z.

- A multi-dimensional array is represented by the space Z
- The dimensionality of Z is $(Z) = q$
- The shape of Z is $Z[z_0, \ldots, z_{q-1}]$
- An address of Z is $Z(u_0, \ldots, u_{q-1})$ where $\forall u_i, 0 \leq u_i \leq z_{i-1}$
- The index space of Z is $index(Z)$
- The size of Z is $size(Z)$

2.4 Examples of spaces

Spaces representing multi-dimensional arrays come from a number of sources in Computer Science. Multi-dimensional arrays often represent the storage and presentation of data. The following is a list of shapes of common spaces representing multi-dimensional arrays consisting of bytes.

- An 8 bit greyscale 640×480 image, $Z[640, 480]$.
- A 24 bit colour (3 bytes) 3D data set with dimensions $10000 \times 10000 \times 10000$, $Z[3, 10000, 10000, 10000]$. The first dimension in Z represents the 3 colours; red, green and blue.
- A satellite image, (2D image$\times$spectral band), $Z[5, 1000000, 1000000]$.. The first dimension represents the 5 spectral bands in the image, infra-red, red, green, blue and ultra violet.
- A 1MB file on disk, $Z[1048576]$

2.5 Summary

This chapter has described a definition for multi-dimensional arrays which have been referred to as spaces. A space consists of a data element type, a shape and an index space. The data element type for all spaces discussed here will be a single byte, with more than one byte represented as an extra dimension in the space. The shape of a space represents the size of each

dimension in the space. The index space of a space is the set of all addresses to elements in the space.

This definition will be used in subsequent chapters to define mappings from one space to another.

Chapter 3

The k-tile format

In order to perform a mapping from a multi-dimensional array onto another multi-dimensional array a method of mapping the index space of the source array onto the index space of the target array is required. The k-tile format is an algebra which presents a general framework for describing the mapping of the index space of one multi-dimensional array into the index space of another multi-dimensional array. In addition to this, reordering, tiling, rotation, replication and padding can also be performed if desired. Since arrays can represent a large range of different types of data and devices, the potential uses for the k-tile format are wide ranging.

By using the definition for a multi-dimensional array given in chapter 2 which defines an array as a space with a shape, the basics of the k-tile format can be described. Extensions to the k-tile format which allow operations such as rotations, replication and padding of data will be described in chapter 4.

3.1 The k-tile spaces

In its simplest form, the k-tile format defines a mapping between 3 multi-dimensional arrays. Each one of these arrays is known as a space and will be denoted the *data space*, the *k-tile space* and the *device space*. The data space is the source data to be mapped from. The device space is the target of the source data after the mapping. The data space and the device space can potentially have different shapes. Finally, the k-tile space is an abstract multi-dimensional array which is mapped onto from the data space and is mapped to the device space.

3.1.1 The data space

The data space is a multi-dimensional array with a shape and index space. In the k-tile format, the data space is assigned the letter A and represents the source data for a k-tile mapping. Any data that needs to be mapped onto a device space and can be represented as a multi-dimensional array can be a data space. Files on a disk for example can be represented as single-dimensional arrays. An example of a few typical data spaces are:

- A 1MB file: A[1048576]
- A greyscale 324x324 image: A[324,324]
- A tiled image of 3x3 tiles each 108 x 108 in size: A[108,108,3,3]
- A 24bit colour 640x480 image: A[3,640,480]

3.1.2 The device space

The device space is a multi-dimensional array with a shape and index space. In the k-tile format, the device space is assigned the letter D and represents the target space for a k-tile mapping. This space can be represented by one or more files on disk or any number of processors in a parallel machine. Like the data space, anything that can be represented as a multi-dimensional array can be a device space. An example of a few typical device spaces are:

- A 1MB file: D[1048576]
- A group of 9 files, each 1MB in size: D[1048576,9]
- 32 arrays of 64 bytes distributed over a single processor: D[64,32]

3.1.3 The k-tile space

The k-tile space defines the data-to-device space mapping and is assigned the letter K. The data space is mapped onto the k-tile space which is in turn mapped onto the device space. Like the data and device spaces, the k-tile space is a multi-dimensional array.

A mapping vector m is associated with the k-tile space which represents the mapping of the dimensions in the k-tile space onto the dimensions of the device space. This mapping vector specifies a permutation of the k-tile space changing the order in which dimensions of K are mapped onto D. By changing the mapping vector and the shape of the k-tile space, tiling and row or column order mapping can be performed to the data.

3.2 A summary of the k-tile format

The k-tile format describes a data space A, a k-tile space K, a mapping vector m and a device space D where:

- A is the shape of the data space
- K is the shape of the k-tile space
- m is the mapping between the k-tile space and the device space
- D is the shape of the device space

A k-tile mapping uses these components to map the index space of the data space A onto the device space D. Both the data space and the device space represent multi-dimensional arrays, where the data space A is the source array and the device space D is the target array. A summary of the elements that make up the k-tile format are:

$$\begin{aligned} A[a_0, \ldots, a_{p-1}] &\text{ where } p = (A) \\ K[k_0, \ldots, k_{q-1}] &\text{ where } q = (K) \\ m(m_0, \ldots, m_{q-1}) &\text{ where } q = (K) \\ D[d_0, \ldots, d_{r-1}] &\text{ where } r = (D) \end{aligned}$$

When a k-tile mapping is performed, the data stored in A can be copied to D, using the mapping of the index space of A onto the index space of D. Each of the arrows in the following diagram shows the mapping of the index space from A onto D:

$$\boxed{A} \rightarrow \boxed{K} \rightarrow \boxed{D}$$

Note that this is a one-to-one mapping, which does not require A, K and D to have the same shape.

Before describing the mathematical basis behind the mapping process, and more specifically, the k-tile mapping process, this section will examine a simple mapping of the data space A onto the device space D.

3.2.1 Mapping the data space onto the k-tile space

A k-tile mapping is made up of 2 separate mappings, a mapping from the data space A onto the k-tile space K and a mapping from the k-tile space K onto the device space D. The data space represents the source data, and for

this example will be a single dimensional array with 4 elements. The formal definition of the k-tile to be demonstrated here is:

$A[4]$
$K[2,2]$
$m(1,0)$
$D[4]$

This definition describes the shape of the spaces and the mapping vector m. Thus the shape of the data space A is $[4]$, the shape of the k-tile space K is $[2,2]$ and the shape of the device space D is $[4]$. The index space of the data space is all of the addresses of elements inside the array A. In this case the index space for A is $index(A) = ((0),(1),(2),(3))$. Each of these addresses refers to a single byte in the array.

The mapping of the index space of A onto the index space of D first requires the mapping of the index space of A onto the index space of K. This mapping is performed using an *implicit mapping*. Implicit mappings are covered later in this chapter, however, for now assume them to be a mapping not requiring the m vector. The k-tile space K has the shape $[2,2]$ and the mapping of all of the addresses in the data space onto the k-tile space is shown in table 3.1.

Data index space	k-tile index space
(0)	(0,0)
(1)	(1,0)
(2)	(0,1)
(3)	(1,1)

Table 3.1: Mapping of addresses from $A[4]$ to $K[2,2]$. Shown is the entire index space of A mapped onto the index space of K. Each row represents a corresponding address in both spaces.

3.2.2 Mapping the k-tile space onto the device space

With the index space of A mapped onto the index space of K, the mapping of the index space K onto D can now be performed, completing the mapping. This mapping is called a *non-implicit* mapping and requires the m vector.

The vector m defines a permutation of the dimensions in the k-tile space during the mapping. The mapping vector m to be used for this mapping is $(1,0)$. When performing the mapping from K to D, the mapping vector is

read for each dimension in K mapped onto D. The first number in m is 1, this indicates that the second dimension of K will be mapped D first. The next number in m is 0 meaning the first dimension in K will now be mapped onto D. The result of this mapping can be seen in table 3.2.

k-tile index space	Device index space
(0,0)	(0)
(1,0)	(2)
(0,1)	(1)
(1,1)	(3)

Table 3.2: Mapping of addresses from $K[2,2]$ to $D[4]$. Each row represents a corresponding address in both spaces.

This completes the k-tile mapping as the index space of A has been mapped onto the index space of K which in turn has been mapped onto the index space of D.

3.2.3 Result of mapping

By studying tables 3.1 and 3.2, it can be easily seen that address (1) in the data space has been mapped to address (2) in the data space. Conversely address (2) in the data space has been mapped to address (1) in the device space. Table 3.3 shows the final mapping from the data space to the device.

Data index space	Device index space
(0)	(0)
(1)	(2)
(2)	(1)
(3)	(3)

Table 3.3: Mapping of addresses from $A[4]$ to $D[4]$. Each row represents a corresponding address in both spaces.

Now that the mapping of every address in the data space onto the device space is known, the data stored in the array A can be copied to the array D. The contents of both arrays can been seen in figure 3.1.

The cross over seen in figure 3.1 shows the effect of the m vector on the non-implicit mapping. The mapping that has resulted is actually a row-order mapping. If the m vector was the identity $(0,1)$ then there would be no difference between the arrays A and D. The effect of the mapping however

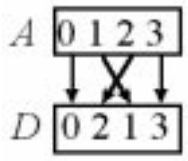


Figure 3.1: The contents of the array A and the array D after the k-tile mapping has been performed.

is more easily seen in an image. Figure 3.2 shows a 324x324 pixel image with the k-tile described above applied.

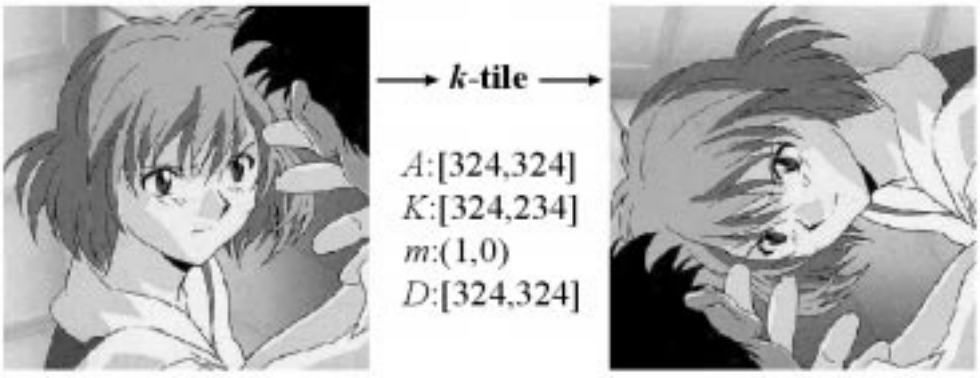


Figure 3.2: The original image on the left has a k-tile applied to it, resulting in a row order mapping of the image shown on the right.

The row order mapping causes the flipping around of the top left and bottom right hand corners in figure 3.2. Changing the m vector to $(0, 1)$ would result in the image being duplicated with no change. The actual specification for the k-tile can also be seen in figure 3.2. The shape of A,K and D has been changed to reflect the larger data but this does not effect the k-tile mapping being performed.

3.3 Workings of the k-tile format

In order to describe the k-tile format, it will first be necessary to examine the mapping between two spaces.

To perform a mapping between two spaces requires mapping the index space of the first space onto the second. It is possible to generate a formula to perform this mapping trivially. Take for example a data space A with the shape [3,4] and a k-tile space K with the shape [12]. If the index space of A

is represented as $A(w_0, w_1)$ then equation 3.1 represents a mapping from A to K.

$$A(w_0, w_1) = K(w_0 + a_0 * w_1) = K(w_0 + 3 * w_1) \tag{3.1}$$

Equation 3.1 maps the index space of A onto the index space of K. Such a mapping however could also be represented by equation 3.2.

$$A(w_0, w_1) = K(w_1 + a_1 * w_0) = K(w_1 + 4 * w_0) \tag{3.2}$$

The number of equations that actually exist is the factorial of the dimensionality[1] of the space that is being mapped from. If for example a mapping was being performed from a k-tile space K that consisted of 3 dimensions onto a device space D consisting of 1 dimension, then 6 different equations could be derived to perform the mapping. This is shown by the equations in 3.3.

$$\begin{aligned}
K(w_0, w_1, w_2) &= D(w_0 + k_0 * (w_1 + k_1 * w_2)) \\
K(w_0, w_1, w_2) &= D(w_1 + k_1 * (w_0 + k_0 * w_2)) \\
K(w_0, w_1, w_2) &= D(w_2 + k_2 * (w_0 + k_0 * w_1)) \\
K(w_0, w_1, w_2) &= D(w_0 + k_0 * (w_2 + k_2 * w_1)) \\
K(w_0, w_1, w_2) &= D(w_1 + k_1 * (w_2 + k_2 * w_0)) \\
K(w_0, w_1, w_2) &= D(w_2 + k_2 * (w_1 + k_1 * w_0))
\end{aligned} \tag{3.3}$$

In order to perform a mapping, there must exist some way to specify which of the equations in 3.3 is the mapping required.

3.3.1 The m vector

In order to specify which mapping of the equations in 3.3 is the correct one a mapping vector m is required. This vector specifies a permutation of the vector defining the shape of K in equations 3.3. The following list of m vectors in 3.4 would produce each of the equations in the order they appear in 3.3.

$$\begin{aligned}
m &= (0, 1, 2) \\
m &= (1, 0, 2) \\
m &= (2, 0, 1) \\
m &= (0, 2, 1) \\
m &= (1, 2, 0) \\
m &= (2, 1, 0)
\end{aligned} \tag{3.4}$$

[1] Dimensionality is the number of dimensions in the array.

It can be seen by comparing 3.3 and 3.4 how m selects which dimension in K to use. The definition for the m vector is:

$$m = (m_0, \ldots, m_{(K)-1})$$

Where m is a permutation of the set of integers $\{0, \ldots, (K) - 1\}$. The following properties are associated with the m vector:

- No dimension in the k-tile space appears more than once in m; m is in fact a permutation of the vector $(0, \ldots, q - 1)$ where $q = (K)$. This ensures that every k-tile dimension is only mapped once to a dimensions in the device space D.
- The product of the k-tile space dimensions specified by m being mapped onto a single dimension in D must be equal to the size of that dimension.

The first condition establishes that the m vector is a permutation and there is no repetition of dimension in the vector. The second condition ensures that the mapping specified maps dimensions in K neatly onto dimensions in D. To test this condition however requires the successful calculation of what is know as the c vector.

3.3.2 The c vector

The c vector assigns sections of the source space[2] to dimensions in the target space[3]. The reason for assigning sections of the source space to the target space is to ensure that the mapping onto the target space does not exceeded the size of the dimension being mapped to.

To use an example, the correct c vector derived for the equations in 3.4 to assign all dimensions in the k-tile space K onto a single dimension of the device space D would be:

$$c = (0, 3) \tag{3.5}$$

The length of c always is one more than the number of dimensions in the target space, in this case D and only contains numbers which do not exceed the number of dimensions in the source space which is K. The first item in the c vector is always 0, referring to the first dimension in the source space. After that the numbers only increase, representing slices of the source space to be mapped onto the target space. The c vector in 3.5 defines the entire index space of K to be mapped onto a single dimension in D. This would

[2]The source space can be either a data space or a k-tile space.

[3]The target space can be a k-tile space or a device space.

have not been the case if D had more than one dimension. The example in 3.6 shows the mapping from a 4 dimensional space onto a 2 dimensional space.

Calculating c in 3.6 involves looking at the size of the dimensions in K and D from left to right. The first dimension in K has the size 2, and the first dimension in D has the size 4. since 2 is not equal to 4, we move to the next dimension in K, multiplying this dimension by the product of the previous dimensions examined. In doing this, we have the number 4 which equals the size of the first dimension in D. The value 2, representing the second dimension in K is now put in c. The dimension that is being examined in D is now moved to the right, resulting in the second dimension of D being compared. The same process now starts over, where the size of the third dimension in K is examined to see if it equals D, which it doesn't. The dimension in K being examined moves to the right, and the size of the forth dimension in K is multiplied by the size of the third dimension and compared to D. The two are equal, and the value 4, representing the forth dimension is put in c.

$$\begin{aligned} K &= [2,2,2,2] \\ c &= (0,2,4) \\ D &= [4,4] \end{aligned} \tag{3.6}$$

The permutation of K by the m vector would result in the dimensions in K being used in a different order from this. For now, the vector $m = (0,1,2,3)$.

The diagram in figure 3.3 shows more clearly how c works in this example.

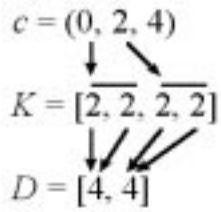


Figure 3.3: Shown is the relationship of K to D, defined by c. The first value of c is always 0 and indicates the first dimension of K. The final value of c is the last dimension in the space K.

As can be seen, the calculation of c involves finding the product of each of the dimensions in the target space to make one of the dimensions in the source space. The c vector must however satisfy the following conditions for

all $i : 0 \leq i < (Z)$ such that:

$$\begin{aligned} c_{i+1} &\geq c_i && (3.7) \\ d_i &= \prod_{j=c_i}^{c_{(i+1)-1}} k_{m(j)} && (3.8) \\ c_{(D)} &= (K) && (3.9) \end{aligned}$$

Equation 3.7 defines that the values of c are in a strictly increasing order. Equation 3.8 insures that the dimensions of K fit into the dimensions of D. Finally equation 3.9 ensures that all the dimensions in K are assigned to dimensions in D, in that the final value in c is the number of dimensions in K.

Deriving the c vector can be done by using the formula in 3.10. This does not appear in Fletcher (1993).

$$\begin{aligned} n : 1 &\leq n \leq (D) \\ c_o &= 0 \\ c_n &= j \text{ where } \prod_{i=c_{(n-1)}}^{j} k_{m(i)} = d_{n-1} && (3.10) \end{aligned}$$

3.3.3 The non-implicit k-tile mapping

With the various elements of performing a mapping now covered with the m vector and the c vector, it is now possible to define some actual mapping techniques and equations. The first type of mapping to be discussed is the *non-implicit mapping*. This mapping is dependent on the m vector and in the k-tile format is performed from the k-tile space K onto the device space D. The m vector defines a permutation of the space K. If a c vector can be derived for the mapping from K to D, as described section 3.3.2 then the k-tile mapping from K to D is defined as:

$$\begin{aligned} \text{kmap}(K,D,m) &= K \rightarrow D \\ D(u_0, \ldots, u_{((Z)-1)}) &= K(w_0, \ldots, w_{((K)-1)}) \\ \text{where } u_i &= \sum_{l=c_i}^{c_{(I+1)-1}} w_{m(l)} \times \prod_{j=c_i}^{l-1} k_{m(j)} && (3.11) \end{aligned}$$

Applying equation 3.11 results in the mapping of the index space of K onto the index space of D.

3.3.4 The implicit k-tile mapping

The *implicit* mapping is performed from the data space A onto the k-tile space K and is essentially the same as the non-implicit mapping, except the mapping is not dependent on the m vector. Instead m is assumed to be the identity $m = (0, 1, \ldots, n)$ where $n = (A)$. The calculation of c is performed in the same way as it was with the non implicit mapping, with the exception of the m vector is not used. The calculation of c must satisfy the conditions in 3.12 and its calculation is shown in equation 3.13.

$$\begin{aligned} c_{i+1} &\geq c_i \\ k_i &= \prod_{j=c_i}^{c_{(i+1)-1}} a_j \\ c_{(k)} &= (A) \end{aligned} \tag{3.12}$$

$$\begin{aligned} n:1 &\leq n \leq (K) \\ c_o &= 0 \\ c_n &= j \text{ where } \prod_{i=c_{(n-1)}}^{j} a_i = k_{n-1} \end{aligned} \tag{3.13}$$

The implicit k-tile mapping can now be defined as:

$$\begin{aligned} \text{ikmap}(A,K) &= A \rightarrow K \\ K(u_0, \ldots, u_{((K)-1)}) &= A(w_0, \ldots, w_{((A)-1)}) \\ \text{where } u_i &= \sum_{l=c_i}^{c_{(I+1)-1}} w_l \times \prod_{j=c_i}^{l-1} a_j \end{aligned} \tag{3.14}$$

The thing to note about both the non-implicit and implicit mappings discussed thus far is that they are only good when the number of dimensions being reduced in the mapping, for example a mapping from $K[2, 2, 2, 2]$ to $D[4, 4]$ where K has more dimensions than D. These types of mappings are referred to as *reduction* mappings. It is often the case however in a k-tile mapping that the number of dimensions in the target space is greater than the source space. Such mappings will be called *expansion* mappings.

Reduction and expansion mappings are not defined in Fletcher 1993, instead the use of a *inverse mapping* is used to described an expansion mapping. The inverse mapping is a non-implicit mapping done in reverse. This is a valid way to perform the mapping, but not practical for an implementation of the k-tile format. This is because the k-tile format is made of an

implicit and non-implicit mapping, performing one of these mappings in reverse would require an implementation to map all index spaces of the data space A onto the device space D. The reason for this is that if the mapping of a single address in A is done using an inverse mapping from A to K, then it is impossible to know which address in K corresponds to the given address in A to be mapped without searching through all addresses in K. As a result, the entire index space of K needs to be mapped onto A to determine what the corresponding address is. This is inefficient if only a hand full of index spaces in A need to be mapped and for this reason, I introduce the concept of expansion mappings. Expansions mappings are the equivalent of the inverse mapping, but are done in the same direction to the implicit and non-implicit reduction mappings discussed.

3.3.5 Expansion mappings

Expansion mappings will be discussed, with a number of new formula derived. Expansion mappings map from a source space to a target space, where there exists more dimensions in the target space. In order to perform these mappings, the calculation of the c vector is first required. The equation for the calculation of c for a non-implicit mapping from K to D is shown in equation 3.15.

$$\begin{aligned} n:1 &\leq n \leq (K) \\ c_o &= 0 \\ c_n &= j \text{ where } \prod_{i=c_{(n-1)}}^{j} d_i = k_{m(n-1)} \end{aligned} \tag{3.15}$$

The *non-implicit expansion* mapping is performed using different formula from the reduction mapping. Equation 3.16 shows how to derive a non-implicit expansion mapping from K to D.

$$\begin{aligned} \text{ekmap}(K,D,m) &= K \to D \\ D(u_0,\ldots,u_{(Z)}) &= K(w_0,\ldots,w_{(K)}) \\ u_i &= W \textbf{ mod } d_i \\ \text{where} & \\ i &= c_l \ldots c_{(L)+1} \\ l &= 0 \ldots (K) \\ W &= w_{m(l)} \div U \\ U &= \prod_{j=c_l}^{i} d_j \end{aligned} \tag{3.16}$$

The *implicit expansion* mapping is done the same way, with the exception that m is not used. Equation 3.17 shows how to derive the implicit expansion mapping from A to D.

$$\begin{aligned} \text{eikmap}(A,K) &= A \to K \\ K(u_0,\ldots,w_{(K)}) &= A(w_0,\ldots,w_{(A)}) \\ u_i &= W \textbf{ mod } k_i \\ \text{where} & \\ i &= c_l \ldots c_{l+1} \\ l &= 0 \ldots (A) \\ W &= w_l \div U \\ U &= \prod_{j=c_l}^{i} k_j \end{aligned} \tag{3.17}$$

Calculating the c vector for 3.17 is also done the same way as a non-implicit expansion mapping, except minus the m vector. The Equation for the calculation of c for an implicit mapping from A to K is shown in equation 3.18.

$$\begin{aligned} n:1 &\leq n \leq (A) \\ c_o &= 0 \\ c_n &= j \text{ where } \prod_{i=c_{(n-1)}}^{j} k_i = a_{n-1} \end{aligned} \tag{3.18}$$

To demonstrate expansion mappings, take the mapping from a data space with the shape $A = [4,4]$ onto a k-tile space with the shape $K = [2,2,2,2]$. There are clearly more dimensions in K such that $(K) > (A)$. By deriving the c vector using equation 3.15 we get $c = (0,2,4)$. This is now used in the implicit expansion mapping equation to map the address in A onto K. Each address in A is now mapped to K.

3.4 The k-tile format definition

With the implicit reduction, non-implicit reduction, implicit expansion and non-implicit expansion mappings defined, it is possible to define the k-tile format. The format itself is describe by the 3 spaces A,K,D and the vector m. The definition of these spaces as stated earlier is:

- A The shape of the data space

- K The shape of the k-tile space
- m The mapping between the k-tile space and the device space
- D The shape of the device space

The k-tile mapping itself is made up of an implicit mapping from A to K and a non-implicit mapping from K to D. Fletcher defines the mapping from A to K to be an inverse implicit mapping and the mapping from K to D a non-implicit mapping.

Described here will be a new k-tile format which uses the newly defined expansion and reduction mappings to allow any combination of expansion and reduction mappings in the k-tile format. The use of an implicit mapping from A to K and a non-implicit mapping from K to D remains, but these mappings can be either expansion or reduction in nature. The two mappings are defined as g_{AK} and g_{KD} and are described by equations 3.19 and 3.20 respectively.

$$\begin{aligned} g_{AK} &: A \rightarrow K, \text{ where} \\ g_{AK} &= kmap(A, K), \text{ if } (A) \geq (K) \\ g_{AK} &= ekmap(A, K), \text{ if } (A) < (K) \end{aligned} \tag{3.19}$$

$$\begin{aligned} g_{KD} &: K \rightarrow D, \text{ where} \\ g_{KD} &= ikmap(K, D, m), \text{ if } (K) \geq (D) \\ g_{KD} &= eikmap(K, D, m), \text{ if } (K) < (D) \end{aligned} \tag{3.20}$$

By using the definitions of g_{AK} and g_{KD} given it is now possible to define the k-tile mapping as:

$$ktile(A, K, m, D) = g_{AK} \rightarrow g_{KD} \tag{3.21}$$

3.5 Example k-tile mappings

In order to clarify the k-tile format further, I shall now examine a number of example mappings, using the k-tile format notation. The derived c values will also be shown, but these do not need to be specified as they should ideally be derived by any program using the k-tile format.

Each of the data sets represented here are filled with numbers so that the movement of data in each of the multi-dimensional arrays can be seen. The last k-tile examples will be pictures which show the effect of some of the more interesting k-tiles.

3.5.1 Simple 1D mapping

The first mapping shown here is possibly the most simplest mapping that can be done with the k-tile format. Figure 3.4 represents a mapping from a single dimensional array onto another single dimensional array. The mapping being performed is an identity mapping with $m = (0)$ so there is no reordering of the elements in the array. In figure 3.4 the c vectors derived for A to K and K to D are shown. Also shown is the contents of A, K and D, showing different aspects of the mapping from A to D.

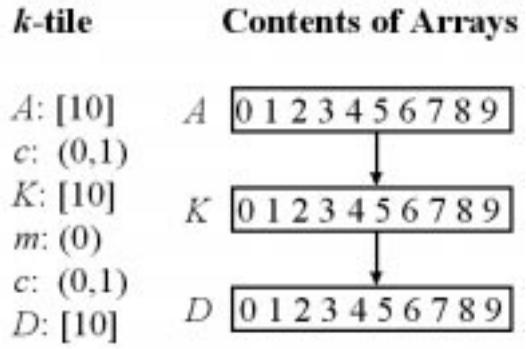


Figure 3.4: This represents a very simple mapping of a single dimensional array being mapped onto another single dimensional array. the mapping vector m specifies an identity mapping.

3.5.2 Column major ordering

In figure 3.5 a single dimensional array is mapped onto a 2 dimensional array. This 2 dimensional array is then mapped onto a 2 dimensional array with the same shape. The mapping vector m which is used to map K onto D is the identity. This results in a column major ordering where columns of the 2 dimensional array are mapped first.

3.5.3 Row major ordering

In figure 3.5 a single dimensional array is mapped onto a 2 dimensional array which is then mapped onto the target 2 dimensional array. The mapping vector m used was the identity mapping. The other mapping possible is now shown in 3.6.

By changing the mapping vector m, the behaviour of the mapping is altered. Instead of columns being mapped to columns as in the 3.5, rows are

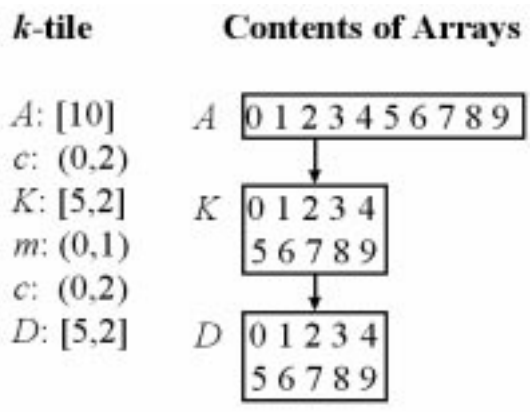


Figure 3.5: This represents a mapping of a single dimensional array onto a multi-dimensional array. The mapping vector m specifies a column major ordering.

being mapped to columns. This results in a row major ordering of the array. Figure 3.6 shows the result of the new m vector.

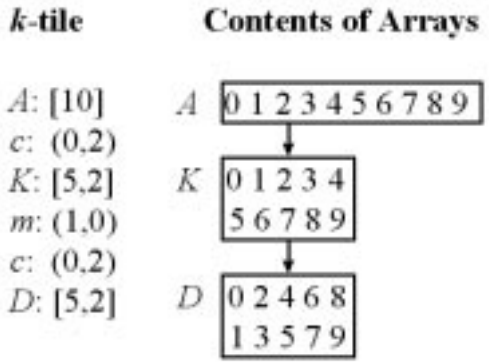


Figure 3.6: The single dimensional array becomes a 2 dimensional array which is mapped onto a third 2 dimensional array using row major ordering

3.5.4 Tiled mapping

This example is one of the more interesting and useful k-tiles. Figure 3.7 shows the mapping of a 2 dimensional array mapped onto a 4 dimensional array and then onto a 2 dimensional array. The mapping that results is a tiling of the original 2 dimensional array. This can be seen in 3.7 by moving along the top row of D. The numbers 0,1,4,5 correspond to the top left part of the original array A. The second line of D is the top right part of the

original and so on. This tiling of the array has a number of uses in computer science, especially with 2 and 3 dimensional images.

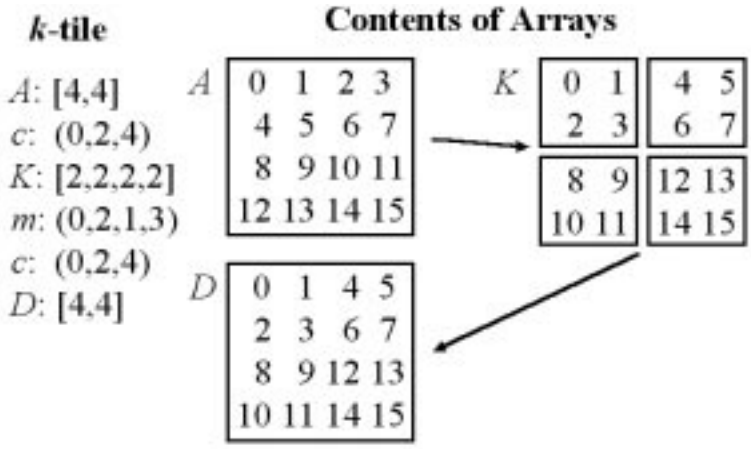


Figure 3.7: Demonstrating a more interesting k-tile is a 2 dimensional array being mapped onto another 2 dimensional array. The mapping that results is a tiling of the original array.

3.5.5 Crinkle mapping of an image

To further demonstrate the previous k-tile in figure 3.7 requires the use of a larger data set. Images provide an excellent way to view k-tiles as it is easier to see at a glance the reordering of data that the k-tile is performing. Figure 3.8 shows a single image transformed by an enlarged version of the k-tile in figure 3.7. The original image on the left is broken up into a grid of 18×18 tiles by the k-tile. The size of each tile is 18×18, but this does not necessarily have to be the case. The size of the grid and each tile can be anything that will divide into the original image evenly.

3.5.6 Crinkle mapping into tiles

Figure 3.9 demonstrates the same image in figure 3.8 mapped using a slightly different k-tile. The result is a 3×3 grid of 108×108 size tiles. The modification made to the k-tile in figure 3.8 is the shape of K. The original k-tile in figure 3.8 has the shape $K = [18, 18, 18, 18]$. In figure 3.9 this becomes $K = [108, 3, 108, 3]$. It can now be seen in the new k-tile that the size of the tile is dependent on the first and third dimensions in K, and the size of the grid made up of the tiles is dependent on the second and forth dimensions in K.

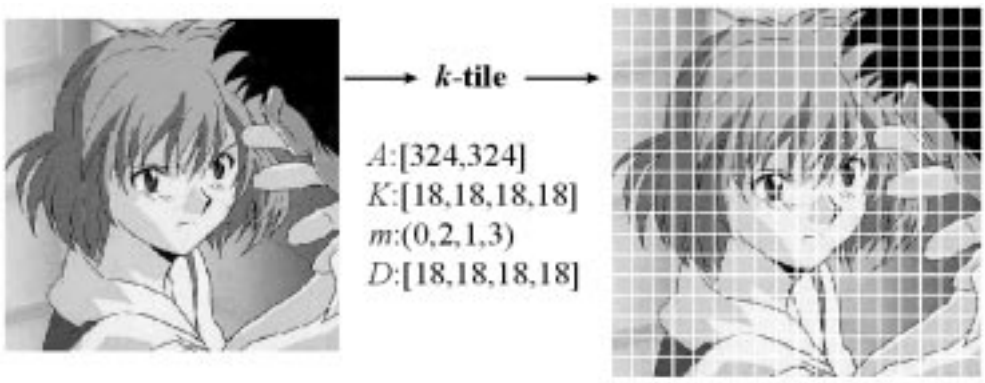


Figure 3.8: The k-tile applied to the original image on the left results in a tiling of the image shown on the right. This tiling is described as an 18×18 grid of 18×18 tiles.

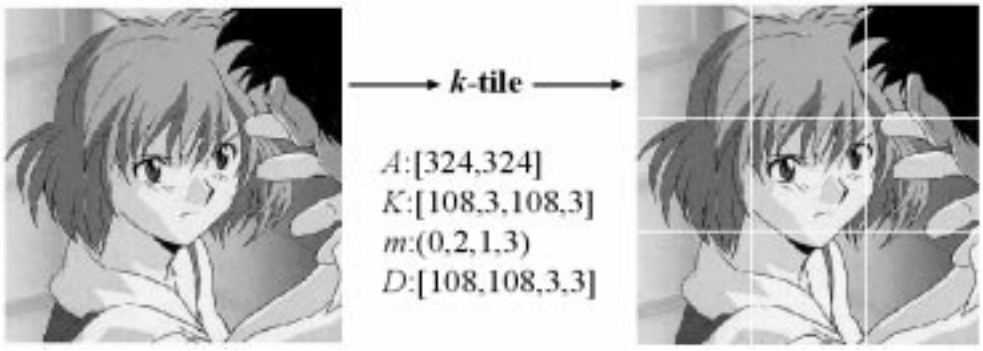


Figure 3.9: The k-tile is applied to the original image on the left, resulting in the tiling of the image on the right. This tiling is described as a 3×3 grid of 108×108 tiles.

3.6 Summary

This chapter has described the fundamentals for the mapping between two multi-dimensional arrays using the k-tile format. The spaces used by the k-tile format have been introduced as the data space, the k-tile space and the device space. These spaces are assigned the letters A, K and D respectively.

The mapping between these spaces are described as implicit mappings representing the mapping from A to K and the non-implicit mapping from K to D which is dependent on a mapping vector m. Both of these mappings come in two different flavours, the expansion mapping and the reduction mapping. This makes a total of 4 different mappings which form the underlying frame work of the k-tile format. In order to perform these mappings

however requires the calculation of the c vector which assigns sections of dimensions from a source space to a single dimension in a target space.

The k-tile mapping is hence defined as an implicit expansion or reduction mapping from A to K and a non-implicit expansion or reduction mapping from K to D, using the mapping vector m. The use of an expansion or reduction mapping in both case is dependent on the number of dimensions in A with respect to K and the number of dimensions in K with respect to D.

Finally this chapter has given a number of examples of k-tile mappings. The ability of the k-tile format to work with much larger data sets is then demonstrated with the application of the k-tile format to images. These images show the effect of various k-tiles better than smaller data sets.

Chapter 4

Extensions to the k-tile format

The k-tile format in its most basic form can perform a number of mappings. However there exist many regular mappings that the k-tile format cannot specify. By extending the k-tile format these mappings can be performed within the k-tile format. The following extensions to the k-tile format to be presented here are:

- *Empty* dimensions extend the size of a space by adding extra dimensions
- *Sense* indication allows the reversal of the storage order of elements
- *Templates* increase the size of dimensions, adding extra empty storage
- *Offsets* shift the mapping inside a dimension a set amount
- *Replication* makes a second copy of a dimension, mapping it to an empty dimension
- *Subsections* allow only part of the k-tile index space to be mapped.

These extensions will be described with changes made to the extensions from the work of Fletcher [Fletcher, 1993]. Most of these changes have resulted from the building of an implementation, which revealed a number of problems with the k-tile extensions. Subsections represent a new extension to the k-tile format described here for the first time.

With the introduction of extensions to the k-tile format, the mapping from the data space onto the device space is no longer a one-to-one mapping since there can be more data in the device space than in the data space.

4.1 Empty dimensions

A k-tile mapping works by mapping the data space A onto the device space D. There may arise a situation where additional space needs to be added to the device space D. The *Empty* dimension extension allows for extra space to be added to the k-tile space K or the device space D[1]. To demonstrate this, figure 4.1 shows a k-tile mapping where K has had an additional dimension added.

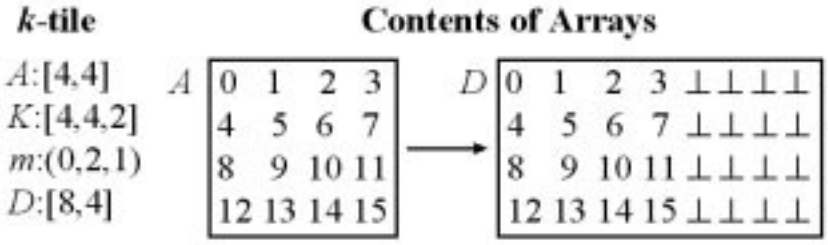


Figure 4.1: This represents an example of a empty dimension in use. K has had an additional dimension added to it, increasing the storage from A. The additional storage is assigned the empty symbol $\perp$.

The empty storage symbol $\perp$ indicates the empty space created. In an implementation the $\perp$ symbol would be replaced with 0. This is the case with the image in figure 4.2 which is the familiar 324×324 image with empty storage added, in a simular fashion to figure 4.1.

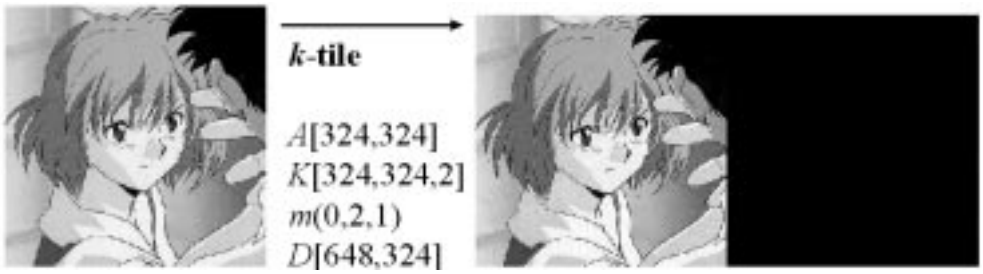


Figure 4.2: This represents an example of the empty dimension extension used in an image. $\perp$ is replaced with the value 0 which is black.

Implementation of the empty dimension extension requires the removal of a restriction on the definition of c. The new updated definition of c for

[1]This differs from the work of Fletcher [Fletcher, 1993], where only the k-tile space could have extra space.

the implicit mapping from A to K is:

$$c_{i+1} \geq c_i \tag{4.1}$$

$$k_i = \prod_{j=c_i}^{c_{(i+1)-1}} a_j \tag{4.2}$$

$$c_{(K)} \leq (A) \tag{4.3}$$

Equation 4.3 defines that only the first $c_{(A)}$ dimensions of K are needed to calculate the c vector. This change to the definition of c does not effect equations 3.10 and 3.13 which calculate the c vector. The redefined reduction implicit mapping from A to K is now shown in equation 4.4.

$$\begin{aligned} \text{ikmap}(A,K) &= A \to K \\ K(u_0, \ldots, u_{(D)-1}) &= \begin{cases} \perp & \text{if } u_i > 0 \text{ for } i \geq c_{(K)} \\ A(w_0, \ldots, w_{(A)-1}) & \text{otherwise} \end{cases} \\ \text{where } u_i &= \sum_{l=c_i}^{c_{(I+1)-1}} w_l \times \prod_{j=c_i}^{l-1} k_j \end{aligned} \tag{4.4}$$

The redefined expansion implicit mapping from A to K has the same modification as shown in equation 4.5.

$$\begin{aligned} \text{eikmap}(A,K) &= A \to K \\ K(u_0, \ldots, w_{(K)}) &= \begin{cases} \perp & \text{if } u_i > 0 \text{ for } i \geq c_{(K)} \\ A(w_0, \ldots, w_{((A)-1)}) & \text{otherwise} \end{cases} \\ u_i &= W \textbf{ mod } k_i \\ \text{where} & \\ i &= c_l \ldots c_{l+1} \\ l &= 0 \ldots (A) \\ W &= w_l \div U \\ U &= \prod_{j=c_l}^{i} k_j \end{aligned} \tag{4.5}$$

The non-implicit mapping from K to D also requires the restriction on the definition of c to be removed. The redefined definition of c for the non-implicit mapping from K to D is:

$$c_{i+1} \geq c_i \tag{4.6}$$

$$d_i = \prod_{j=c_i}^{c_{(i+1)-1}} k_j \tag{4.7}$$

$$c_{(D)} \leq (K) \tag{4.8}$$

The redefined reduction non-implicit mapping from K to D is now shown in equation 4.9.

$$\begin{aligned}
\text{kmap}(K,D,m) &= K \to D \\
D(u_0,\ldots,u_{((D)-1)}) &= \begin{cases} \perp & \text{if } u_i > 0 \text{ for } i \geq c_{(D)} \\ K(w_0,\ldots,w_{((K)-1)}) & \text{otherwise} \end{cases} \\
\text{where } u_i &= \sum_{l=c_i}^{c_{(I+1)-1}} w_{m(l)} \times \prod_{j=c_i}^{l-1} k_{m(j)}
\end{aligned} \tag{4.9}$$

The redefined expansion non-implicit mapping requires a simular modification and is defined in equation 4.10.

$$\begin{aligned}
\text{ekmap}(K,D,m) &= K \to D \\
D(u_0,\ldots,u_{(Z)}) &= K \to D \\
D(u_0,\ldots,u_{((D)-1)}) &= \begin{cases} \perp & \text{if } u_i > 0 \text{ for } i \geq c_{(D)} \\ K(w_0,\ldots,w_{((K)-1)}) & \text{otherwise} \end{cases} \\
u_i &= W \textbf{ mod } d_i \\
\text{where} & \\
i &= c_l \ldots c_{(L)+1} \\
l &= 0 \ldots (K) \\
W &= w_{m(l)} \div U \\
U &= \prod_{j=c_l}^{i} d_j
\end{aligned} \tag{4.10}$$

Using these modified implicit and non-implicit equations it is possible to use empty dimensions in the k-tile format.

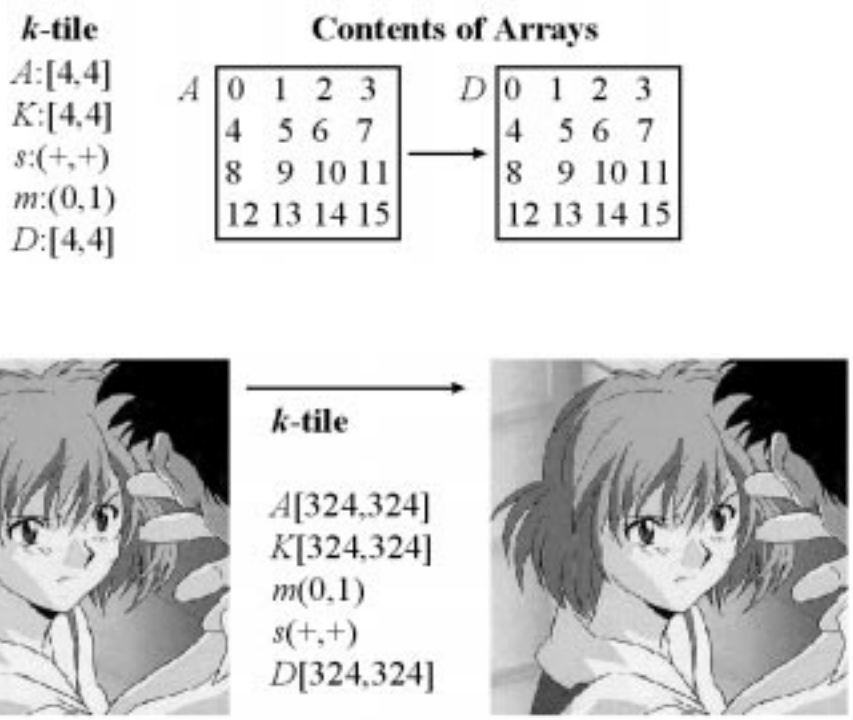
4.2 Sense indication

The k-tile format makes it possible to reverse the order of storage for dimension in the k-tile space. This is achieved by indicating the order of storage for each dimension being mapped. A "+" or "-" symbol can be used to indicate whether or not the storage order will be reversed, where "-" indicates that it is to be reversed. This is useful for performing rotations in multiples of 90^o to a data set. The *sense* of a k-tile is represented by a vector called s which consists of "+" and "-"' symbols.

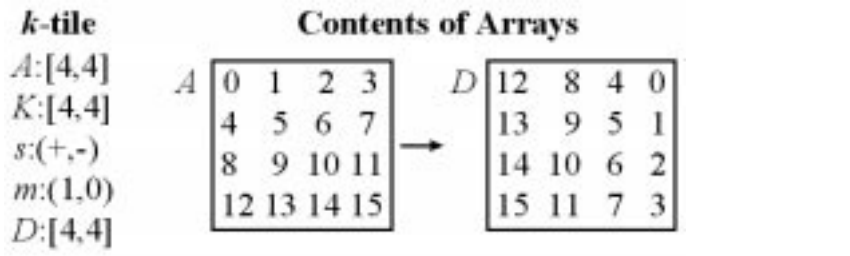
The s vector is only associated with the k-tile space, and has the same length as the mapping vector m. The following examples describe a 0^o rotation, a 90^o rotation, a 180^o rotation, and a 270^o rotation. The text examples

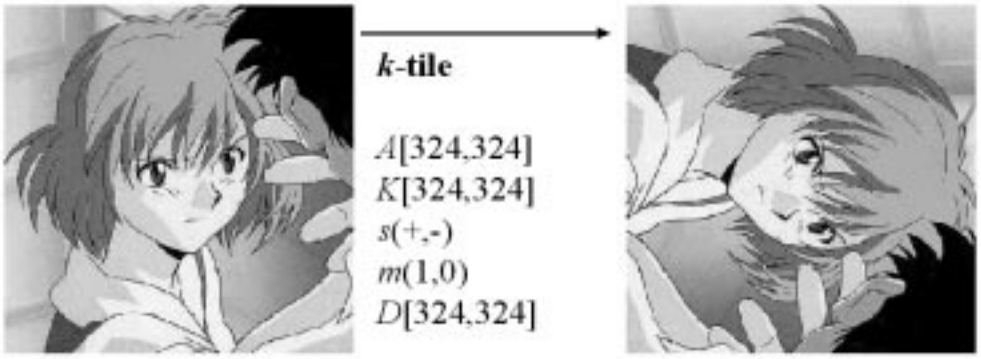
are from Fletcher [Fletcher, 1993] and, the images were created using the implementation of this extension.

- 0^o rotation:



- 90^o rotation:





- 180^o rotation:

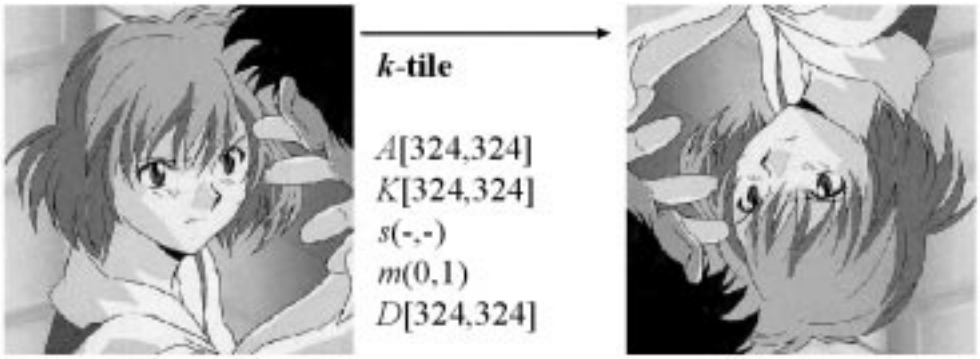
k-tile
A:[4,4]
K:[4,4]
s:(-,-)
m:(0,1)
D:[4,4]

Contents of Arrays

A				
	0	1	2	3
	4	5	6	7
	8	9	10	11
	12	13	14	15

→

D				
	15	14	13	12
	11	10	9	8
	7	6	5	4
	3	2	1	0



- 270^o rotation:

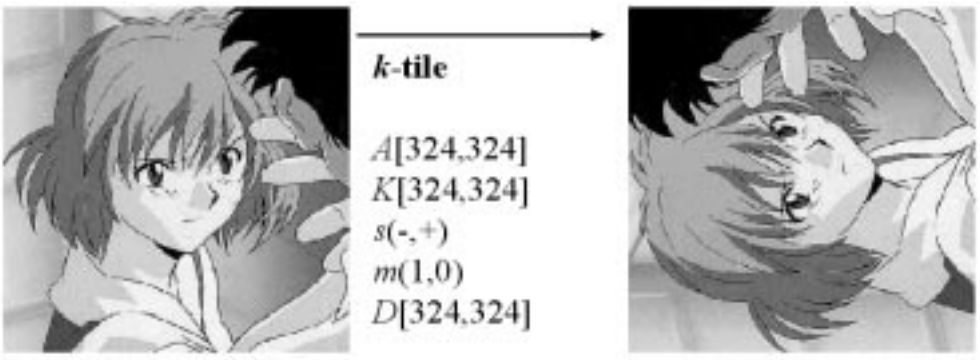
k-tile
A:[4,4]
K:[4,4]
s:(-,+)
m:(1,0)
D:[4,4]

Contents of Arrays

A				
	0	1	2	3
	4	5	6	7
	8	9	10	11
	12	13	14	15

→

D				
	3	7	11	15
	2	6	10	14
	1	5	9	13
	0	4	8	12



A formal definition for the s vector is:

$$s = (s_0, \ldots, s_{(K)-1}), \text{ where } S_i \in \{+, -\} \forall i, 0 \leq i \leq (K) - 1$$

The reduction non-implicit k-tile mapping with sense and empty k-tile dimensions is now defined as:

$$\begin{aligned}
\text{ikmap}(K,D) &= K \to D \\
D(u_0, \ldots, u_{(D)-1}) &= \begin{cases} \perp & \text{if } u_i > 0 \text{ for } i \geq c_{(D)} \\ K(w_0, \ldots, w_{(K)-1} & \text{otherwise} \end{cases} \\
\text{where } u_i &= \sum_{l=c_i}^{c_{(i+1)-1}} \begin{cases} w_{m(l)} & \text{if } s_l = + \\ k_{m(l)} - w_{m(l)-1} & \text{if } s_l = - \end{cases} \times \prod_{j=c_i}^{l-1} k_{m(j)}
\end{aligned} \tag{4.11}$$

The expansion implicit k-tile mapping with sense and empty k-tile dimensions is now defined as:

$$\begin{aligned}
\text{ekmap}(K,D,m) &= K \to D \\
D(u_0, \ldots, u_{(Z)}) &= K \to D \\
D(u_0, \ldots, u_{((D)-1)}) &= \begin{cases} \perp & \text{if } u_i > 0 \text{ for } i \geq c_{(D)} \\ K(w_0, \ldots, w_{(K)-1}) & \text{otherwise} \end{cases} \\
u_i &= W \textbf{ mod } d_i \\
&\text{where} \\
i &= c_l \ldots c_{(L)+1} \\
l &= 0 \ldots (K) \\
W &= \begin{cases} w_{m(l)} & \text{if } s_l = + \\ k_{m(l)} - w_{m(l)-1} & \text{if } s_l = - \end{cases} \div U
\end{aligned}$$

$$U = \prod_{j=c_l}^{i} d_j \tag{4.12}$$

4.3 Templates

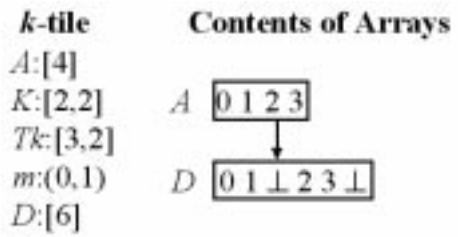
When working with a large data set, it is useful to be able to include empty padding around the data. This may need to be done if the length of a dimension is a prime number or is not multiple of the dimensions of the target space, and thus cannot be mapped. Another reason for including padding is to add empty storage space around the data. This may be useful for performing neighbourhood operations.

Templates allow the padding of data, changing the shape of the space they are applied to. Unlike an empty space however, the number of dimensions is maintained. A separate space is used for A, K and D to specify the template in the k-tile format. These spaces are Ta, Tk and Td respectively. A formal definition of these spaces where Z represents A, K and D is:

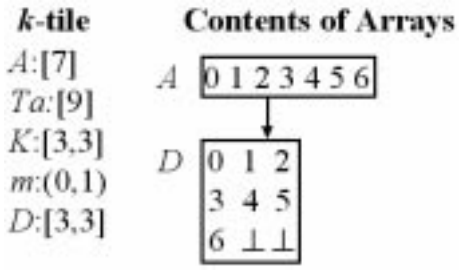
$$\begin{aligned} Z &= (z_0, \ldots, z_{(Z)-1}) z_i \in \mathbb{N} \forall 0 \leq i \leq (Z) - 1 \\ T_Z &= (t_0, \ldots, t_{(Z)-1}) t_i \in \mathbb{N} \forall 0 \leq i \leq (Z) - 1 \\ \text{where} & \\ t_i &\geq z_i \text{ for all } i, 0 \leq i \leq (Z) - 1 \end{aligned}$$

The following examples show mappings using these template spaces.

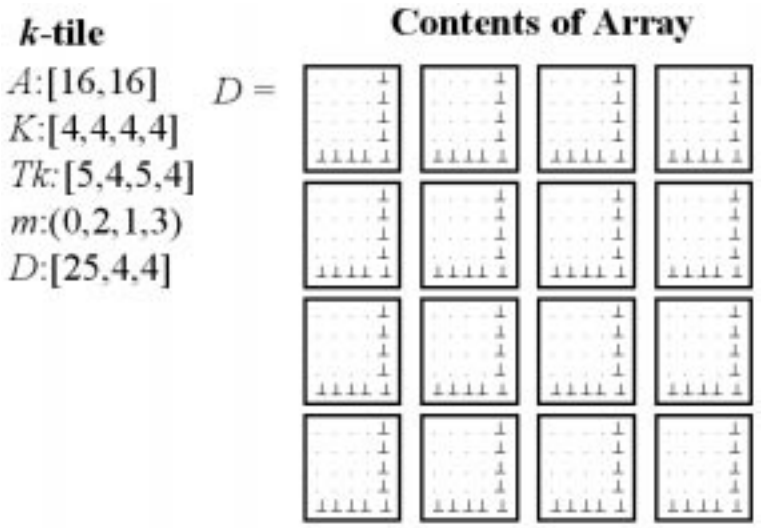
- A 4-element array, with empty space inserted at every second element



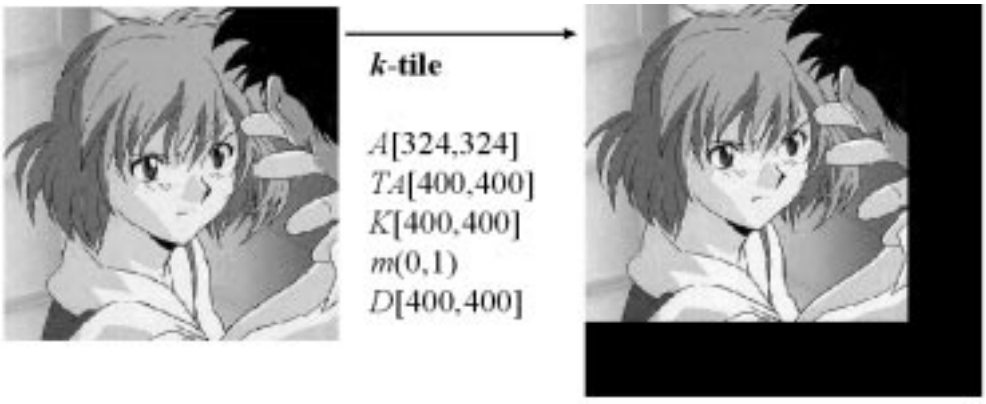
- A 7-element array mapped to a 9 element array and into a 3×3 device



- A crinkle mapping can be adjusted to add padding for neighbourhood operations.



- A template mapping that adds padding around an image



The implementation of templates is relatively straightforward, and does not require any modifications to the existing formula for the implicit and non-implicit mappings. The mapping from A to K is achieved by performing an implicit mapping of Ta onto K while using the index space of A. The remainder of the mapping is achieved in a similar fashion. These mappings can be illustrated by the following diagram:

$$\boxed{A} \rightarrow \boxed{T_A} \rightarrow \boxed{K} \rightarrow \boxed{T_K} \rightarrow \boxed{D} \rightarrow \boxed{T_D} \tag{4.13}$$

Diagram 4.13 shows the mapping of A onto T_A then T_A onto K and so on. Any implementation of the k-tile format will use the mapping of Ta onto K for the implicit mapping, and the mapping of Tk onto D for the non-implicit mapping. The index space used is not from Ta or Tk but instead from A and K. This is because Ta and Tk represent empty space which does not exist in the original data space and should not need to be mapped. The final mapping of D onto Td is achieved in the same manner, where the size of the device space is represented by Td, but the index space of D is used.

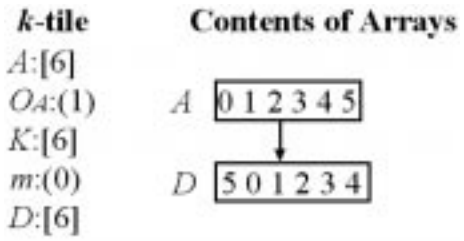
4.4 Offsets

Offsets represents a translation of the index space in a single dimension. These translations can be specified for each space in the k-tile format. This gives rise to 6 additional elements in the k-tile format, namely O_A, T_{OA}, O_K, T_{OK}, O_D and T_{OD}. The definition for a single offset is:

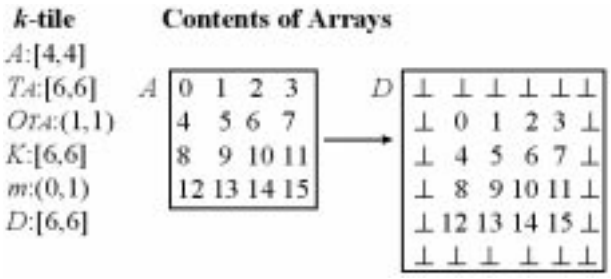
$$O = (O_0, \ldots, 0_{(Z)}), \forall O_i, 0 \leq i \leq (Z) - 1, O_i \in \mathbb{N}0 \leq O_i \leq z_i$$

This represents the offset vector for the space Z which can represent any space. Data elements that are translated beyond the end of the dimension are wrapped around so offsets do not change the shape of a space. The following examples show mappings using offsets:

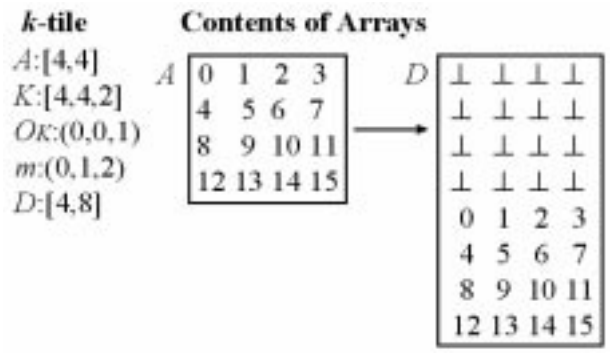
- A 6 element array, with the k-tile space offset by 1



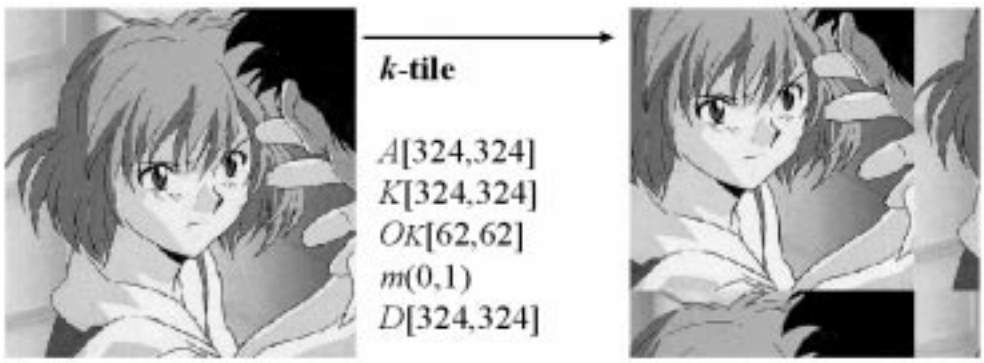
- Centering a 4x4 space in a 6x6 template



- Assigning empty space to the top of a 4x4 space



- An offset applied to an image



Applying an offset to a space is simple. Before an address is implicitly or non-implicit mapped, the offset is applied to the address if it exists. Every offset, including template offsets use the following formula to work out the

offset address:

$$\begin{aligned} Z &= (z_0, \ldots, z_{(Z)}) \\ O_z &= (o_0, \ldots, o_{(Z)}) \\ \text{where } z_i &= (z_i + o_i) \bmod z_i, \text{for } 0 \leq i < (Z) \end{aligned} \tag{4.14}$$

Using equation 4.14, offsets are easily calculated. For templates, Z is replaced with the template space, while using the index space of the space being mapped. Take for example a mapping from space A to space K where there exists a template space T_A and two offsets, O_A and O_{TA}. In this case, O_A is applied to A using equation 4.14, then O_{TA} is applied to T_A using the index space of A after O_A has been applied. The index space can now mapped using an implicit mapping onto K.

Equation 4.14 is a much simpler version of the formula given by Fletcher [Fletcher, 1993], and is the result of building an implementation for this extension.

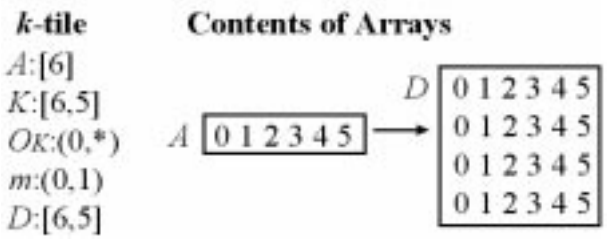
4.5 Replication

This represents the last extension to the k-tile format described by Fletcher [Fletcher, 1993]. Replication adds the "*" operator to the offset operator O_K which is specified to represent the replication of data across the dimension. The replication operator can only be used in the k-tile offset vector. A revision of O_K which includes the replication operator is:

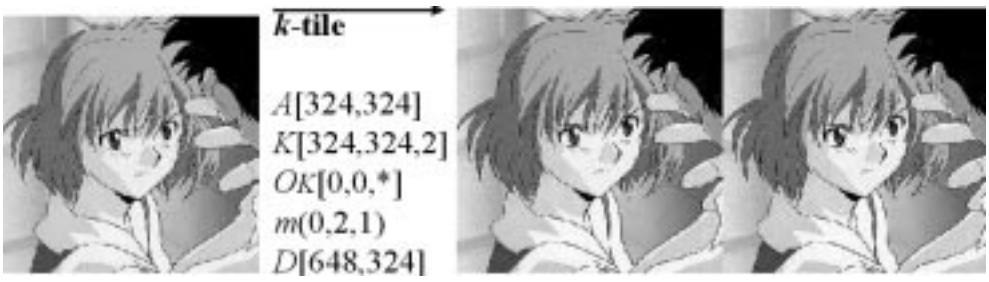
$$O_K = (O_0, \ldots, 0_{(K)-1}), \forall i, 0 \leq i \leq (K) - 1, O_i \in \{\mathbb{N} \cup *\} 0 \leq O_i \leq k_i$$

The following examples show how the new O_K operator is used:

- Mapping many copies of a single dimensional array



- Graphical example of an image mapped twice



The mathematics behind making offsets work is relatively straight forward and involves setting a single part of an address to 0 if "*" is specified. This makes the mapping of the k-tile onto the device space re-read a section of the k-tile array, replicating the data. Equation 4.15 shows the modification made to the offset equation given in 4.14.

$$\begin{aligned} K &= (k_0, \ldots, k_{(K)}) \\ O_k &= (o_0, \ldots, o_{(K)}) \\ \text{where } k_i &= \begin{cases} 0 & \text{if } O_{Ki} = * \\ (z_i + o_i) \bmod z_i, \text{ for } 0 \leq i < (K) & \text{otherwise} \end{cases} \end{aligned} \tag{4.15}$$

The use of the "*" offset in a non-empty k-tile dimension would result in the destruction of data. For this reason it is recommended that the "*" offset only be used in empty dimensions. The desiccation of data however may be useful in some circumstances, but controlling this using the "*" offset is not trivial.

The use of offsets in an implementation of the k-tile format however presents some difficulties. The primary difficulty is that the index space of the data space is not sufficient to map to the entire index space of the device space. This is because the amount of data mapped to the device space is greater than the data in the data space.

The mapping from the space A to the space T_D in equation 4.13 is no longer sufficient. A new mapping to replicate offsets is shown in 4.16.

$$\begin{aligned} &\boxed{A} \rightarrow \boxed{T_A} \rightarrow \boxed{K} \\ &\boxed{K} \rightarrow \boxed{T_K} \rightarrow \boxed{D} \rightarrow \boxed{T_D} \end{aligned} \tag{4.16}$$

The break at K in 4.16 means that the index space of A is mapped onto K and then the index space of K is mapped onto D. In any implementation

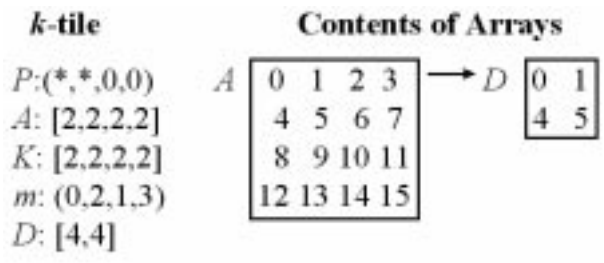


Figure 4.3: The subsection of the device space results in a partial mapping of the data space.

this will require mapping all of the data from the array A into K and then mapping this array to the device space D. This is not as efficient as the previous mapping definition in 4.13, where the entire mapping of the index space from the space A to the space D was performed at once, but the mapping defined 4.16 does allow the use of replication in the k-tile format.

4.6 Subsections

Subsections are a new concept for the k-tile format, and are described here for the first time. A subsection provides a method with which to specify that only a subset of the index space in A will be mapped. This is achieved by defining which parts of the data space A will be mapped onto D, and which will not be mapped. This reduced index space is then mapped onto the device space D. Since it is envisaged that this extension will be used for extracting subsets of a device space D which has been mapped from a data space, data will be extracted from the device space to the data space. Hence as each address from A to D is discovered, data is copied from the address found in the device space to the address in the data space. The data space represents the target array in this situation.

The letter P is assigned the subsection operator since s is already taken. The P stands for Partial mapping. A formal definition of P is:

$$P = (p_0, \ldots, p_{(A)}), \forall 0 \leq i \leq (A) - 1, 0 \leq P_i \leq a_i, P_i \in \{\mathbb{N} \cup *\}$$

Any number in the P vector causes a subset of the index spaces in A to be used. In this case, only addresses in A where $u_i = p_i$ for $A(u_0, \ldots, u_{(A)})$ where $p_i \leq a_i$. If P consisted completely of numbers, then only a single address would be mapped. If however the "*" operator for p_i was used then all addresses for u_i are used. Figure 4.3 and 4.4 show a simple k-tile where the subsection extension is used.

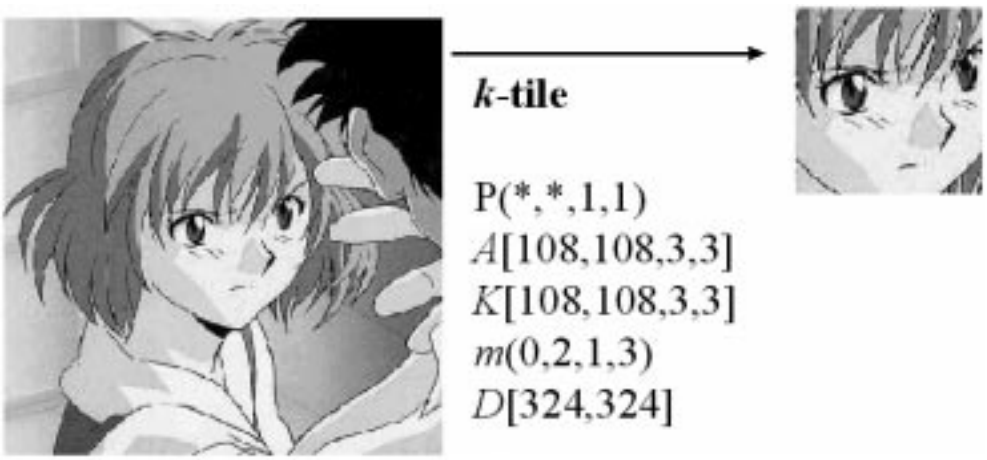


Figure 4.4: The subsection of the device space results in a small tile of the image being extracted to the data space.

An implementation which used the subsections extension to the k-tile format would perform the copying of data from the device space to the data space when ever the subsection extension was used. The reason why the mapping of data is performed this way, is because mapping from the data space to the device space, where the subsection is on the device space side results in problems. There is no way to determine if an address being mapping from the data space is in the subspace of addresses in the device space. The only way to determine this would be to map the entire index space, then filter out the addresses that do not fall into the subspace of the device space. This is very inefficient since the data space can be very large. Hence mapping only a subset of addresses to the device space and copying the contents of these addresses to the data space is more appropriate.

4.7 Summary

The k-tile format is a flexible way of mapping data from a data space A to a device space D. The extensions discussed in this chapter add even more flexibility to the k-tile format over its basic form described in chapter 3. This chapter has discussed the addition of empty space, templates and sense to the k-tile format. These additions have added a number of additional spaces to the k-tile format. All of the spaces which now define a k-tile are given in figure 4.5. Stars next to items in the k-tile format are optional and do not need to be specified. It is unlikely that a real k-tile would actually use all of

$P(p_0, \ldots, p_{(A)}) *$
$A[a_0, \ldots, a_{(A)}]$
$O_A(o_0, \ldots, o_{(A)}) *$
$T_A[t_0, \ldots, t_{(A)}] *$
$O_{TA}(o_0, \ldots, o_{(A)}) *$

$K[k_0, \ldots, k_{(K)}]$
$m(m_0, \ldots, m_{(K)})$
$s(s_0, \ldots, s_{(K)}) *$
$O_K(o_0, \ldots, o_{(K)}) *$
$T_K[t_0, \ldots, t_{(K)}] *$
$O_{TK}(o_0, \ldots, o_{(K)}) *$

$D[d_0, \ldots, d_{(D)}]$
$O_D(o_0, \ldots, o_{(D)}) *$
$T_D[t_0, \ldots, t_{(D)}] *$
$O_{TD}(o_0, \ldots, o_{(D)}) *$

Figure 4.5: A summary of all elements making up a k-tile. A "*" next to an element indicates that it is optional.

the extensions in 4.5 at the same time, but their use can greatly add to the capabilities of a k-tile mapping.

Chapter 5

k-tile oddities and k-blocking

Not all implicit and non-implicit mappings can be performed. Two problems exist which shall be referred to as *reduction expansion problem* and the *incompatible dimension problem*. Both of these difficulties are not described by Fletcher [Fletcher, 1993] and as such, they are introduced here for the first time.

The reduction expansion problem arises when mappings are both expansion mappings and reduction mappings at the same time. Such mappings cannot be performed by any of the mapping formulae described in chapters 3 and 4. This is despite the fact that it is possible to perform a mapping by building a custom formula.

The incompatible dimension problem results from a mapping from the space K to the space D where it is desirable for a device space to have a particular shape, but this shape cannot be accommodated as the dimensions in K do not multiply or divide into D. This problem is also linked to problems with building an implementation, where the data and device spaces are limited in the sizes they can be. This problem can arise when using a file on disk, which is represented as a single dimensional array. This single dimensional data space A may not represent the actual data stored in this file, which could be an array of more than one dimension.

The solution to both of these problems is *k-blocking*, which specifies two external spaces S and T where S maps onto the data space A and the device space D maps onto T.

5.1 k-blocking

Two new spaces S and T called the *source space* and *target space* respectively represent the k-blocking extension. Both S and T work the same way as the

data space A and device space D, except that they can be used to solve certain problems with both mappings and building implementations of the k-tile format.

The mapping of the k-tile format does not change when using k-blocking, except that the source multi-dimensional array begin mapped is represented by the space S and the target multi-dimensional array is represented by the space T. Both of the mappings from S onto A and D onto T are implicit so no m vector is required.

5.2 The reduction expansion problem

Implicit and non-implicit mappings can be either expansion mappings or reduction mappings. It is impossible for a reduction mapping to perform an expansion mapping and vise versa. However, mapping do exist which contain both an expansion component and a reduction component. These mappings cannot be mapped in the k-tile format, despite the two spaces being the same size, and the existence of a valid mapping of the two spaces. Figure 5.1 shows a typical expansion mapping from a 2-dimensional space onto a 4-dimensional space. Figure 5.2 shows a reduction mapping which is the same mapping in reverse. Both of the mappings in these figures are implicit. Finally figure 5.3 shows a mapping containing a mixture of the mappings in figure 5.1 and 5.2. This mapping cannot be performed and demonstrates the expansion reduction problem.

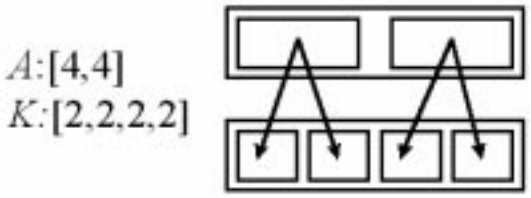


Figure 5.1: Shown here is an implicit expansion mapping, which is possible can be performed.

The mapping in figure 5.3 cannot be perfoormed, since c cannot be calculated for a combined expansion and reduction mapping.

5.2.1 The inability to derive c

In a mapping from the data space A to the k-tile space K, the coordinates of c need to be calculated by producing products of dimensions in A to match

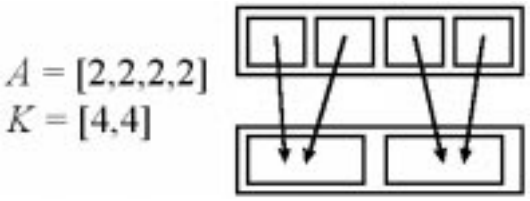


Figure 5.2: Shown here is an implicit reduction mapping, which is can also be performed.

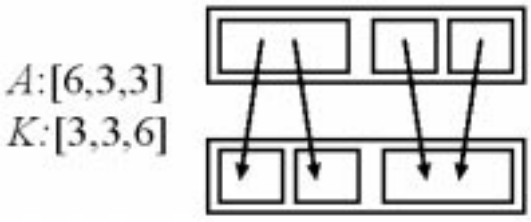


Figure 5.3: Shown here is an implicit expansion reduction mapping which cannot be performed.

a single dimension in K. In the case of an expansion mapping where the division of dimensions needs to be performed, the calculation of c can be achieved using it in reverse, that is use K instead of A, and A is used instead of K. Thus the division calculations become product calculations, allowing for the calculation of c. In the case of an expansion reduction mapping however, reversing the spaces A and K does not work, as the division of dimensions is still necessary in the calculation of c.

5.2.2 Solution to the expansion reduction problem

The solution to this problem is to use k-blocking. This introduces the S space to the mapping of A to K, now causing the mapping to be performed from S to A to K. By making S the shape of the data space A, and A equal to the shape of a single dimensional array, the results is that a single reduction expansion mapping becomes a reduction mapping followed by an expansion mapping. Figure 5.4 shows a possible method to perform the mapping in figure 5.3, using S.

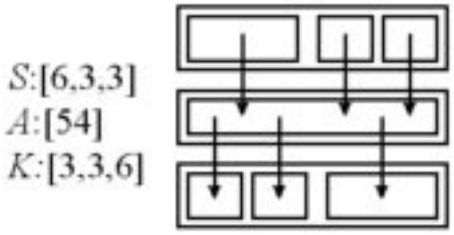


Figure 5.4: Represented here is an implicit reduction mapping followed by an expansion mapping. This mapping can be performed.

5.3 The incompatible dimension problem

The incompatible dimension problem occurs when the mapping between two spaces cannot be performed, since one dimension is not a factor of the other dimension. Thus c cannot be derived since the product of the dimensions in the space being mapped from does not equal the dimension in the space being mapped to. The following k-tile illustrates this problem:

$A : [768, 768]$
$K : [256, 3, 256, 3]$
$m : (0, 2, 1, 3)$
$D : [768, 768]$

The data space A and the device space D refer to 2 dimensional arrays. The mapping however from K to D is not possible, as the non-implicit mapping between the space K and the space D will map the first and third dimensions of k onto the first dimension of D. The size of the two dimensions combined is 65536, which is greater than the size of the corresponding dimension in D which in this case is 768. The size of the device and data space cannot be changed as they represent the shape of the multi-dimensional arrays being mapped. The introduction of the spaces S and T from k-blocking solves this problem by freeing up A and D:

$S : [768, 768]$
$A : [768, 768]$
$K : [256, 3, 256, 3]$
$m : (0, 2, 1, 3)$
$D : [589824]$
$T : [768, 768]$

The k-tile mapping is now performed from the source space S to the target space T where S and T represent the shape of the multi-dimensional arrays being mapped.

5.4 Implementation difficulties

Using S and T simplifies the implementation of the k-tile format, since the actual data and device shapes available may be limited in an implementation. Having S and T hiding the true shape of the source and target multi-dimensional arrays allows a greater freedom in the specification of the k-tile. This limitation of device and data shapes may be due to a limited set of shapes that the source and target arrays can be. Disks represent files as single dimensional arrays of bytes, which may be a limitation if the data stored in a file is multi-dimensional in nature. A mapping to a number of parallel processors is another example where the shape of the device spaces available is limited. Using k-blocking provides a solution to these difficulties by separating the actual shape of the data and device spaces from A and D in the k-tile definition.

5.5 Summary

This chapter has introduced two difficulties within the k-tile format. These are the expansion reduction problem and the incompatible dimension problem. Both of these difficulties with the k-tile format can be solved by using k-blocking, which adds two additional spaces to the k-tile format. Namely the source space S and the target space T. The source space S is implicitly mapped onto the data space A and the device space D is implicitly mapped onto T. Implementation difficulties where there exist a limitation on the shape of the data space and device space can also be overcome, so that the k-tile mappings can be performed without being limited by the implementation.

Chapter 6

The generic k-tile

A single k-tile specifies two aspects of a mapping the size of the data and device spaces and the mapping to be performed. A generic k-tile captures the mapping aspect of a k-tile, and not the shape of each of the spaces making up the k-tile.

6.1 Generics

The generic k-tile works by allowing the user to define a number of variables to describe the k-tile for many different data and device space shapes. Letters combined with numbers are used to describe the shape of the different spaces in the k-tile format. Mathematical constructs can also be used inside the definition for each of the k-tile spaces. These binary constructs include $\times$,$\div$,$+$,$-$ and MOD. Any extension to the k-tile format can also be used in this context.

A special set of variables are used in the generic k-tile format to specify the size of each dimension in a space. These variables can be used by other spaces further down in the mapping. For example, the shape of A contains the size of a number of dimensions. The size of each of these dimensions can be used by the spaces T_A, K, T_K, etc for the calculation of their shape. These special variables are called *component* variables. The definition for each of the component variables of each space in the k-tile format is shown in figure 6.1. The sense vector s, the mapping vector m, and the offset operators do not have component variables as they are vectors, not spaces. The k-blocking spaces S and T do not have component variables either, as they are used in the implementation of the k-tile format and are not visible to the user.

Any variable defined in a generic k-tile which is not one of the variables defined in figure 6.1 is called a *parameter*. Parameters are used to define

$$
\begin{aligned}
a0, \ldots, a(A) &\rightarrow \text{size of dimensions in } A \\
Ta0, \ldots, Ta(A) &\rightarrow \text{size of dimensions in } T_A \\
k0, \ldots, k(K) &\rightarrow \text{size of dimensions in } K \\
Tk0, \ldots, Tk(K) &\rightarrow \text{size of dimensions in } T_K \\
d0, \ldots, k(D) &\rightarrow \text{size of dimensions in } D
\end{aligned}
$$

Figure 6.1: Component variables define the size of dimensions in a space to other spaces further down in the mapping

the behaviour of the generic k-tile and the size of the spaces. Most generic k-tiles will require the shape of the data space to be a parameter, but it is possible for a generic k-tile to have a set shape size for the data space. In this case, only one size of data space could be used by the generic k-tile, but other aspects of the k-tile could still be controlled by parameters.

6.1.1 Examples of generic k-tiles

With components and parameters for a generic k-tile defined, it is now possible to construct examples of generic k-tiles and their use. Figure 6.2 shows an example of a generic k-tile which performs a crinkle mapping of a two dimensional data set.

$$
\begin{aligned}
&A[x, y] \\
&K[xx, a0 \div xx, yy, a1 \div yy] \\
&m(0, 2, 1, 3) \\
&D[k0, k2, k1, k3]
\end{aligned}
$$

Figure 6.2: The specification of a generic k-tile

Figure 6.2 illustrates the use of component variables to define aspects of the k-tile space and device space. The parameters that must be given to the generic k-tile are x, y, xx and yy. The shape of the data space is specified by the x and y parameters and the shape of each tile is specified by the xx and yy parameters. The component variables $a0, a1, k0, k1, k2$ and $k3$ are used by the generic k-tile to define the shape of other spaces. There is no rule that all component variables must be used, or that any need to be used at

all. If in figure 6.2 the parameters 324,324,108 and 108 were given for x,y,xx and yy respectively, the resultant k-tile would be as shown in figure 6.3.

$$A[324, 324]$$
$$K[108, 3, 108, 3]$$
$$m(0, 2, 1, 3)$$
$$D[108, 108, 3, 3]$$

Figure 6.3: The mapping that results from using x=324, y=324, xx=108 and yy=108

The k-tile produced in figure 6.3 would crinkle map the 2 dimensional array a onto a 4 dimensional device space, representing a 3×3 grid of 108×108 tiles.

Any 2 dimensional data space can be mapped by the generic k-tile in figure 6.2 provided that each dimension is a multiple of the size of tile specified. If the dimensions are not multiples of the specified tile sizes, then the generic k-tile cannot be used. It is possible however to derive a generic k-tile without this limitation. An improved version of the k-tile in figure 6.2 is given in figure 6.4.

$$A[x, y]$$
$$T_A[a0 + (xx - (x\ MOD\ xx)), a1 + (yy - (y\ MOD\ yy))]$$
$$K[xx, ta0 \div xx, yy, ta1 \div yy]$$
$$m(0, 2, 1, 3)$$
$$D[k0, k2, k1, k3]$$

Figure 6.4: A generic k-tile which performs a crinkle mapping. This generic k-tile uses templates to pad dimensions which are not divisible by the size of the tiles

This generic k-tile uses a template space to space out the size of the image, so that it can be divided by the size of each tile. This demonstrates that any extension can be used in a generic k-tile. The generic k-tile will work even if x and y are prime numbers.

The k-tile produced when $x = 450, y = 450, xx = 100$ and $yy = 100$ is shown in figure 6.5.

$A[450, 450]$
$T_A[500, 500]$
$K[100, 5, 100, 5]$
$m(0, 2, 1, 3)$
$D[100, 100, 5, 5]$

Figure 6.5: The k-tile mapping created from 6.4 when x=450, y=450, xx=100 and yy=100.

6.2 Summary

Generic k-tiles are described here for the first time as a method of defining a k-tile which can be used for any shape data and device space, without modifying the k-tile. Parameters specify to the generic k-tile what is needed to derive a k-tile to perform the mapping of a given data space to a given device space. Each parameter is comparable to a variable in a programming language and specifies aspects such as like the shape of the data space and other aspects of the mapping. Internal variables known as components define the size of a dimension in a space to the following spaces in the mapping. The user can make use of these components to simplify the definition of the generic k-tile.

Any extension can be used with a generic k-tile allowing for example the automated calculation of a template for the data space.

Chapter 7

The k-tile implementation

The implementation of the k-tile format requires the design of a package or set of packages which perform the formulae described in chapters 3 and 4. These packages then form a library of functions which can be used to perform a k-tile mapping in an application. It is important that this library be as general as possible, in order to implement the entire k-tile format and be simple to use.

This chapter describes the algorithms behind the implementation of the k-tile format developed in Java [Patrick, 1999]. The shape of the spaces in the k-tile format are described by arrays which contain the size of each dimension. Hence the implementation maps a single address from a data space onto an address in a device space. The copying of information from one array to another is left up to the user, or alternatively a subclass around the implementation specifies the data space to be mapped.

The building of a subclass is trivial and one is already provided which works for arrays consisting of bytes. This subclass can be used with or without the replication extension. Not using the replication extension saves memory, and results in a faster implementation, and as such is recommended if replications are not required.

7.1 An implementation of the k-tile format

The implementation built consists of two parts. The first part is an underlying implementation of the mappings and calculation of the c vectors required to perform the k-tile mappings. This requires the development of a number of algorithms which implement the formulas given in chapter 3.

The second part of the implementation is a program which uses these mapping elements to perform a k-tile mapping. A number of different k-tile

mappings need to be developed, since different applications have different mapping requirements. Some k-tile mappings do not require the k-tile extensions, hence a more efficient k-tile mapping implementation could be used. Other mappings require the k-tile mapping to be performed over a large data set, requiring an implementation that uses as little memory as possible. All of these k-tile implementations however, would use the same underlying mapping implementation for the different types of mappings that exist in a k-tile format.

7.2 Mapping algorithms

Listed here are 10 algorithms which perform the calculation of the various aspects of the k-tile format. This includes the calculations of the c vector, so that the implicit and non-implicit mappings can be performed. Both the implicit and non-implicit, expansion and reduction mappings are shown.

7.3 The implicit mapping

The implicit mapping is a mapping that does not require the use of the mapping vector m. Instead, dimensions are mapped "implicitly" which is the equivalent of using a mapping vector equal to $(0, \ldots, P)$ where P is the number of dimensions in the space being mapped from. Typically an implicit mapping is done from the space A to the space K in the k-tile format, but implicit mappings exist from S to A and D to T when k-blocking is used.

7.3.1 The implicit reduction mapping

A reduction mapping from the space A to the space K is a mapping, in which the number of dimensions in K, is less than the number of dimensions in A. As a result, dimensions in A are combined to form single dimensions in K. If this cannot be achieved, then the mapping is not possible, or it is not a reduction mapping. In the case of an impossible, or invalid mapping where the amount of data represented by A is equal to that represented by K, the expansion reduction problem or incompatible dimension problem may have arisen. In this case the S space should be used to make the mapping valid.

If the mapping is not a reduction mapping, then it is an expansion mapping where the number of dimensions in K is greater than A. If this is the case, then different algorithms are used to perform the mapping. These are described later in section 7.3.2.

Calculating c for an implicit reduction mapping

In order to perform an implicit mapping from A to K first requires the calculation of the c vector. In the case of a reduction mapping from A to K, the algorithm to calculate c is given in figure 7.1. This algorithm is derived from equation 3.13 in chapter 3.

```
function implicit_reductionC(A,c,K)

j = 0
c[0] = 0

for i=0...(K) loop
    product = 1

    do
      product = product * A[j]
      j=j+1
    while product != K[i]

    c[i+1] = j

    if j > (A) break
end for
```

Figure 7.1: The implicit reduction mapping performed from A to K requires a c vector which is calculated by this algorithm.

The **if** statement in figure 7.3 will cause the program to exit the loop if all of the dimensions in A have been mapped, but dimensions still exist in K. This may be the case if K is using the extra dimension extension and thus has additional dimensions.

The implicit reduction mapping

The implicit reduction mapping is a mapping of the addresses in one space into the addresses in another space. The m vector is not used in an implicit mapping since every implicit mapping is an identity mapping[1]. Figure 7.2 shows the algorithm to perform the implicit mapping from A to K.

[1] An identity mapping is where $m = (0, 1, \ldots, n)$ where n is the number of dimensions in the space being mapped from

```
function implicit_reduction(w,A,c,K,u)

u = 0
for i=0...(K) loop
    for l=c[i]..c[i+1] loop
        product = 1;
        for j=c[i]..l loop
            product = product * A[j]
        end for

        u[i] = u[i] + (w[l]*product)

    end for
end for
```

Figure 7.2: The implicit reduction mapping from A to K is performed using this algorithm.

In figure 7.2, the implicit reduction mapping algorithm maps addresses in w representing the data stored in a multi-dimensional array described by A into addresses in u representing the multi-dimensional array described by K. The extra dimension extension described in chapter 4 is handled by the algorithm in figure 7.2, provided that the length of c is one greater than the number of dimensions in K and any undefined part of c is equal to 0. This is because, when i in figure 7.2 results in $c[i+1]$ being equal to 0, a loop from $c[i]$ to 0 results. Such a loop exits immediatly in the C [Schildt, 1990] and Java programming languages. Implementation in other languages may need to verify that $c[i+1]$ is not 0 before performing the loop if this is not the case.

7.3.2 The implicit expansion mapping

The implicit expansion mapping is differs from the implicit reduction mapping in that the number of dimensions in K is now larger than the number of dimensions in A. As a result, none of the algorithms defined for the implicit reduction mapping will work. Hence a new set of algorithms need to be derived, which perform expansion mappings.

Expansion c calculation

The calculation of c for an implicit expansion mapping is performed the same way as it is for a reduction mapping, with the exception that the calculation is performed in reverse. Figure 7.3 shows the algorithm to perform the calculation of c for an implicit expansion mapping from the space A to the space K.

```
function implicit_expansionC(A,c,K)

j = 0
c[0] = 0

for i=0...(A) loop
    product = 1

    do
      product = product * K[j]
      j=j+1
    while product != A[i]

    c[i+1] = j
end for
```

Figure 7.3: The implicit expansion mapping performed from A to K requires a c vector which is calculated by this algorithm. Note that this algorithm is the exact reverse of the algorithm given in figure 7.1.

The calculation of c for an implicit mapping is the same as it was for a non-implicit mapping except that the if statement is required. The reason for this is that even if K did have extra dimensions, they would be ignored, since the algorithm loops until all dimensions of A are used. Since extra dimensions can only be added to K and D, there is no need to verify whether A has extra dimensions.

The implicit expansion mapping

With an expansion mapping the number of dimensions in K is greater than the number of dimension in A and thus, a new algorithm to perform the mapping is required. Figure 7.4 shows the algorithm to perform an implicit expansion mapping from A to K.

```
function implicit_expansion(w,A,c,K,u)

for l=0...(A) loop
    W = w[l]

    for i=c[l]..c[l+1] loop
      u[i] = W mod K[i]
      W = W / D[i]
    end for
end for
```

Figure 7.4: The implicit expansion mapping from the shape A to the shape K is performed using this algorithm. This mapping is performed when there are more dimensions in K then in A.

7.4 The non-implicit mapping

The non-implicit mapping is a mapping that requires the use of the mapping vector m which defines a permutation of the dimensions to be mapped. A non-implicit mapping is performed from the space K to the space D in the k-tile format. The following algorithms describe the various elements that make up a non-implicit mapping from K to D.

7.4.1 The non-implicit reduction mapping

A non-implicit reduction mapping is a mapping from K to D where the number of dimensions in D is less than the number of dimensions in K.

The non-implicit reduction c calculation

In the case of a reduction, non-implicit mapping, calculation of the c vector is performed by calculating the product of dimensions in D that equal the size of a dimension in K. The dimension in K used to make the comparison is determined by the m vector. Figure 7.5 shows the algorithm to perform the calculation of c for a non-implicit reduction mapping.

The number of dimensions in D is represented by (D) in the algorithm. The values stored in the arrays K and D in the algorithm are the size of each dimension making up the spaces. The actual data represented by the k-tile space K and device space D is not used in the calculation of c.

```
function non_implicit_reductionC(K,m,c,D)

c[0] = 0
j = 0

for i=0...(D) loop
    product = 1
    loop
        product = product * K[m[j]]
        j = j  + 1
    while product != D[i]
    c[i+1] = j

    if j > (K) break
end for
```

Figure 7.5: The calculation of c for the non-implicit reduction mapping shown from K to D

The non-implicit reduction mapping from K to D

With the calculation of the c vector for a non-implicit mapping, it is possible to perform the mapping from K to D. This requires an algorithm that calculates the equation 3.11 in chapter 3 which describes how to perform a non-implicit reduction mapping. The algorithm that performs this equation is shown in figure 7.6.

In figure 7.6 w represents an address in K and u represents the corresponding address in D. The mapping of the entire index space of K onto D requires running each address w in K through the algorithm in figure 7.6 to find the corresponding address u in D. In doing this, the data stored in a multi-dimensional array which has the shape represented by K can be copied to a multi-dimensional array which has a shape represented by D.

The non-implicit reduction mapping with sense

Using the sense extension discussed in section 4.2 of chapter 4 requires a modification to the algorithm given in figure 7.6. This modification is shown in figure 7.7.

Since the sense extension uses "+" and "-" symbols, where "-" indicates a reversal in the storage order of a dimension, an if statement is required to choose the appropriate mapping of a dimension in K onto D. When a reversal of a dimension is required the address in the dimension is subtracted

```
function reduction_non_implicit(w,K,m,c,D,u)

u = 0

for i=0...(D) loop
    for l=c[i]..c[i+1] loop
        product = 1;
        for j=c[i]..l loop
            product = product * K[m[j]]
        end for

        u[i] = u[i] + (w[m[l]]*product)

    end for
end for
```

Figure 7.6: The non-implicit reduction mapping from K to D is performed using this algorithm.

```
function reduction_non_implicit_sense(w,K,m,s,c,D,u)

u = 0

for i=0...(D) loop
    for l=c[i]..c[i+1] loop
        product = 1;
        for j=c[i]..l loop
            product = product * K[m[j]]
        end for

        if s[m[l]] == '+' then
            u[i] = u[i] + (w[m[l]]*product)
        else
            u[i] = u[i] + ((k[m[l]] - w[m[l]] - 1)*product)
        end if

    end for
end for
```

Figure 7.7: The non-implicit reduction mapping from K to D, using the s vector is performed using this algorithm.

from the size of the dimension minus one.

7.4.2 The non-implicit expansion mapping

A non-implicit expansion mapping is a mapping that occurs when the number of dimensions in D is greater than the number of dimensions in K for the mapping of K onto D. Different algorithms are required to perform an expansion mapping from those for a reduction mapping.

The non-implicit expansion c calculation

In the case of a non-implicit expansion mapping, the calculation of c is performed differently from the algorithm given in figure 7.5. In this case the role of K and D in the algorithm is reversed, to reflect the nature of a expansion mapping being the opposite of a reduction mapping. The algorithm that results is shown in figure 7.8.

```
function non_implicit_expansionC(K,m,c,D)

c[0] = 0
j = 0

for i=0...(K) loop
    product = 1
    loop
        product = product * D[j]
        j = j  + 1
    while product != K[m[i]]
    c[i+1] = j
end for
```

Figure 7.8: The calculation of c for the non-implicit expansion mapping shown from K to D

The non-implicit expansion mapping from K to D

The non-implicit expansion mapping not only requires a change in the calculation of c, it also requires a different algorithm to perform the mapping. The equation given in 3.16 of 3 shows how the mapping from K to D is performed, when the number of dimensions in D is greater than K. The algorithm for a non-implicit expansion mapping is given in figure 7.9.

```
function expansion_non_implicit(w,K,m,c,D,u)

for l=0...(K) loop
    W = w[m[l]]

    for i=c[l]..c[l+1] loop
      u[i] = W mod D[i]
      W = W / D[i]
    end for
end for
```

Figure 7.9: The non-implicit expansion mapping from K to D is performed using this algorithm. This mapping is performed when there are more dimensions in D than there are in K.

The non-implicit expansion mapping with sense

Adding sense indication to the non-implicit expansion mapping is straight forward, and requires changing the order that values from w are assigned to W in figure 7.10. The algorithm in figure 7.10 was derived from equation 4.12 in chapter 4.

```
function expansion_non_implicit_sense(w,K,m,s,c,D,u)

for l=0...(K) loop
    W = w[m[l]]

    if S[l] == '+'
        W = w[m[l]]
    else
        W = K[m[l]] - w[m[l]] -1
    end if

    for i=c[l]..c[l+1] loop
      u[i] = W mod D[i]
      W = W / D[i]
    end for
end for
```

Figure 7.10: The non-implicit expansion mapping from K to D is performed using this algorithm. This mapping is performed when there are more dimensions in D than there are in K.

7.5 Other extensions

Only the sense and extra dimension extensions have been described in the implementation of the k-tile basics. The remaining extensions to the k-tile format described in chapter 4 can be easily implemented with additional algorithms. The offset extension requires the use of an additional algorithm to perform the mapping and will be described in this section. The template extension is part of the k-tile implementation and is discussed in section 7.6, as is the subsection extension which limits the index space being mapped.

7.5.1 The offset extension

The offset extension is best implemented by offsetting the address, before it is given to the implicit or non-implicit mapping. Since the algorithms given for the mapping of the space A onto the space K or the space K onto the space D only map a single address at a time, this is easy to implement. The actual mapping of the entire index space of A onto K, then K onto D is not described by the mapping algorithms, rather the implementation of the k-tile format itself. The algorithm to perform an offset for any mapping is shown in figure 7.11.

```
function offset(w,Z,OZ)

for i=0...(Z) loop
    if OZ[i] != '*' then
        w[i] = (w[i] + OZ[i]) mod Z[i]
    else
        w[i] = 0
    end if
end for
```

Figure 7.11: The offset algorithm shown handles offsets when $O[i]$ is a number and replication when $O[i]$ is a "*". The space Z can represent any space in the k-tile format, with O representing the offset of that space.

In figure 7.11, Z represents any space, with O representing the offset of that space. The "*" operator can only be applied to the k-tile space K, hence its use must be verified by the implementation of the k-tile format.

7.6 k-tile mapping algorithms

Performing a k-tile mapping requires the use of all of the previously defined algorithms to perform an implicit mapping from the data space A onto the k-tile space K, and a non-implicit mapping from the k-tile space K onto the device space D. The implicit mapping from the space A to the space K can potentially be an expansion or reduction mapping, so detection of the correct mapping to use must be considered, as must calculation of the correct value for the c to perform the mapping. The same must be performed for the non-implicit mapping from K to D. Performing this is simple, and the correct mapping to use can be found by comparing the number of dimensions in A, K and D. Care needs to be taken however, if K or D are using the extra dimensions extension, to ensure that the extra dimension is not used in this comparison.

7.6.1 The basic k-tile implementation

If the correct implicit and non-implicit c calculation can be represented by a single function, and the correct implicit and non-implicit mapping can also be represented by a single function, then the algorithm to perform a k-tile mapping is represented in figure 7.12.

```
function ktile(w,A,K,m,s,D,u)

implicitC(A,c1,K)
non_implicitC(K,m,c2,D)

implicit_mapping(w,A,c1,K,g)
non_implicit_mapping(g,K,m,s,c2,D,u)
```

Figure 7.12: A combination of the above algorithms to perform a k-tile. No extensions are used in this implementation

The functions implicitC and non_implicitC choose the correct exapansion or reduction calculation of the c vector for the implicit and non implicit mappings. The functions implicit_mapping and non_implicit_mapping choose the correct expansion or reduction mapping. The vector w in the algorithm represents a single address in A, and u represents the final address in D. The vector g is a temporary address that is mapped from the address u and is mapped onto the address w. Also represented in figure 7.12 is the sense extension, represented by the lower case s in the non_implicit_mapping function.

Since only one address is mapped by this k-tile, there must exist a loop which executes this algorithm a number of times for each address. This can be difficult to code, since the number of the dimensions in space A is not known when building the implementation, and each dimension will require a loop. In order to account for the large range of data spaces that are possible in a k-tile mapping, S and T are used. The source space S and target space T are not specified by the user but by the implementation. Typically S can be any shape that can be handled by the implementation. For instance, S could be a single dimensional array when mapping from a file, where A would represent a 2 dimensional image in the file. This means that only a handful of possible data spaces need to be considered by the implementation, while allowing for flexibility in the definition of A.

When using S, T is also used by the implementation which is mapped onto from the device space D. In the same way that S only needs to represent a small set of the possible data space shapes, T only needs to represent the device spaces supported by the implementation. This allows the device space D to be any space required by the user, since the final mapping will be written to T which is specified by the implementation.

The use of the algorithm given in figure 7.12 to map the entire index space of S onto T is given by the algorithm in figure 7.13.

```
function mapping(data,S,A,K,m,s
                 ,D,T,device)
implicitC(S,cs,A)
implicitC(D,ct,T)

for w=index(S)
    implicit_mapping(w,S,cs,A,g)
    ktile(g,A,K,m,s,D,h)
    implicit_mapping(h,D,ct,T,u)

    device[u] = data[w]
end for
```

Figure 7.13: Demonstrated here is a mapping from S to T, which represent the size of the data and device space respectively. The device and data arrays represent the actual data being mapped.

In figure 7.13, the for loop executes a loop over the entire index space of S. Each address in S is then put into w. The implicit mapping from S

to A then results in the address in w begin put into g. The **ktile** function represents the algorithm in figure 7.12, and is given a mapping from g to h, using A, K, m, s and D as defined. The mapping of an address in A onto the space D results in the address h. This is now implicitly mapped onto T resulting in the address u.

The address w and u are now used to map the source data onto the target data. The arrays data and device on the second to last line in figure 7.13 represent this source and target data where the address stored in w and u point to a single element in these arrays. This element is copied from the source array to the target array.

7.6.2 Using templates

All templates are represented by the letter T followed by the letter of the space this template is applied to. The algorithm in figure 7.14 represents a k-tile mapping which uses template spaces.

```
Function template_mapping(data,S,A,TA,K,TK,m,s
                          ,D,TD,T,device)
if TA == null then TA = A
if TK == null then TK = K
if TD == null then TD = D

implicitC(S,cs,A)
implicitC(TA,c1,K)
non_implicitC(TK,m,c2,D)
implicitC(TD,ct,T)

for w=index(s)
    implicit_mapping(w,S,cs,A,f)
    implicit_mapping(f,TA,c1,K,g)
    non_implicit_mapping(g,TK,m,s,c2,D,h)
    implicit_mapping(h,TD,ct,T,u)
    device[u] = data[w]
end for
```

Figure 7.14: Shown here is the algorithm that performs a mapping from S to T using the template spaces T_A, T_K and T_D. The data and device arrays represent the actual data being mapped.

This algorithm uses all of the templates specified in the k-tile format. If a template for a space does not exist then the template needs to be made

equal to the shape of the space it refers to. For example, if the template space for A, K and D is not specified, then T_A, T_K and T_D would equal A, K and D respectively. The result is an algorithm that implements the template extensions if they are specified in the k-tile.

7.6.3 Using offsets

The use of offsets is simple and is a matter of applying the algorithm in figure 7.11 to each of the spaces in the k-tile. Template spaces must also be offset, if there exists an offset mapping for the template. Figure 7.15 shows an algorithm that implements both offsets and templates.

7.6.4 Using replication

Using replication in a mapping is difficult, and requires more memory as an extra copy of the data needs to be kept during the mapping. Two loops are required, one around the mapping from A to K and another around K to D.

Since K can have any number of dimensions, mapping from K to D requires mapping from a single dimensional space onto K first, so only a single loop is required to loop through the index space of K.

Figure 7.16 shows the algorithm to perform a mapping from A to D using replication.

7.6.5 Using subsections

The final challenge for an implementation is using the subsections extension. This requires limiting the index space mapped from A to D by using only addresses specified by P. Also, since the flow of data is now from the device space array to the data space array, changes need to be made to how data is copied.

The mapping using subsections is performed by two separate algorithms, one to create the subsection of A from P and the other to perform the reversed mapping. The algorithm in figure 7.17 illustrates how the P operator is manipulated to produce an index space of A to be mapped.

The algorithm in figure 7.17 maps S, the shape of data being extracted from T onto A which is larger than S. Using the subsection variable P, sections of A which are highlighted with a star are looped through, dimensions of A that P has a number for are kept at that address.

The second algorithm for the subsection extension uses the algorithm in figure 7.17 to perform the mapping from S to T. This algorithm is shown in figure 7.18.

```
Function offset_mapping(data,S,A,OA,TA,OTA,K,OK,TK,OTK
                        ,m,s,D,OD,TD,OTD,T,device)
if TA == null then TA = A
if TK == null then TK = K
if TD == null then TD = D

implicitC(S,cs,A)
implicitC(TA,c1,K)
non_implicitC(TK,m,c2,D)
implicitC(TD,ct,T)

for w=index(s)
    implicit_mapping(w,S,cs,A,f)

    if OA != null then offset(f,A,OA)
    if OTA != null then offset(f,TA,TOA)

    implicit_mapping(f,TA,c1,K,g)

    if OK != null then offset(f,K,OK)
    if OTK != null then offset(f,TK,TOK)

    non_implicit_mapping(g,TK,m,s,c2,D,h)

    if OD != null then offset(f,D,OD)
    if OTD != null then offset(f,TD,TOD)

    implicit_mapping(h,TD,ct,T,u)

    device[u] = data[w]
end for
```

Figure 7.15: Illustrated here is the algorithm for performing a mapping from S to T which uses the template spaces T_A, T_K, T_D and the offset mappings. The device and data arrays represent the actual data being mapped.

The initial implicit mapping from S onto A is replaced by the function **mapPtoA**, which expands the index space of S so it fits into A. The information to do this is stored in P, which defines which dimensions in A are looped through, and which ones remain constant. The final copying from the data array to the device array which contains the data being mapped is performed in reverse. This is since the subsections extension reduces the

```
Function replication_mapping(data,S,A,OA,TA,OTA,K,OK,TK,OTK
                             ,m,s,D,OD,TD,OTD,T,device)
if TA == null then TA = A
if TK == null then TK = K
if TD == null then TD = D

implicitC(S,cs,A)
implicitC(TA,c1,K)
non_implicitC(TK,m,c2,D)
implicitC(TD,ct,T)

for w=index(s)
    implicit_mapping(w,S,cs,A,f)

    if OA != null then offset(f,A,OA)
    if OTA != null then offset(f,TA,TOA)

    implicit_mapping(f,TA,c1,K,g)

    cache[g] = data[w]
end for

for w=index(K)
    g = w
    if OK != null then offset(g,K,OK)
    if OTK != null then offset(w,TK,TOK)

    non_implicit_mapping(w,TK,m,s,c2,D,h)

    if OD != null then offset(h,D,OD)
    if OTD != null then offset(h,TD,TOD)

    implicit_mapping(h,TD,ct,T,u)

    device[u] = cache[g]
end for
```

Figure 7.16: Demonstrated here is a mapping from S to T in which performs a replication is used. This requires a cache to temporarily store the data being mapped

```
Function mapPtoA(w,S,P,A,u)

j = 0
for i=0..(P) loop
    if P[i] == * then
        R[j] = a[i]
        j=j+1
    end if
end for

implicitC(S,c,R)
implicit_mapping(w,S,c,R,g)

for i=0..(P) loop
    if P[i] == * then
      u[i] = g[i]
    else
      u[i] = p[i]
    end if
end for
```

Figure 7.17: In order to perform a subsection extension, only part of the index space of A's is mapped. This algorithm changes the index space of S, which is the shape of the array being mapped to be given in terms of A.

index space of A, and performs the mapping on the reduced index space. Each address mapped from A to D results in the data in the device space being mapped onto the data space.

```
Function subsection_mapping(data,S,P,A,OA,TA,OTA,K,OK,TK,OTK
                            ,m,s,D,OD,TD,OTD,T,device)
if TA == null then TA = A
if TK == null then TK = K
if TD == null then TD = D

implicitC(S,cs,A)
implicitC(TA,c1,K)
non_implicitC(TK,m,c2,D)
implicitC(TD,ct,T)

for w=index(s)
    mapPtoA(w,S,P,A,f)

    if OA != null then offset(f,A,OA)
    if OTA != null then offset(f,TA,TOA)

    implicit_mapping(f,TA,c1,K,g)

    if OK != null then offset(f,K,OK)
    if OTK != null then offset(f,TK,TOK)

    non_implicit_mapping(g,TK,m,s,c2,D,h)

    if OD != null then offset(f,D,OD)
    if OTD != null then offset(f,TD,TOD)

    implicit_mapping(h,TD,ct,T,u)

    data[w] = device[u]
end for
```

Figure 7.18: The mapping from S onto A is replaced by mapPtoA. The mapping from the data array to the device array is replaced by a mapping from the device array to the data array.

7.7 Summary

Described in this chapter are the different algorithms that have been developed using the theory from chapters 3 and 4. These algorithms are fundamental to the implementation of the k-tile format. None of the algorithms dis-

cussed here have a limitation on the number of dimensions, or the size of each dimension that can be used in the data or device spaces. This differs from the design of Fletcher [Fletcher, 1993], which required each dimension had to be a power of 2. All of the extensions described by Fletcher [Fletcher, 1993], and the subsection extension added in chapter 4 have been implemented by the algorithms described in this chapter. The concept expansion and reduction mappings discussed in chapter 3 is used to describe algorithms where the mapping from A to D is performed in a single step. This is an improvement over the inverse mappings which prevented the mapping from A to D to be performed in a single step. Since expansion and reduction mappings do no limit the number of dimensions in spaces, the implementation results in a more general version of the k-tile format.

Chapter 8

The k-script language

The k-script language was developed in order to perform k-tile mappings. The user can perform the mappings by writing a simple k-script file, which is then parsed and compiled by a k-script compiler. The k-tile can then be run by an emulator with the code generated by the k-script compiler. This emulator would load in one or more data spaces, perform one or more k-tile mappings to the data spaces loaded, and then save the results into the device spaces on available disk. Alternatively, another emulator may distribute the data space over the available processors in a parallel machine.

This chapter introduces the k-script language as a method of describing and executing k-tile mappings. The language itself has been designed in XML [Bosak and Bray, 1999] using a simple syntax.

By using the k-script language, programs to perform k-tile mappings can be developed quickly for a range of applications. A k-script compiler called /textitkScript is discussed in this chapter which has a built in emulator. That automatically performs the k-tile mapping specified by the k-script.

8.1 Early development of k-script

The development of the k-script language resulted in a number of versions of the language before a scripting language that could be used to specify k-tiles was decided upon. The scripting language for VRML [Pesce, 1995] (Virtual Reality Modeling Language) was explored as a possible way to program a language for k-tiles. The language built, worked by rewriting the user load in data spaces from disk, perform the mapping and then save the resulting mapped device spaces to disk. The definition for this language is given in figure 8.1.

Every file for this version of k-script needed to start with a hash, followed

```
<Script> ::= {<Node>}*
<Node> ::= <DiskRaw> | <Cache> | <Copy> | <Ktile>

<DiskRaw> ::= DiskRaw "{" {<Diskfield>}+ "}"
<Cache> ::= Cache "{"{<Cachefield>}+"}"
<Copy> ::= Copy "{"{<Copyfield>}+"}"
<Ktile> ::= Ktile "{"{<Ktilefield>}+"}"

<Diskfield> ::= <Label> | <Filename> | <Dimension>
<Cachefield> ::= <Label> | <Dimension>
<Copyfield> ::=  <Source> | <Target>
<Ktilefield> ::= <Source> | <Target> | <A> | <K> | <m> | <D>

<Label> ::= label A..Z {A..z}
<Source> ::= source A..Z {A..z}
<Target> ::= target A..Z {A..z}
<Dimension> ::=dimension "["{0,9}+ {","{0..9}+}*"]"
<Filename> ::=filename {A..z}*
<A> ::=a "["{0,9}+ {","{0..9}+}*"]"
<K> ::=k "["{0,9}+ {","{0..9}+}*"]"
<M> ::=m "("{0,9}+ {","{0..9}+}*")"
<D> ::=d "["{0,9}+ {","{0..9}+}*"]"
```

Figure 8.1: The syntax for the VRML like k-script

by kScript v1.0. The language itself was clumsy, as many low level elements of a mapping needed to be stated, in order to perform even a simple mapping. An actual script written in this language would be too long to include here, so an example is given in Appendix B.

It became apparent through the development of this version of k-script (kScript 1.0) that it was necessary to use not only a better k-script language, but a better format. Improving the k-script language resulted in the removal of much of the low level syntax and a better format for the k-script language was to use XML.

8.2 XML

XML stands for eXtensible Markup Language and is an extension of SGML [Jelliffe, 1998]. XML is fast becoming the standard for data interchange on the web, but unlike HTML[White, 1996], the tags in XML tell the program what the data means, rather than how to display it. There are a number of

reasons why XML is an important development in Computer Science, but the main reasons which are applicable to the development of a *k*-script language in XML are:

- XML is a plain text format, thus can be edited easily.
- The markup tags in XML identify information in the document to be processed.
- The XML language is easily processed owing to the use of a regular and consistent notation. This makes it easy to build a program to process XML data, in this case a *k*-script file.

A tag acts like a name for a piece of data, for example <definition><definition/ >. The data stored between the start and finish tags is up to the user.

```
<definition>
    <item>Blah</item>
    <text>
        Hello to everyone in XML land.
    </text>
</definition>
```

Each tag can have attributes, which contain additional information that can be included as part of the tag:

```
<definition parameter="stuff">
    <item>Blah</item>
</defintion>
```

A tag which only contains attributes, or does not contain any data, can be shortened to <definition/ >.

The important thing to note about XML is that it is a *well formed* language. A short summary of the rules that define if a XML document is well formed are:

- All tags have a closing tag.
- All tags are completely nested.

  For example < defintion >< item >< / item >< / defintion > is valid, < definition >< item >< / defintion >< / item > is not.

A complete list of requirements of any XML document is given by the World Wide Web Consortium [Marsh and Orchard]. The actual tags that can be used in a XML document are specified by a DTD (Document Type

Definition). A DTD specifies the valid tags and attributes that an XML document can use. Hence a language built around XML would use a DTD to specify the correct syntax for the language.

It was decided, that *k*-script would use XML as a base its the language. A Java XML parser called JAXP [Armstrong, 2000] would allow the writing of a Java program to parse XML, thus simplifying the building of a compiler.

8.3 The k-script language

The *k*-script language developed specifies where to find the source data for a *k*-tile, where to put the target data for a *k*-tile and the *k*-tile to use. A different tag is used for describing files on disk representing the source and target data, and the *k*-tile mapping. Every tag that makes up a *k*-script file is contained in a tag called kScript. To ensure that only the correct combination of tags are used in the *k*-tile format, a DTD describes the *k*-script language. The actual DTD for the *k*-script language is too long to be included here, and is shown in Appendix A.

The DTD specification contains !ELEMENT tags which specify valid tags in the *k*-script language, and !ATTLIST tags which specify attributes for each tag. Each tag that is nested inside a tag is specified by the !ELEMENT tag. A "*" at the end of a nested tag means that there can be zero or more instances of the tag. A "+" at the end of a nested tag, means that there can be one or more instances of the tag while, a "?" means that there can be zero or one instances of the tag and no option at the end of the nested tag means that the tag must be used.

Using these rules, the parser can determine if an XML file is syntactically correct. Figure 8.2 gives an example of a syntactically correct *k*-script file.

The *k*-script file given in figure 8.2 performs a row order mapping on a small 324×324 pixel image.

The Disk tag nested in the kScript tag specifies the data to be used by the ktile tag. Each Disk tag in a *k*-script file has a name and a shape. The shape represents the amount of data that Disk contains. In order to define the data contains in Disk, the Raw tag is used. The Raw tag is nested inside Disk and contains the name of a file on disk. This file is either read from, or written to, depending on the use of the Disk tag.

The total amount of storage represented by the one or many Raw tags inside the Disk tag must be equal to the amount of data represented by Disk tag. If this is not the case, then the *k*-script compiler will abort with an error.

The two Disk tags in figure 8.2 represent a source and target file on disk

```
<?xml version='1.0' encoding='us-ascii'?>
<!DOCTYPE kScript SYSTEM "kScript.dtd">

<kScript>
    <Disk label="A" size="104976">
        <Raw filename="25.raw" size="104976"/>
    </Disk>

    <Disk label="B" size="104976">
        <Raw filename="roworder.raw" size="104976"/>
    </Disk>

    <Ktile source="A" target="B">
        <A size="324 324"/>
        <K size="324 324"/>
        <m value="1 0"/>
        <D size="324 324"/>
    </Ktile>
</kScript>
```

Figure 8.2: A simple row order mapping performed by a k-script file on the file 25.raw which contains a 324×324 pixel image.

for the k-tile mapping. The specification of which Disk tag represents the source data and which represents the target data is performed using the Ktile tag.

The Ktile tag in figure 8.2 specifies the k-tile that will be applied to the data, and what is used as the data space and device space. This defines if a Disk tag will be the source of a k-tile mapping, or the target of a k-tile mapping.

In figure 8.2, a simple row order mapping of the data is specified, from Disk tag labeled A onto Disk tag labeled B. The result of the k-tile mapping from Disk A to B is shown in figure 8.3

8.4 The k-script compiler

A simple compiler was built to compile the k-script language into code which could be run by a k-tile implementation. Using Java [Patrick, 1999] and the JAXP [Armstrong, 2000] API, a simple compiler was built to perform k-tile mappings using the k-script language. The compiler was called kScript and consisted of a parser which extracted the necessary information from the

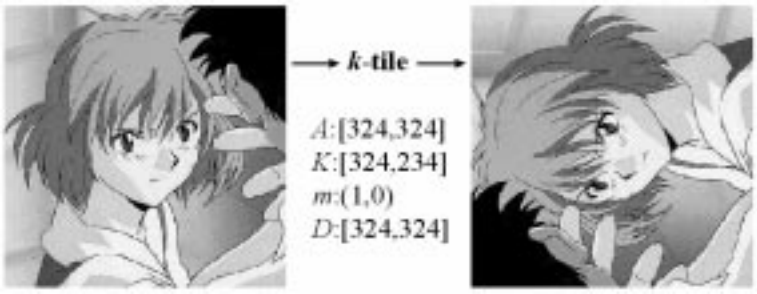


Figure 8.3: The original image on the left has the k-script described applied to it, resulting in a row order mapping of the image shown on the right.

XML file and a k-tile implementation which would execute the mapping. The DTD for the k-script language was used to ensure that the XML file parsed had the correct syntax.

The information extracted from the XML file, was run by a k-script emulator built into kScript. This emulator used the k-tile implementation developed in chapter 7 to perform the mapping.

The compiler and emulator were built in stages, adding functionality as the implementation for the k-tile format was developed. Additional error checking that cannot be handled by the DTD is handled by kScript. The final build for kScript was version 2.8. This version uses an implementation of the k-tile format and the extensions.

Extremely large data sets cannot be handled by version 2.8, as it uses a large amount of memory for replication and caching of data. Version 2.6 of kScript can handle extremely large data spaces yet it lacks replication and caching of the data to be mapped.

8.4.1 Sample k-scripts

Described here are some of the aspects of the k-script language and k-script compiler developed.

Simple k-script

The row order mapping given previously in figure 8.3 users a simple k-script to perform a k-tile mapping. Figure 8.4 is an example of a k-script which performs a crinkle mapping of an image represented by the file 25.raw.

The result of the mapping performed by the k-script compiler, using the k-script code in figure 8.4 is shown in figure 8.5.

```
<?xml version='1.0' encoding='us-ascii'?>
<!DOCTYPE kScript SYSTEM "kScript.dtd">

<kScript>
    <Disk label="A" size="104976">
        <Raw filename="25.raw" size="104976"/>
    </Disk>

    <Disk label="B" size="104976">
        <Raw filename="tiled.raw" size="104976"/>
    </Disk>

    <Ktile source="A" target="B">
        <A size="324 324"/>
        <K size="108 3 108 3"/>
        <m value="0 2 1 3"/>
        <D size="108 108 3 3"/>
    </Ktile>
</kScript>
```

Figure 8.4: A simple crinkle mapping of an image into a 3×3 grid of 108×108 tiles.

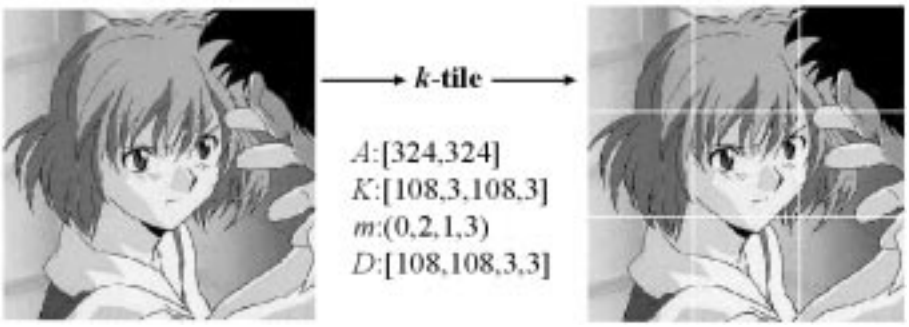


Figure 8.5: The original image on the left has the k-script described in 8.4 applied.

The actual size of each tile and the number of tiles can be easily changed by altering the values for K in figure 8.4 to which shape tile is required. The compiler will however produce an error if the dimensions in K cannot be mapped from A to K or from K to D.

The actual size of A and D in figure 8.4 does not correspond to the size of the data and device space given by the Disk tag. This is since k-blocking is

implemented by kScript. The shape of S is the shape of the source data and the shape of T is the shape of the target data. The space S is then mapped onto the data space A and the device space D is mapped onto the space T. This makes the specification of the k-tile independent of the actual shape of the source data being mapped.

Colour images can be easily represented in k-script by specifying an extra dimension which represents red, green and blue, essential processing a 24 bit image. The k-tile in figure 8.6 shows the specification for a colour image being tiled the same way as in figure 8.5. Figure 8.7 shows the colour mapping that results.

```
<?xml version='1.0' encoding='us-ascii'?>
<!DOCTYPE kScript SYSTEM "kScript.dtd">

<kScript>
    <Disk label="A" size="314928">
        <Raw filename="25.raw" size="314928"/>
    </Disk>

    <Disk label="B" size="314928">
        <Raw filename="tiled.raw" size="314928"/>
    </Disk>

    <Ktile source="A" target="B">
        <A size="3 324 324"/>
        <K size="3 108 3 108 3"/>
        <m value="0 1 3 2 4"/>
        <D size="3 108 108 3 3"/>
    </Ktile>
</kScript>
```

Figure 8.6: A simple tiling of a 24bit image into a 3×3 grid of 108×108 tiles.

k-script with multiple-files

Multiple files for either the source or target data can be handled by kScript. The capability of using multiple files allows for cases where the data space is distributed over a number of files, or the device space is to be mapped onto a number of files. The sample XML code in figure 8.8 is what the target Disk tag in figure 8.4 would be replaced by to use a separate file for each of the tiles created.

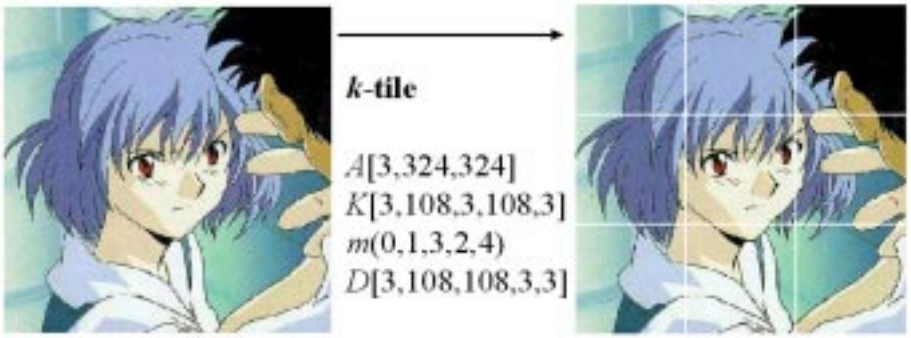


Figure 8.7: The original image on the left has the k-script described in figure 8.6 applied.

```
<Disk label="B" size="104976">
    <Raw filename="1_1_tiled.raw" size="11664"/>
    <Raw filename="2_1_tiled.raw" size="11664"/>
    <Raw filename="3_1_tiled.raw" size="11664"/>

    <Raw filename="1_2_tiled.raw" size="11664"/>
    <Raw filename="2_2_tiled.raw" size="11664"/>
    <Raw filename="3_2_tiled.raw" size="11664"/>

    <Raw filename="1_3_tiled.raw" size="11664"/>
    <Raw filename="2_3_tiled.raw" size="11664"/>
    <Raw filename="3_3_tiled.raw" size="11664"/>
</Disk>
```

Figure 8.8: A sample piece of XML which produces multiple files if used by the k-script in figure 8.4.

Since it is possible that data could be tiled over a large number of files, the k-script compiler conveniently provides a short cut. This short cut involves giving the Raw tag more than one dimension. These additional dimensions result in multiple files being created. Figure 8.9 shows a Disk tag which replaces the one given in figure 8.8.

Figure 8.10 shows the 9 files created by the k-script compiler when executing the XML code in figure 8.4 combined with the XML code in figure 8.9.

```
<Disk label="B" size="104976">
    <Raw filename="tiled.raw" size="11664 3 3"/>
</Disk>
```

Figure 8.9: A sample piece of XML using multiple files that is an improvement over 8.8.

Figure 8.10: 9 images representing each file made from combining the XML code in figure 8.9 and figure 8.4.

Complex k-scripts using extensions

More complex examples of k-scripts use the extensions to the k-tile format. Templates, sense, offsets, replication and subsections can all be used with k-script by using the correct tag for each of the extensions. In the case of extensions where the "*" operator is used such as replication or subsections, the number -1 is used. The sense tag however, uses the "+" and "-" operators.

An example k-script file that uses all of the extensions is shown in figure 8.11, and the image created from this XML file is shown in figure 8.12.

The subsections extension adds some complexity to the implementation of kScript since the data in the data space is no longer copied to the device space. Also since only a subset of the data space A is being mapped, there needs to be a way to control the index space being mapped. Fortunately this is handled by the implementation of the subsection extension.

Figure 8.13 shows an XML file that extracts a 108×108 tile from a

```
<?xml version='1.0' encoding='us-ascii'?>
<!DOCTYPE kScript SYSTEM "kScript.dtd">

<kScript>
    <Disk label="A" size="104976">
        <Raw filename="25.raw" size="104976"/>
    </Disk>

    <Disk label="B" size="480000">
        <Raw filename="tiled.raw" size="480000"/>
    </Disk>

    <Ktile source="A" target="B">
        <A size="324 324"/>
        <Oa value="0 0"/>
        <Ta size="400 400"/>
        <K size="400 400 2"/>
        <Ok value="0 0 -1"/>
        <m value="0 1 2"/>
        <s value="+ + +"/>
        <D size="400 800"/>
        <Od value="38 38"/>
        <Td size="600 800"/>
    </Ktile>
</kScript>
```

Figure 8.11: An XML file that uses many of the extensions in the k-tile format.

324×324 image. Figure 8.14 shows the results of this k-tile on the data.

Extremly large data sets

Extremely large data sets need to be processed by k-script, using as little memory as possible. This makes replication difficult to implement and random access files need to be used in order to work with the data sets. Version 2.6 of kScript was designed to work with extremely large data sets and is the same as version 2.8, without the replication extension. The data set used to test the k-tile implementation was a satellite image of Adelaide. The image was 24bit colour, and was taken at a resolution of 4001×3600, making the size of the data set over 40MB.

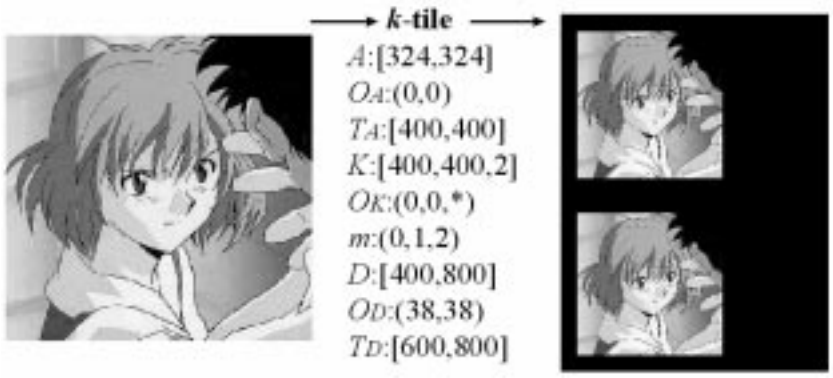


Figure 8.12: The original image on the left has the k-script described in 8.11 applied to it.

```
<?xml version='1.0' encoding='us-ascii'?>
<!DOCTYPE kScript SYSTEM "kScript.dtd">

<kScript>
    <Disk label="A" size="104976">
        <Raw filename="25.raw" size="104976"/>
    </Disk>

    <Disk label="B" size="11664">
        <Raw filename="tiled.raw" size="11664"/>
    </Disk>

    <Ktile source="A" target="B">
        <P value="-1 -1 1 1"/>
        <A size="108 108 3 3"/>
        <K size="108 108 3 3"/>
        <m value="0 2 1 3"/>
        <D size="324 324"/>
    </Ktile>
</kScript>
```

Figure 8.13: An XML file which extracts a 108×108 section from 25.raw.

Using version 2.6 of kScript, a k-tile was used that split up the image into 200×200 tiles. The processing time was approximately 30 minutes on a Cyrix686-166p (Intel Pentium clone). Figure 8.15 shows the k-script file used and figure 8.16 shows a heavily scaled down image of Adelaide, and one of the 378 tiles generated.

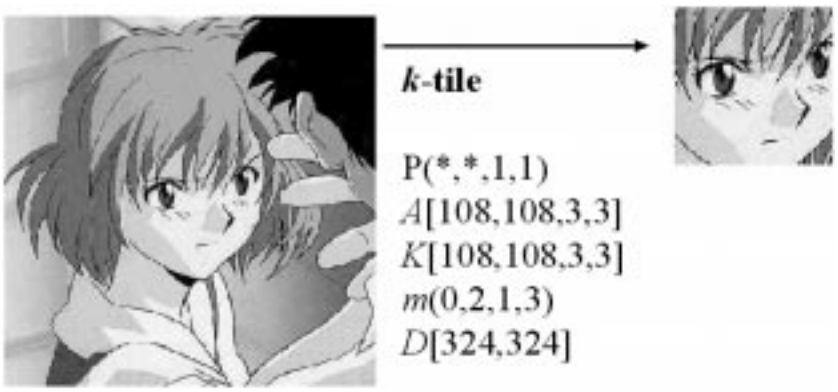


Figure 8.14: The result of the XML file in 8.13 on 25.raw.

```
<?xml version='1.0' encoding='us-ascii'?>
<!DOCTYPE kScript SYSTEM "kScript.dtd">

<kScript>
    <Disk label="A" size="43210800">
        <Raw filename="adelaide.raw" size="43210800"/>
    </Disk>

    <Disk label="B" size="120000 21 18">
        <Raw filename="adelaide.raw" size="120000 21 18"/>
    </Disk>

    <Ktile source="A" target="B">
        <A size="3 4001 3600"/>
        <Ta size="3 4200 3600"/>
        <K size="3 200 21 200 18"/>
        <m value="0 1 3 2 4"/>
        <D size="3 200 200 21 18"/>
    </Ktile>

</kScript>
```

Figure 8.15: An XML file which breaks up the satellite image of Adelaide into 200×200 tiles.

8.4.2 Generic k-tiles

The use of generic k-tiles with k-script was not implemented in the DTD for the k-script language. The XML language does not yet have a standard

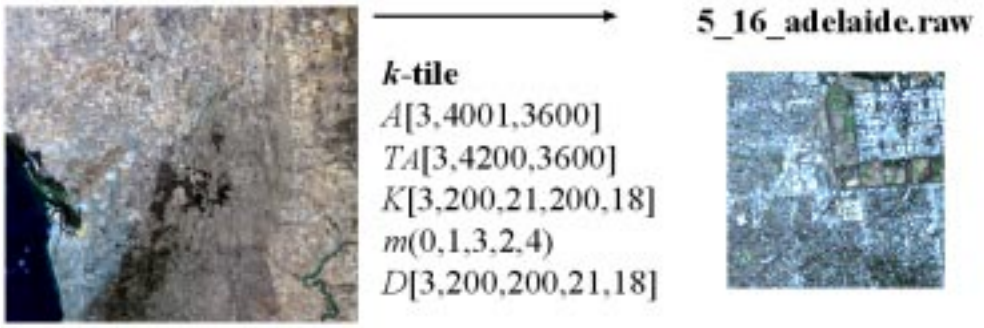


Figure 8.16: Scaled down version of the satellite image of Adelaide, with one of the tiles created by the XML code in 8.15.

for linking a XML file inside another XML file, something that is necessary to perform a generic k-tile. The syntax for generic k-tiles in k-script was developed using the "Import" tag. Figure 8.17 shows the definition of a single generic k-tile in k-script.

```
<Generic name="crinkle" parameters="x y xx yy">
    <A size="x y"/>
    <TA size="a0+(xx-(x%xx)) a1+(yy-(y%yy))"/>
    <K size="xx ta0/xx yy ta1/yy"/>
    <m size="0 2 1 3"/>
    <D size="k0 k1 k2 k3"/>
</Generic>
```

Figure 8.17: Performs a crinkle mapping. The size of the data space can be controlled with the parameters x, y, xx and yy.

This generic k-tile performs a crinkle mapping on any size 2 dimensional data set. Dimension sizes that the size of the tiles can be performed since a template space is used to pad the data. Even prime numbers can be used, since padding results in a dimension size that can be divided by the given tile size is a multiple of the given tile size.

A number of these generic k-tiles could be contained in a single k-script file forming a library of functions. To use these functions, a k-script file needs to import the generic library, and supply the parameters to complete the generic k-tile. In the case of the generic k-script in figure 8.17 the parameters x and y need to be specified. The parameters x and y define the shape of the data space, and parameters xx and yy define the shape of each tile. Figure 8.18 shows a k-script file which imports the library containing the generic k-script

in figure 8.17 called library.xml. The parameters are then provided to the generic k-tile called "crinkle". This performs the k-tile mapping from the source data space "A" to the target data "B".

```
<Import file="library.xml"/>

<RunGeneric name="crinkle"
            parameters="450 450 100 100"
            source="A"
            target="B"/>
```

Figure 8.18: Performs a crinkle mapping using the generic k-script in figure 8.17, with the parameters x=450 y=450 xx=100 yy=100.

8.5 Summary

The k-script language provides a way to specify and perform a k-tile mapping. Based on XML, k-script requires a compiler and emulator to perform a k-tile mapping. Described in this chapter is the syntax for the k-script language and a combined compiler and emulator called kScript. The kScript program uses the implementation developed for the k-tile format and can implement the entire k-tile format, excluding generic k-tiles.

The compiler uses the JAXP API to parse XML documents and the DTD in Appendix A to ensure that the k-script syntax given to the parser is valid. The kScript program works by parsing data out of the XML k-script document, and the building a description of the k-tile. Once the XML file has been parsed, it is possible to perform the k-tile using the information collected. The mapping of the source data to the target data is performed using the k-blocking extensions which separate the k-tile definition from the actual shape of the source and target data.

The k-script language allows for the use of multiple files. This allows for the division of large data sets into a number of smaller data sets that can be more easily processed. For example the Adelaide satellite data set which consists of a 24bit 4001×3600 image was broken up into a number of 200×200 tiles.

Using k-script and the kScript compiler built, the rapid building and testing of k-tile mappings can be performed.

Chapter 9

Summary and future work

Described here is a version of the k-tile format and the k-tile extensions. An implementation in Java of the k-tile format and extensions is described, as is the k-script language, which provides a way to specify and execute k-tile mappings. In addition to this the subsection extension to the k-tile format is described, as well as k-blocking and generic k-tiles.

9.1 The k-tile format

The k-tile format describes a mapping between two multi-dimensional arrays called a data space and a device space. These two spaces are assigned the letters A and D respectively. A third space called the k-tile space specifies a mapping between A and D and is given the letter K. Associated with the k-tile space is the mapping vector m which causes a permutation of the mapping from K to D.

The mapping from A to K in the k-tile format is implicit and the mapping from K to D is non-implicit. Both mappings can be either expansion mapping where the number of dimensions in the device space to is greater than the number of dimensions in the data space, or in a reduction mapping where the number of dimensions in the space being mapped to is smaller.

Extensions to the k-tile format allow a number of additional mappings to be performed such as rotations, padding, replication and subsections. Subsections allows part of a device space to be copied to the data space, which is useful in situations where only a small part of a multi-dimensional array is required. Additional extensions to the k-tile format are k-blocking and generics, where k-blocking solves problems that can arise with certain mappings and generics allow the specification of a k-tile that is not tied to a data space or device space shape.

9.2 The k-tile implementation

The implementation consists of two classes implemented in Java. The first class consists of all of the basic functions necessary to perform a mapping. This includes functions to calculate c and perform reduction and expansion mappings. Additional functions handle offsets and subsections. The second class uses the elements in the first class to perform actual k-tile mappings. This class can be extended to add extra functionality. The algorithms used by this implementation are described in chapter 7.

9.3 The k-script language and compiler

The k-script language provides a way to specify a k-tile mapping and then perform the mapping on a data set. Based on XML, the k-script language is compiled by kScript into code which is run by a k-tile emulator which then performs the mapping. The DTD in Appendix A describes the syntax for the k-script language and is used by the compiler to correctly parse the language.

kScript has a built in emulator which performs a k-tile mapping. In using the k-tile implementation developed, kScript can perform any k-tile mapping, including extensions.

9.4 Future work

In this section, a proposed improvement to the implementation is discussed, as well as an improvement to the kScript compiler. These improvements would provide for a faster implementation, and a version of kScript that consists of a separate emulator and compiler.

9.4.1 An improved implementation

Improvements to the implementation can be made by pre-calculation of some of the elements that make up a single implicit or non-implicit mapping. One of the loops in both the implicit and non-implicit reduction mapping continually calculates the same set of values to perform a mapping. These values could be calculated before a mapping is performed, in the same way as the c vector is. The result of the pre-calculation of thesis values would be to change the cost of performing a reduction mapping from $O(N^3)$ to $O(N^2)$.

9.4.2 An improved *k*-script compiler

Improving the *k*-script compiler involves an approach as used by the gcc [Stallman] compiler. In this case, kScript would compile a program into an executable that could be run by an emulator. Different emulators would exist for different purposes, for instance large data sets would require an emulator that uses little memory. Other emulators would perform many small *k*-tiles as quickly as possible.

Another improvement to kScript is the addition of formats other than "raw" which kScript currently supports. Any format that represents a regular multi-dimensional array can potentially be used as a data space and device space for a *k*-tile mapping.

Finally the generic *k*-tile format needs to be implemented by the kScript compiler. This requires the compiling of the generic *k*-script file into a .o file as is the case for gcc. This file would then be linked into a *k*-script file to perform the mapping.

9.4.3 Other uses for *k*-script

k-script is a language which can be incorporated into other languages such as C and HPFortran to allow these languages to perform *k*-tile mappings of data. Parallel languages would benefit from using the *k*-script notation to distribute information across the nodes in a parallel computer.

In C, *k*-script could be used to map data in one array onto another array. In HPFortran, *k*-script would provide a powerful way to map data across the nodes in a parallel machine. In addition to this, there is no limitation on the data formats that can be processed by *k*-script, providing that the data can be represented as a regular multi-dimensional array. An implementation of another data format could be integrated into the *k*-script compiler or emulator allowing JPEG images for example to be used as a data source or target.

Appendix A

The k-script language DTD

The DTD definition for the k-Script language. Covers all of the extensions in the k-tile format. The file format used is raw, which is just bytes of data on disk. Multiple files can be used to form multi-dimensional data sets.

```
<?xml version='1.0' encoding='us-ascii'?>

<!-- DTD for a revision 3.3 k-script -->
<!-- DTD by Matthew Swan, 25.6.2000, 28.8.2000 18.9.2000-->

<!ELEMENT kScript (Disk*,Ktile*)>

<!-- Raw disk elements -->

<!ELEMENT Disk (Raw+)>
<!ATTLIST Disk
label    CDATA    #REQUIRED
size     CDATA    #REQUIRED
>

<!ELEMENT Raw EMPTY>
<!ATTLIST Raw
filename    CDATA    #REQUIRED
        size        CDATA    #REQUIRED
>

<!ELEMENT Ktile (P?,A,Oa?,Ta?,Ota?,K,Ok?,Tk?,Otk?,m,s?,D,Od?,Td?,Otd?)>
<!ATTLIST Ktile
source    CDATA    #REQUIRED
target    CDATA    #REQUIRED
```

```
>

<!-- Every element that makes up a k-tile and all extensions -->

<!ELEMENT P EMPTY>
<!ATTLIST P
value CDATA #REQUIRED
>

<!ELEMENT A EMPTY>
<!ATTLIST A
size CDATA #REQUIRED
>

<!ELEMENT Oa EMPTY>
<!ATTLIST Oa
value CDATA #REQUIRED
>

<!ELEMENT Ta EMPTY>
<!ATTLIST Ta
size CDATA #REQUIRED
>

<!ELEMENT K EMPTY>
<!ATTLIST K
size CDATA #REQUIRED
>

<!ELEMENT Ok EMPTY>
<!ATTLIST Ok
value CDATA #REQUIRED
>

<!ELEMENT Tk EMPTY>
<!ATTLIST Tk
size CDATA #REQUIRED
>

<!ELEMENT m EMPTY>
<!ATTLIST m
value CDATA #REQUIRED
>
```

```
<!ELEMENT s EMPTY>
<!ATTLIST s
value CDATA #REQUIRED
>

<!ELEMENT D EMPTY>
<!ATTLIST D
size CDATA #REQUIRED
>

<!ELEMENT Od EMPTY>
<!ATTLIST Od
value CDATA #REQUIRED
>

<!ELEMENT Td EMPTY>
<!ATTLIST Td
size CDATA #REQUIRED
>

<!ELEMENT Ota EMPTY>
<!ATTLIST Ota
value CDATA #REQUIRED
>

<!ELEMENT Otk EMPTY>
<!ATTLIST Otk
value CDATA #REQUIRED
>

<!ELEMENT Otd EMPTY>
<!ATTLIST Otd
value CDATA #REQUIRED
>
```

Appendix B

An early version of k-script

A example file for the first version of k-script. Based around VRML, this language was superseded by the XML version of k-script.

```
#kScript v1.0

DiskRaw {
  label DiskA
  filename logo.raw
  dimension [324,324]
}

DiskRaw {
  label DiskD
  filename tiled.raw
  dimension [324,324]
}

Cache {
  label A
  dimension [324,324]
}

Cache {
  label D
  dimension [324,324]
}

Copy {
  source DiskA
```

```
  target A
}

Ktile {
  source A
  target D

  a [324,324]
  k [324,324]
  m (1,0)
  d [324,324]
}

Copy {
  source D
  target DiskD
}
```

Bibliography

[Armstrong, 2000] Armstrong, E. (2000). The Java API for XML parsing (JAXP). *http://java.sun.com/xml/docs/tutorial/index.html*.

[Baylor and Wu, 1999] Baylor, S. J. and Wu, E. C. (1999). Parallel i/o workload characteristics using vesta. *IBM research division*.

[Bosak and Bray, 1999] Bosak, J. and Bray, T. (May 1999). XML and the second-generation web. *Issue 599, Scientific American*.

[Dror G Feitelson and Prost] Dror G Feitelson, P. F. C. and Prost, J.-P. Performance of the vesta parallel file system. *IBM Research*.

[Fletcher, 1993] Fletcher, P. (1993). *Regular Mapping of Multi-Dimensional Data on Parallel Processors*. PhD thesis, Adelaide University.

[Fraser, 1988] Fraser, D. (1988). Mutlidimensional image data formats. *Consultation report, CSIRO Division of Information Technology*.

[Jelliffe, 1998] Jelliffe, R. (1998). *The XML and SGML cook book: recipes for structured information*. Prentice Hall.

[Marsh and Orchard] Marsh, J. and Orchard, D. XML inclusions. *http://www.w3.org/XML/*.

[Patrick, 1999] Patrick, N. and Patrick, H. (1999). *Java 2: The complete reference*. Osborne McGraw-Hill.

[Pesce, 1995] Pesce, M. (1995). *VRML Browsing and Building Cyberspace*. New Riders Publishing.

[Schildt, 1990] Schildt, H. (1987,1990). *C: The Complete Reference, second edition*. Osborne McGraw-Hill.

[Stallman] Stallman, R. M. *Using and porting GNU cc*. Free software foundation.

[Waldby, 2000] Waldby, C. (2000). *The visible human project: informatic bodies and post human medicine*. London: Routledge.

[White, 1996] White, B. (1996). *HTML and the art of authoring for the world wide web*. Academic Publishers.

[Wong] Wong, B. RAID: What does it mean to me? *Bowen Group Inc, Belligham, Wa*.