Design of a Generic, Vectorised, Machine-Vision library

Bing-Chang Lai & Phillip John McKerrow

1. Introduction

Generic programming is a mechanism by which generic algorithms can be implemented. These generic algorithms are obtained by abstracting concrete algorithms (Musser & Stepanov, 1989). Generic programming implements generic algorithms with comparable runtime performance to hand-coded programs (Geraud *et al.*, 2000). The most widely known and used library based on generic programming is the Standard Template Library (STL), which is now part of the C++ standard. Other libraries based on generic programming are available. For image processing, one currently available generic library is Vision with Generic Algorithms (VIGRA) (Köethe, 2001; Köethe, 1999; Köethe, 2000*c*; Köethe, 1998; Köethe, 2000*a*; Köethe, 2000*b*). VIGRA does not use the vector processor.

A vector processing unit (VPU) applies instructions to vectors. A vector is an array of scalars. Desktop VPUs usually have fixed sized vectors, and all vectors are of the same overall size. Since the number of bits in a vector remains constant, the number of scalars in a vector varies across types. Examples of desktop vector technologies include MMX, 3DNow!, SSE, and AltiVec. VPUs are suitable for applications where the same instructions are applied to large amounts of data – Single Instruction Multiple Data (SIMD) problems. Examples of applications suitable for VPUs include video, image and sound processing.

Despite the VPU being ideally suited for image processing applications, and generic libraries having excellent runtime performance and being flexible, there are currently no generic, vectorised libraries for image processing. Unfortunately, adding VPU support to existing generic, image processing libraries, namely VIGRA, is non-trivial, requiring architectural changes.

This paper discusses the problems that occur when combining vector processing with generic programming, and proposes a solution. The problems associated with vectorisation, and the reasons why existing generic libraries cannot be vectorised directly are covered first. This is followed by detailed descriptions of the more important facets of the proposed solution. The performance of the solution is then compared to hand-coded programs.

In the interest of clarification, this paper will use the following terminology.

VVIS: An abbreviation for Vectorised Vision. VVIS is the generic, vectorised, machine-vision library discussed in this paper.

VVM: An abbreviation for Virtual Vector Machine. VVM is the abstract vector processor

Source: Cutting Edge Robotics, ISBN 3-86611-038-3, pp. 784, ARS/plV, Germany, July 2005 Edited by: Kordic, V.; Lazinica, A. & Merdan, M.

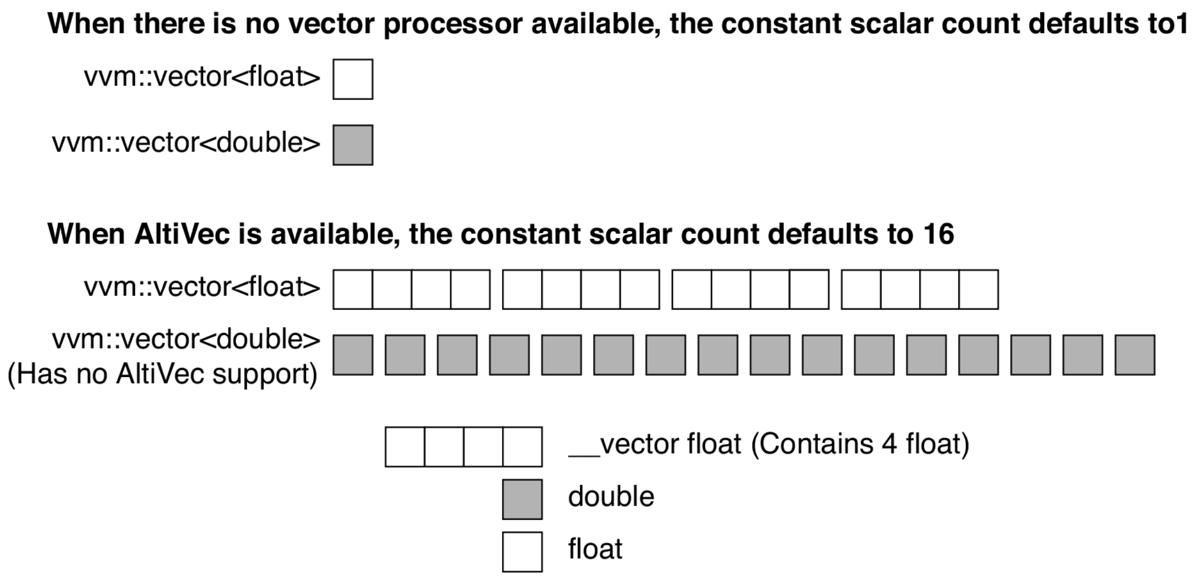

When there is no vector processor available, the constant scalar count defaults to1
vvm::vector<float>
vvm::vector<double>
When AltiVec is available, the constant scalar count defaults to 16
vvm::vector<float>
vvm::vector<double>
(Has no AltiVec support)
__vector float (Contains 4 float)
double
float

Figure 1. Layout of vvm::vector<float> and vvm::vector<double>

(see Section 3), which is used by VVIS.

vp::vector: This refers to a real VPU's native vector type. A vp::vector contains a number of scalars. An example of vp::vectors in AltiVec is __vector unsigned char.

element: In scalar mode, element refers to scalars. When a VPU is available, element refers to vp::vectors.

vvm::vector<T>: This refers to VVM's vector type which contains scalars of type T. A vvm::vector contains a number of elements. These elements are either vp::vectors (which contains a number of scalars) or scalars. See Figure 1 for a graphical overview of the relationship between vvm::vectors, vp::vectors and scalars.

vector: This is used to refer to all vectors, which includes both vp::vectors and vvm::vectors.

2. Problems with Vectorising Existing Generic Libraries

Existing generic libraries, like STL and VIGRA, are difficult to vectorise because the way pixels are arranged in memory is hidden from algorithms by iterators. While such decoupling increases flexibility, hiding how pixels are arranged from algorithms makes it difficult for algorithms to decide whether processing the pixels using the VPU will be beneficial to runtime performance. A vector program that operates on scattered data can be slower than a scalar program, because the scattered data has to be massaged into a form suitable for VPU consumption.

For efficient vectorisation, libraries need to consider:

1. How will algorithms load and write vectors efficiently? How will algorithms handle situations where it is impossible to load and write vectors efficiently?

   Contiguous aligned data are the easiest for the VPU to load efficiently, followed by contiguous unaligned data. Regardless of alignment, contiguous data can be prefetched effectively, which leads to faster loads and stores. When obtaining vectors efficiently is impossible, the easiest and safest solution is to process the data using the scalar processor.

2. How are unaligned data handled? In addition how will two images that have different alignments be handled?

   Since data are unlikely to be aligned in all circumstances, the library needs to consider how unaligned data should be handled. Even when an image is aligned, a region of interest within the image is likely to be unaligned. Apart from unaligned data, an operation that involves more than one input or output needs to consider situations where the alignments of the input and output are different.

3. How are edges handled?

   Edges can be processed using either the scalar processor or the VPU. Processing edges using the VPU is more difficult. When using the VPU, programs can trigger exceptions by accessing memory that does not belong to the current process if the edge is loaded directly. Moreover, using the VPU only produces correct output when the current operation is not affected by the CPU performing more work than necessary. In Section 4.3, we show that quantitative operations cannot have their edges processed using the VPU.

   The scalar processor can always be used to process edges. In addition, because scalar instructions can be executed at the same time as vector instructions, using the scalar processor to handle edges might actually be faster than using the VPU (Apple Computer Inc., 2002).

4. Who handles prefetching?

   Prefetching moves data from memory to the caches before they are used. (Lai *et al.*, 2002) shows that prefetching increased the performance of a vector program significantly on a PowerPC G4. Not all VPUs handle prefetching in the same manner. For example, while the PowerPC G5s also support AltiVec, they require less manual prefetching control because they have automatic prefetching and a larger bus. In fact, some prefetching instructions can be detrimental to speed on the G5, because they cause execution serialisation on the G5 (Apple Computer Inc., 2003). On the G4, such instructions do not cause execution serialisation, and can be used effective within a loop.

3. The Abstract Vector Processor

Generic libraries provide generic functors that can be used with any type that supports the functions used by the functor. For this to be feasible, all types must provide the same functionality through the same interface. Unfortunately, VPU instructions are typically non-uniform across types. An instruction might be available only for some types, and some operations require different instructions for different types. To solve this problem, our generic, vectorised library uses an abstract VPU called Virtual Vector Machine (VVM).

VVM is a templated abstract VPU designed to represent desktop VPUs, such as AltiVec, MMX and 3DNow! . It is a virtual VPU that represents a set of real VPUs with an idealised instruction set and common constraints (Lai *et al.*, 2005). Real VPUs can be abstracted into a virtual VPU when those real VPUs share constraints and have common functionality. For example, AltiVec, MMX and 3DNow! share constraints such as fixed vp::vector sizes and faster access to aligned memory addresses. They all provide operations such as saturated addition and subtraction.

The abstract VPU's idealised instruction set makes the abstract VPU easier to program and portable. It removes the need for programmers to consider instruction availability, and allows a programmer to express his logic using ideal VPU constructs, free from any real VPU's inadequacies. The abstract VPU's common constraints determine which real VPUs it can represent efficiently. Constraints ensure that abstract VPU programs are easier to convert to efficient real VPU programs.

VVM has three constraints common to desktop VPUs: short vvm::vectors, fixed vvm::vector sizes and fast access to aligned memory addresses only. Unlike desktop VPUs, all vvm::vectors have the same scalar count regardless of type. Constant scalar count is important for template programming, which is required for the creation of a generic, vectorised library. Other features that support the creation of a generic, vectorised library include traits, templated vvm::vectors, and consistent functions for both scalar and vvm::vector operations. Due to the high cost of converting types in vector programs, VVM does not provide automatic type conversion; it only supports explicit type conversions.

Constant scalar count: Unlike real VPUs, such as AltiVec, MMX and 3DNow! , which have vp::vectors that are all the same size, and therefore have different scalar counts for different vp::vector types (Motorola Inc., 1999; Mittal *et al.*, 1997; Weiser, 1996), all vvm::vectors consist of the same number of scalars. While the value of this fixed scalar count is not specified by VVM to allow more VPUs to be represented, it is expected to be a small number. A sensible value for this fixed scalar count is the largest number of scalars in a vp::vector. For example, sensible values for scalar counts are 16 for AltiVec, 8 for MMX and 1 for the scalar processor. Constant scalar count is important for template programming, and simplifies type conversions.

Fast access to aligned memory addresses only: VVM can only access aligned memory address quickly. Like SSE and SSE2, VVM also provides slower access to unaligned addresses. In AltiVec, unaligned memory can be accessed by loading aligned memory addresses and performing some transformations (Lai & McKerrow, 2001; Ollmann, 2001; Apple Computer Inc., 2004).

Traits: Traits provide information about types at compile time. Traits are important when automating the generation of more complicated generic algorithms (Köethe, 1999). Trait information can be used in the implementation of VVM itself. They are important for deriving the appropriate boolean type. Unlike scalar programs, vector programs have more than one boolean type. For example, AltiVec has boolean vp::vector types of differing sizes (Motorola Inc., 1999). Promotion traits are important for writing templated code where promotion is necessary, such as in convolutions.

Templated vvm::vectors: Vvm::vectors are templated to support the easy creation of templated vector programs. Templated vector programs are required for a generic, vectorised library implementation in C++.

Consistent functions where applicable: VVM has consistent functions for both scalar and vvm::vector operations where applicable. Consistent functions make VVM easier to use, because programmers can apply their knowledge of scalar operations directly to vvm::vectors. Because VVM has consistent functions, it is often possible to write templated code that performs correctly when instantiated with either a scalar or a vvm::vector. For example, many Standard C++ functors, such as std::plus and std::minus, can be instantiated with either scalars or vvm::vectors.

Comparison operators in VVM however do not return a single boolean, because each vector comparison returns a vector of booleans. In addition, unlike scalar code, true is converted to one's complement of 0 (a value with all bits set to 1, or ~0 in C++) and not 1. VVM maps true to ~0 because in vector programs, the results of comparisons are used as a mask. AltiVec comparison functions also return ~0 for true (Motorola Inc., 1999). Because VVM comparisons return a vvm::vector of booleans and true is converted to ~0, template functions that use comparison operators cannot be instantiated for both scalars and vvm::vectors.

Explicit type conversions only: Type conversions are discouraged in vector programs because type conversions have a pronounced effect on a vector program's performance. Because real vp::vectors typically have different scalar counts, type conversions can change the maximum theoretical speedup. For example, in AltiVec, a __vector unsigned char has 16 scalars, and therefore has a theoretical 16-fold maximum speedup over unsigned char. A __vector unsigned int on the other hand only has 4 scalars and therefore AltiVec has only a 4-fold theoretical maximum speedup. Converting a vvm::vector's type can lead to changes in the theoretical maximum speedup.

The overheads of the VVM implementation used in this paper when compiled with Apple GCC 3.1 20031003 with the -Os switch, were at worst 3.6% slower than hand-coded programs in scalar mode and 0.9% slower in AltiVec mode when operating on char vvm::vectors (Lai *et al.*, 2005; Lai, 2004). For other vvm::vectors in AltiVec mode, the VVM implementation was at worst 23.0% slower than a hand-coded AltiVec program.

4. Categorising Operations Based on their Input-to-Output Correlation

Vector programs are usually difficult to implement efficiently due to memory bottlenecks and non-uniform instructions across types. To minimise this problem, a categorisation scheme based on input-to-output correlation was introduced to reduce the number of algorithms required. Implementing several operations with a common algorithm allows efficiency problems to be solved once.

Operations are categorised by characteristics of the input they require, the output they produce from the input, and how the output is produced from the input. Useful characteristics for categorising image processing operations are the number of input elements required to produce the output, the number of output elements produced from the input, the number of input and output sets, and the types of the input and output elements. The term element is used instead of pixels because separating the type reduces the number of distinct algorithms. For example, since rotation is a one input element (of type coordinate) to one output element (of type coordinate) operation, it can use the same algorithm as threshold, which is a one input element (pixel) to one output element (pixel) operation. The term set is used to refer to a collection of elements.

Number of input and output elements: The number of input and output elements refers to the number of input elements per input set, and number of output elements per output set produced respectively. For example, a threshold operation requires one input element (pixel) per input set (image) to produce one output element (pixel) per output set (image). Arithmetic operations like addition and subtraction also require one input element per input set to produce one output element per output set, but require two input sets.

Number of input and output sets: The number of input and output sets refer to the number of input sets required and the number of output sets produced respectively. In image processing, input sets are typically images, while output sets are images, histograms or statistics. A threshold operation has one input set (an image) and one output set (an image). Binary arithmetic operations, such as addition and subtraction, have two input sets (both images) and one output set (an image). A histogram operation has one input set (an image) and one output set (a histogram).

Input and output element types: Input and output types refer to the type of the input and output elements. All elements have a single type. Most image processing operations have input and output types of pixels. Other possible output types in image processing are histograms, and statistics. In geometric transformations, input and output types are coordinates.

Number of Elements		Number of Sets		Type of Elements		
Input	Output	Input	Output	Input	Output	Examples
1	1	1	1	Pixels	Pixels	Lookup transformations. Colour conversions. Eg. threshold, equalise, reverse and invert.
1	1	2	1	Pixels	Pixels	Arithmetic and logical operations. Eg. addition and subtraction.
Rectangular (eg. 3x3 pixels windows)	1	1	1	Pixels	Pixels	Spatial filters. Eg. convolution filters, edge extraction and edge thickness. Also includes some morphological analysis. Eg. erosion and dilation.
1	0 or more	1	1	Pixels	Number	Quantitative analysis. Eg. perimeter and area.
1	1	1	1	Coordinates	Coordinates	Geometric operations.
M	N	1	1	Coordinates	Coordinates	Scale operations.

Table 1. Some image processing algorithms categorised using input-to-output correlation

Table 1 illustrates how image processing operations can be categorised using the six criteria discussed. M and N refer to the total number of input and output elements respectively. Rectangular refers to a rectangle of input, e.g. a 3x3 pixel window. Spatial filters, like Sobel filters, typically produce a single pixel from a square of pixels centred around a pixel; so they have rectangular input elements. A rectangle was used instead of a square to make the group more general. Since a single pixel is also a rectangle, operations accepting one input element per input set also fall under rectangular.

Whether all input elements are always processed is not part of the criteria, because all input elements are always processed for the operations considered. All the groups shown in Table 1 assume all input elements are processed. An example of an operation that does not always process all input elements is “find first”.

When selecting the correct number of input or output elements per input set, it is useful to visualise how the operation would be implemented using generic programming. For example, threshold operations use one input pixel to decide one output pixel for each iteration. Apart from considering how an operation is implemented using generic programming, it is also helpful to separate the input requirements from the output requirements. For example, histogram operations have one input element per input set, and produce zero or more output elements per output set. They are categorised as one input element per input set because they use each input pixel independently. They produce zero or more elements per output set because the number of elements in the output set is independent of the number of pixels in the image. Histogram operations process each input element one at a time, producing zero output elements until the last pixel is processed, after which many output elements are produced.

4.1 Applying the Categorisation Scheme to Generic Programming

Different criteria are handled by different concepts in a generic library. For example, in the Standard Template Library (STL), the number of input elements, number of output elements, number of input sets and number of output sets are handled by the algorithm. The input element type and the output element type are handled by the iterator. In STL, rotations can be expressed as a std::transform call that takes one input set and one output set, and operates on iterators that return coordinates. In VIGRA, the number of input elements, number of output elements, number of input sets and number of output sets are all handled by the algorithm. However, the input element type and output element type can be handled either by the iterator or the accessor.

Some criteria have more impact on the implementation of the algorithms than others. The most important criteria for categorising operations to reduce the number of algorithms required is the number of input elements because the other characteristics can either be handled by other concepts or have little impact on the algorithm’s implementation. For example, while an algorithm that accepts two input sets instead of one uses more iterators, and has more arguments to the algorithm and to the functors, the structure of the algorithm remains unchanged. The number of output sets, and number of output elements per output set can be handled by either the functor or the accessor. Examples of functors that write output are vigra::FindAverage, vigra::FindBoundingRectangle and vigra::FindMinMax. Input and output element types are handled by iterators and accessors and thus have no impact on the implementation of algorithms.

When applied to generic programming, each defined category will have one main algorithm, with variations for each combination of input and output sets. Once an operation satisfies the criteria for a category, the category’s algorithm can be used, even though it might not be the most efficient. For example, “find first” can be implemented using the same algorithm as histogram’s, since they have the same six criteria. However, “find first” would be faster if it gave up searching after finding the answer. This example shows that more criteria can be added to further narrow the groups. More criteria were not used in this paper, because the objective, a generic, vectorised, machine-vision library, did not require any other criteria.

Under the input-to-output correlation categorisation proposed, an operation can belong to many categories simultaneously. For example, a convolution algorithm can also be used to perform thresholding because one input element is also rectangular input (one pixel is also a 1x1 rectangle). Since all categories are actually subsets of a set whose input element to output element correlation is M to N, only one main algorithm is needed. Such an algorithm would delegate all processing to the functor; it passes all the input to the functor, and waits for all the output. Generally, when an operation cannot be placed into a group smaller than M to N, it is prudent to explicitly implement an algorithm for that operation.

4.2 Inferences

Some of the characteristics discussed have implications for how an operation can be implemented. These characteristics are discussed below.

One input element per input set: This implies that only a 1-D iteration is needed. Hence the algorithm can be written to process images of any shape.

One output element: Having one output element implies that the algorithm can perform more work than necessary, as long as it ensures that the extra results are discarded. Since extra results can be computed, edges can be handled using the VPU (see (Apple Computer Inc., 2002) for more information about edges).

One input element per input set produces one output element per output set: This characteristic suggests that elements can be computed out of turn efficiently. The output order still has to match the input order.

Rectangular input elements per input set: This characteristic indicates that a spatial iterator is required. For vector programs, it is faster to compute the data from left to right, top to bottom, since data loaded from the last iteration can be used in the next.

Zero or more output elements: Zero or more output elements suggest that the algorithm cannot handle the output because the output is unknown. Functors cannot simply return a list of output elements because the answers might be unknown until all elements are processed, and functors generally do not know which element is the last element. While it is possible for the algorithm to inform the functor when the last element is reached, it is easier to let the functor handle the output. In addition, zero or more output elements suggests that performing more work can lead to the wrong output, since the algorithm cannot discard unwanted output. The algorithm cannot discard unwanted output because it does not handle the output.

4.3 Categories for a Generic, Vectorised, Machine-Vision Library

Because the main reason for using this categorisation scheme was to reduce the number of algorithms required while retaining efficiency, the requirements of the algorithms form the basis for defining categories for a generic, vectorised, machine-vision library. From Section 4.1, the most important criteria for categorising operations to reduce the number of algorithms required is the number of input elements. Using only this criterion results in two categories, indicating that only two main algorithms are required to handle the image processing operations considered in Table 1. However, because output characteristics were not used, the functor would be always responsible for the output. Because functors

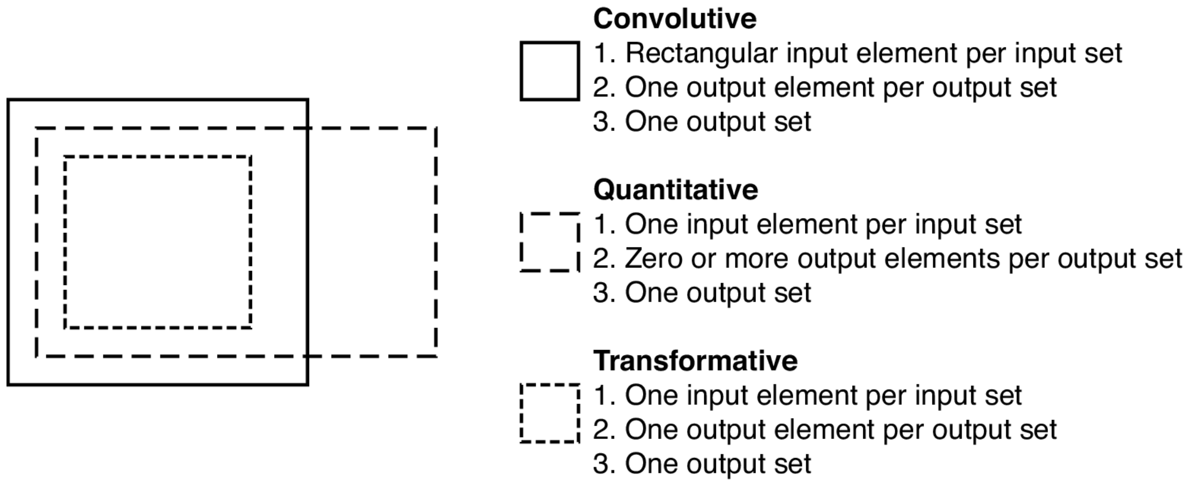

Convolutive
1. Rectangular input element per input set
2. One output element per output set
3. One output set
Quantitative
1. One input element per input set
2. Zero or more output elements per output set
3. One output set
Transformative
1. One input element per input set
2. One output element per output set
3. One output set

Figure 2. Categorisation for a generic, vectorised, machine-vision library

that handle output are more difficult to implement, are not consistent with existing generic libraries, and most image processing operations produce a single image as their output, consequently, the number of output sets was also used to derive the categories for the generic, vectorised, machine-vision library. Since functors are the most common concept created by users, making functors more difficult to implement would make the library more difficult to use.

Number of input elements per input set, number of output elements per output set and number of output sets were used to divide the image processing operations considered in Table 1 into three categories: quantitative, transformative and convolutive. Figure 2 shows how the three categories are related to each other.

Quantitative operations: Quantitative operations have one input element per input set, zero or more output elements per output set and a single output set. Because they have zero or more output elements, the output is handled by the functor. An example of a quantitative operation is the histogram.

Transformative operations: Transformative operations accept one input element per input set. In image processing, transformative algorithms require one or two input sets, and produce one output image set. Since transformative operations involve either one or two input sets, two algorithms are needed. Since they have one output element per output set, edges can be processed using the VPU. In addition, because they have one input element per input set, they are easy to parallelise and can be computed out of order without problems. Examples of transformative operations are thresholds, additions and subtractions.

Convolutive operations: Convolutive operations accept a rectangle of elements per input set, from which they produce one output element for one output set. Convolutive algorithms are named after the operations that mostly fall under it – convolutions. Because they have one output element per input set, edges can be processed using the VPU. Because they have rectangular input, convolutive algorithms require spatial iterators. While convolutions can be computed out of order, there is little reason for vector programs to do this, because most of the input already loaded for the current output is required for computing the next output. Examples of convolutive operations are linear and non-linear filters like Sobel filters, and Gaussian filters.

5. Storages

As mentioned previously, one of the main reasons why existing generic libraries cannot be vectorised easily is because iterators provide no information on the arrangement of pixels in memory to algorithms. To address this problem, the storage concept is introduced.

Storages specify constraints on how the data contained within them are arranged. Instead of passing iterators to algorithms, storages are passed. Storages allow algorithms to make informed decisions regarding the use of the VPU. Since some algorithms operate on specific geometric shapes, it is important to associate a shape with a storage. Because iterators can move and thereby break the constraints guaranteed by a storage, passing iterators to the algorithm makes it difficult to decide whether or not to use the VPU at compile time.

Three storage types are specified: contiguous, unknown and illife. Contiguous and unknown storages are one-dimensional storages, while illife storages are n-dimensional storages. Contiguous storages are processed using the VPU, while unknown storages are processed using the scalar processor. Contiguous storages are actually a subset of unknown storages, and as a result, contiguous storages support the unknown storage's interface. This is important when processing multiple storages together. When a single storage is unknown, all the storages are processed as unknown storages.

5.1 Contiguous Storages

Contiguous storages are one-dimensional storages where components of pixels are adjacent to each other in memory. Originally, components were required to be adjacent from the beginning of the storage to the end. However, this restriction was later eased to allow for chunky vector storages, which are storages where components are interleaved one vector at a time. Contiguous storages require components of pixels to be adjacent for a single vvm::vector. Table 2 shows the interface that all contiguous storages must provide.

Two kinds of contiguous storages are specified: contiguous aligned storages and contiguous unaligned storages. Contiguous unaligned storages must be convertible to contiguous aligned storages. The corresponding contiguous aligned storage type is specified by the contiguous_aligned_storage template metafunction, The inverse conversion is not necessary, because a contiguous aligned storage fulfills the criteria for a contiguous unaligned storage already.

The iterator returned by begin(), points to the first element, and must be aligned for contiguous aligned storages, but may be unaligned for contiguous unaligned storages. end() returns an iterator that points to the first past-the-end scalar, and can be unaligned for both types of contiguous storages. While end() can return an unaligned iterator, contiguous storages must ensure that performing a vvm::vector load from the last aligned position is valid. Contiguous storages are expected to pad the number of bytes allocated, so that the allocation ends on an aligned position. Figure 3, which assumes that there are four scalars in a vvm::vector, illustrates this requirement. Three extra scalars were allocated at the end to allow the last aligned position to be loaded as a vvm::vector.

5.1.1 Iterator Requirements

An iterator for a contiguous storage must be able to traverse forwards and backwards by taking scalar, vvm::vector, or pixel steps. Advancing an iterator by scalar or vvm::vector steps changes the position represented by the iterator by a scalar or vvm::vector respectively. Scalar steps are only valid when the step does not leave the current vvm::vector. Pixel steps are the same as scalar steps, except that they can cross vvm::vector boundaries. Pixel steps are used by convolutive algorithms to adjust the end iterator. Figure 4 illustrates the differences between scalar, vvm::vector and pixel steps when iterating through a chunky vector storage.

Both scalar and pixel steps are required despite both advancing by scalars because pixel steps may require more processing, and crossing vvm::vector boundaries is only required occasionally. For planar storages, scalar and pixel steps have the same implementations. However, for chunky vector storages, a scalar step is easier to implement than a pixel step.

The iterator is expected to treat scalar and vvm::vector components orthogonally. This allows the ! = operator to be applied to each component separately, and removes the need to calculate the location of the right edge. There is no need to calculate the position of the right edge because once the vvm::vector steps are completed, only the right edge remains. While keeping two orthogonal components can increase the cost of creating iterators, since both scalar and vvm::vector components have to be calculated, it simplifies algorithm implementation and reduces processing requirements during execution.

The required interface for an iterator for a contiguous storage is shown in Table 3. Note that there is no operation to read data from or write data to the current location. The required interface for reading and writing data is determined by what accessors the iterator has to be compatible with. Accessors are discussed later in Section 7.

Expression	Return Type	Notes
X::component_tl	ct::typelist of T1, T2, ...	
X::iterator	iterator type pointing to T	Convertible to X::const_iterator
X::const_iterator	iterator type pointing to const T	
a.begin()	iterator; const_iterator for constant a	
a.end()	iterator; const_iterator for constant a	
contiguous_aligned_storage <X>::type	Contiguous aligned storage	
contiguous_aligned_storage <X>::const_type	const contiguous aligned storage	

Table 2. Contiguous storage's required interface

When designing the iterator for a contiguous storage, the following ideas were considered: for iterators to keep a single location, to advance the iterator by adding constants, and to have different iterator types for each step type. These ideas were discarded for reasons discussed below.

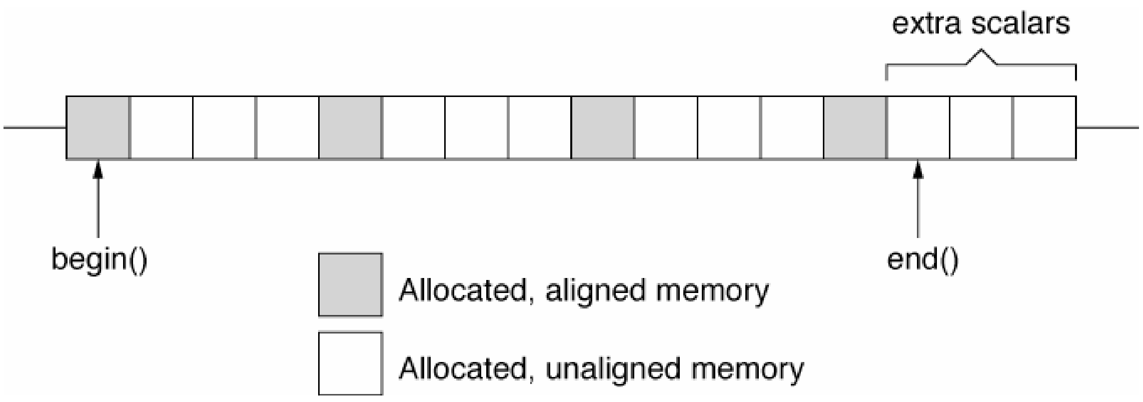


Figure 3. Expected allocation requirements of a contiguous storage

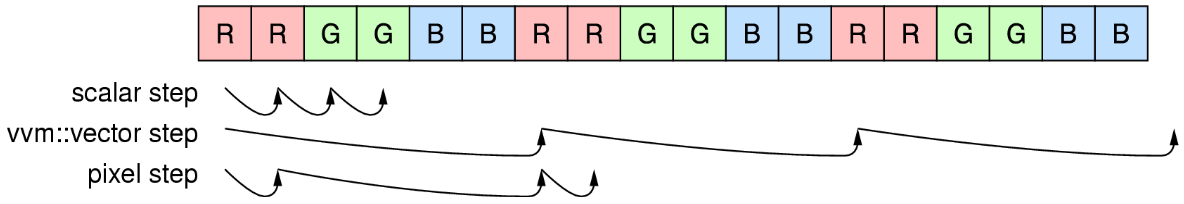


Figure 4. Differences between scalar, vvm::vector and pixel steps when iterating through a chunky vector storage

Keeping a single location in the iterator: The iterator originally kept a single location, and provided methods to move the position by different step types. Keeping a single location forces the algorithm to calculate the position of the right edge, and might increase the execution cost of the != operator, if the operator needs to calculate the real position first.

Advancing the iterator by adding constants: Advancing the iterator by scalar and vvm::vector steps by adding constants was also considered. For example, assuming i is an iterator, i += scalar_step would be a scalar step, and i += vector_step would be a vvm::vector step. For planar storages, scalar_step and vector_step would be equal to one and the VVM scalar count respectively. This method would have allowed iterators to be implemented as pointers, thereby producing code that was probably easier for compilers to optimise.

While this method can iterate through planar storages easily, it cannot iterate through some multi-channel chunky storages with differing channel types. In such cases, vector_step cannot be specified in terms of number of scalars. For example, a multi-channel chunky scalar storage, which is a chunky storage where components are interleaved one scalar at a time, with int RGB channels and a char alpha channel needs to have a scalar_step of 1 int and a vector_step of 3.5 ints. If we use different object types to represent scalar, vvm::vector, and pixel steps, we can overcome this problem. (Weihe 2002) provides some ideas on how such syntax can be implemented correctly. This idea was not investigated further because it seemed likely to introduce performance overheads.

Operation	Result	Notes
difference_type	Type	Returns type of i - j
i = j	iterator&	After operation, i != j is false
i != j	bool	After operation, i == j is false
i.vector != j.vector	bool	
i.scalar != j.scalar	bool	
i - j	difference_type	Returns number of pixels between i and j
++i.vector; i.vector++	iterator&	Move forward by 1 vector
i.vector += j	iterator&	Move forward by j vectors
++i.scalar; i.scalar++	iterator&	Move forward by 1 scalar
i.scalar += j	iterator&	Move forward by j scalars
++i; i++	iterator&	Move forward by 1 pixel
i += j	iterator	Returns an iterator that has moved forward by j pixels
--i.vector; i.vector--	iterator&	Move backward by 1 vvm::vector
i.vector -= j	iterator&	Move backward by j vvm::vectors
--i.scalar; i.scalar--	iterator&	Move backward by 1 scalar
i.scalar -= j	iterator&	Move backward by j scalars
--i; i--	iterator&	Move backward by 1 pixel
i -= j	iterator&	Returns an iterator that has moved backward by j pixels

Table 3. Required interface for an iterator for a contiguous storage

Operation	Result	Semantics
difference_type	Type	Returns type of i - j
i = j	iterator&	After operation, i != j is false
i ! = j	bool	Checks if i and j are pointing to the same location
i - j	difference_type	Returns number of pixels between i and j
++i; i++	iterator&	Move forward by 1 pixel
i += j	iterator&	Move forward by j pixels
--i; i--	iterator&	Move backward by 1 pixel
i -= j	iterator&	Move backward by j pixels

Table 5. Unknown storage's required iterator interface

Using different iterator types for different step types: Using different iterator types for each step type was also considered, because it was thought that this method would allow pointers to be used as iterators with both planar and chunky storages. Unfortunately, this is not possible, because there would have been only two pointer types (scalar and vvm::vector) and three different step types. In addition, the different iterators need to be convertible between one another, because we want to be able to advance by vvm::vector steps when we were using the VPU and switch to scalar steps when we came to the edges. The only way to convert pointers is to use reinterpret_cast; it is not possible to perform automatic conversions between pointer types. reinterpret_cast is not suitable when the iterators are not pointers.

5.2 Unknown Storages

Unknown storages are one-dimensional storages where there are no restrictions on how components of a pixel are arranged. Since there are no restrictions, all one-dimensional storages are unknown storages. The interface of the unknown storages is a subset of the interface of contiguous storages. begin() and end() can both return unaligned iterators. begin() returns an iterator that points to the first element. end() returns an iterator that points to the past-the-end element. Table 4 summarises the interface that unknown storages must implement.

5.2.1 Iterator Requirements

An iterator for an unknown storage is expected to be able to traverse forwards and backwards by taking pixel steps. Scalar and vvm::vector steps are not required, because unknown storages are expected to be processed by the scalar processor. The iterator's interface is shown in Table 4. The interface is the same as the interface of a contiguous storage's iterator, exception there are no .scalar and .vector operations.

Expression	Return Type	Notes
X::component_tl	ct::typelist of T1, T2, ...	
X::iterator	iterator type pointing to T	Convertible to X::const_iterator
X::const_iterator	iterator type pointing to const T	
a.begin()	iterator; const_iterator for constant a	
a.end()	iterator; const_iterator for constant a	

Table 4. Unknown storage's required interface

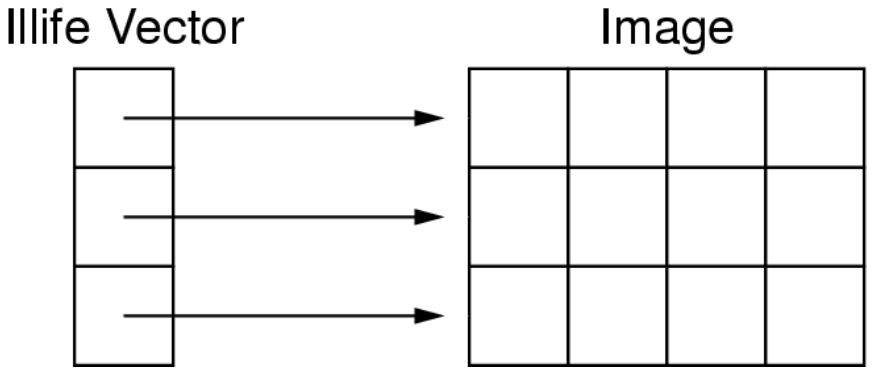


Figure 5. An illife vector

5.3 Illife Storages

Illife storages are storages that contain other storages. Illife storages are named after illife vectors, which are vectors that store row pointers in an image. Figure 5 illustrates an illife vector. Illife storages are expected to be row-major. Pixels adjacent horizontally are adjacent in memory, while pixels adjacent vertically are not. Row-major was chosen because this is the usual major used in the implementation language C++.

Unlike contiguous or unknown storages, the illife storage represents an n-dimensional storage. It is n-dimensional because the storage type of the rows can also be an illife storage. However, some algorithms, like convolutions, require the illife storages to contain only unknown or contiguous storages, that is to be two-dimensional.

The interface of illife storages is shown in Table 6. Since the illife storage's interface is quite similar to STL containers, it should be straightforward to use STL containers to house each row's storage. begin() returns an iterator referring to the first row. end() returns an iterator that refers to the first past-the-end row.

Expression	Return Type	Notes
X::value_type	T	T is assignable
X::iterator	iterator type pointing to T	Convertible to X::const_iterator
X::const_iterator	iterator type pointing to const T	
a.begin()	iterator; const_iterator for constant a	
a.end()	iterator; const_iterator for constant a	

Table 6. Illife storage's required interface

Expression	Return Type	Notes
i = j	iterator&	After operation, i != j is false
i ! = j	bool	
++i; i++	iterator&	Moves down by 1 row
i += j	iterator&	Moves down by j rows
i + j	iterator	Returns an iterator j rows down
--i; i--	iterator&	Moves up by 1 row
i -= j	iterator&	Moves up by j rows
i - j	iterator	Returns an iterator j rows up
*i	value_type	Returns the current row
i[n]	value_type	Returns the row n rows down

Table 7. Illife storages' required iterator interface

The illife storage's interface has functions with the same names as the contiguous storage's interface, which perform different operations. For example, in contiguous and unknown storages, begin() returns an iterator that points to the beginning of image data while in illife, begin() returns an iterator pointing to the first row's storage. Because of this, a storage that supports the illife storage interface cannot support the contiguous or unknown storage interfaces.

In initial designs, the illife storage had an interface that was compatible with unknown and contiguous storages. This allowed the same storage to implement both the interfaces of both the illife storage and a one-dimensional storage. begin and end functions accepted an int, which specifies a row, and returned iterators to image data. However, such an interface makes it more difficult to call the algorithms for one-dimensional storages directly for a row, because algorithms accept storages and not iterators. Since the final design returns iterators to storages instead of to image data, storages can be easily extracted from the iterators.

5.3.1 Iterator Requirements

The iterator's requirements are shown in Table 7. The iterator supports the +, - and [] operators. These operators allow algorithms, such as the convolutive algorithm, to access rows offset from the current row easily.

Expression	Return Type	Notes
A::prefetch_channel_count	int	Returns number of prefetch channels required
A::scalar_type	Type	Scalar Type
A::vector_type	Type	Vvm::vector Type
a.prefetch_read<ch>(i)	void	Prefetch from iterator i for read using channel ch
a.get_scalar(i)	scalar	Returns the scalar at iterator i
a.get_scalar(i, o)	scalar	Returns the scalar at iterator i+o scalar steps
a.get_vector(i)	vector	Returns the vector at iterator i
a.get_vector(i, o)	vector	Returns the vector at iterator i+o vector steps
a.get_vector(a, b, o)	vector	Returns a vector from a and b. The first scalar of the vector is a[o].

For unknown storages, A::prefetch_channel_count, A::vector_type, a.prefetch_read<ch>(i), a.get_vector(i), and a.get_vector(i, o) do not need to be implemented.

Table 8. Read accessor's required interface

6. Regions

Regions are storages that represent regions of interest in other storages. Since regions are storages, regions can be passed to algorithms. In existing generic libraries, iterators mark the region of interest to algorithms. Regions allow for the same region of interest marking in a generic, vectorised library.

Regions are expected to have an admit/release phase like VSIPL (Georgia Tech Research Corporation, 2001; *VSIPL website*, 2001). Like VSIPL, access to a portion of the storage that has been admitted to the region, must be made through that region. A storage can have more than one portion admitted to different regions at one time, as long as the portions do not overlap. This admit/release phase is important in allowing regions to perform pre- and post- processing that will be helpful to runtime performance.

7. Accessors

In STL, data elements are accessed via references to the original data returned by operator* and operator[]. A problem with this approach is that, for multi-channel planar images, there is no efficient way of returning a reference to a pixel. To access a pixel that contains all the channels of a multi-channel planar image, a proxy object is required. This proxy object reduces the efficiency of accessing the data elements (Kühl & Weihe, 1997). Accessors were introduced by (Kühl & Weihe, 1997) to solve this problem by providing an extra level of indirection.

Unlike accessors in VIGRA, or those described in (Kühl & Weihe, 1997), that provide access to a single type, accessors in VVIS provide access to two types – scalars and vvm::vectors. In addition, apart from being responsible for retrieving and writing data, and performing any necessary type conversions, accessors in VVIS are also responsible for prefetching. Read and write accessor requirements for contiguous storage are shown in Tables 8 and 9 respectively.

Expression	Return Type	Notes
A::scalar_type	Type	Scalar Type
A::vector_type	Type	Vvm::vector Type
A::prefetch_channel_count	int	Returns number of prefetch channels required
a.prefetch_write<ch>(i)	void	Prefetch from iterator i for writing using channel ch
a.set_scalar(s, i)	void	Writes scalar s to iterator i
a.set_vector(v, i)	void	Writes vector v to iterator i

For unknown storages, A::prefetch_channel_count, A::vector_type, a.prefetch_write<ch>(i), and a.set_vector(v, i) do not need to be implemented.

Table 9. Write accessor's required interface

Prefetching is handled by accessors because only accessors know exactly what data is being loaded and stored. For example, an accessor that provides access to a single channel only needs to prefetch that single channel instead of all the channels. Prefetching functions are only used for contiguous storages.

Accessors provide functions to get and set both scalars and vvm::vectors at the current location, or at an offset from the current location. There is an extra function for loading a vvm::vector from two vvm::vectors, that can be used to perform unaligned loading.

scalar_type and vector_type refer to the scalar and vvm::vector type used by the accessor. This is useful when the algorithm wishes to keep a copy of the values returned by the accessor. prefetch_channel_count is an integer containing the number of prefetch channels required when prefetching. To prefetch from multiple storages, prefetch_channel_count of the previously prefetching storage should be passed to the next set of prefetch functions, to avoid using the same prefetch channels. The following example illustrates how to prefetch multiple storages.

```
// Assume there are three accessor types, A1, A2, A3
// Assume a1, a2, a3 are of types A1, A2, A3
// Assume there are three iterators i1, i2, i3
// Assume i1 and i2 are for reading, i3 is for writing
```

```
a1.prefetch_read<0>(i1);
a2.prefetch_read<A1::prefetch_channel_count>(i2);
a3.prefetch_write<A1::prefetch_channel_count +
              A2::prefetch_channel_count>(i3);
```

8. Algorithms

Algorithms are responsible for coordinating how the other concepts are used to solve a problem. In Section 4.3, three categories were identified for use with VVIS: quantitative, transformative and convolutive. for_each, transform and convolute are the algorithms for quantitative, transformative and convolutive categories respectively.

Table 10 shows the algorithms that VVIS supports. The transform algorithm has two versions: one for one input set, and one for two input sets. Like VIGRA, algorithms accept an accessor for each input and output set. Without an accessor for each input and output set, it is more difficult to perform operations on images that involve different channels.

Algorithms are expected to process data differently depending on the storage type. Algorithms are expected to use the scalar processor and the VPU to process unknown and contiguous storages respectively. An algorithm in VVIS does not have to provide an implementation for all three storage types. For transformative and quantitative operations, algorithms must provide an implementation for unknown and illife storages. For convolutive operations, which require spatial iterators and thus two-dimensional storages, algorithms must provide only an implementation that processes illife storages which contain unknown row storages. Implementations for contiguous storages are not required

Expression	Return Type	Notes
vvis::convolute(in, ia, out, oa, f)	void	Convolutive algorithm
vvis::for_each(in, ia, f)	void	Quantitative algorithm
vvis::transform(in, ia, out, a, f)	void	Transformative algorithm
vvis::transform(in1, ia1, in2, ia2, out, oa, f)	void	Transformative algorithm

Table 10. VVIS algorithms

because contiguous storages implement the unknown storage interface.

9. Functors

All VVIS functors must provide a scalar and a vvm::vector implementation, because data is processed using both the scalar processor and the VPU. Since VVM uses consistent functions for scalar and vvm::vector operations where applicable, it is possible to implement a single templated functor for both versions in many cases. Having two implementations for each functor prevents STL binders from working with VVIS functors. Thus VVIS also provides its own binders.

The three different categories have different functor requirements. Quantitative and transformative functors both accept input via operator(). Transformative functors are expected to return their answers, while quantitative functors keep their answer. Convolutive functors have a more complex interface because they accumulate input one pixel at a time, and then return the answer for that set of input. Tables 11, 12, and 13 summarises the functor requirements for the quantitative, transformative, and convolutive operations respectively.

Expression	Return Type	Notes
f(s)	Void	s is a scalar
f(v)	Void	v is a vector

Table 11. Quantitative functors' required functor interface

Expression	Return Type	Notes
f(s)	Output	s is a scalar
f(v)	Output	v is a vector

Table 12. Transformative functors' required functor interface

Expression	Return Type	Notes
A::kernel_type	Type	Type of a.kernel()
a.reset()	void	Signals start of new rectangle input
a.accumulate(s)	void	Accumulate a scalar
a.accumulate(v)	void	Accumulate a vector
a.scalar_result()	scalar	Returns the scalar answer
a.vector_result()	vector	Returns the vector answer
a.kernel_width()	const int	Returns the width of the kernel
a.kernel_height()	const int	Returns the height of the kernel

Table 13. Convolutive functors' required functor interface

10. Performance

To give readers some idea on the performance of the generic, vectorised, machine-vision library, VVIS, runtime performance results are presented in this section.

Figure 6 compares the runtime performance of VVIS's transformative algorithm to hand-coded programs, in scalar and in AltiVec mode. All results were collected on an Apple

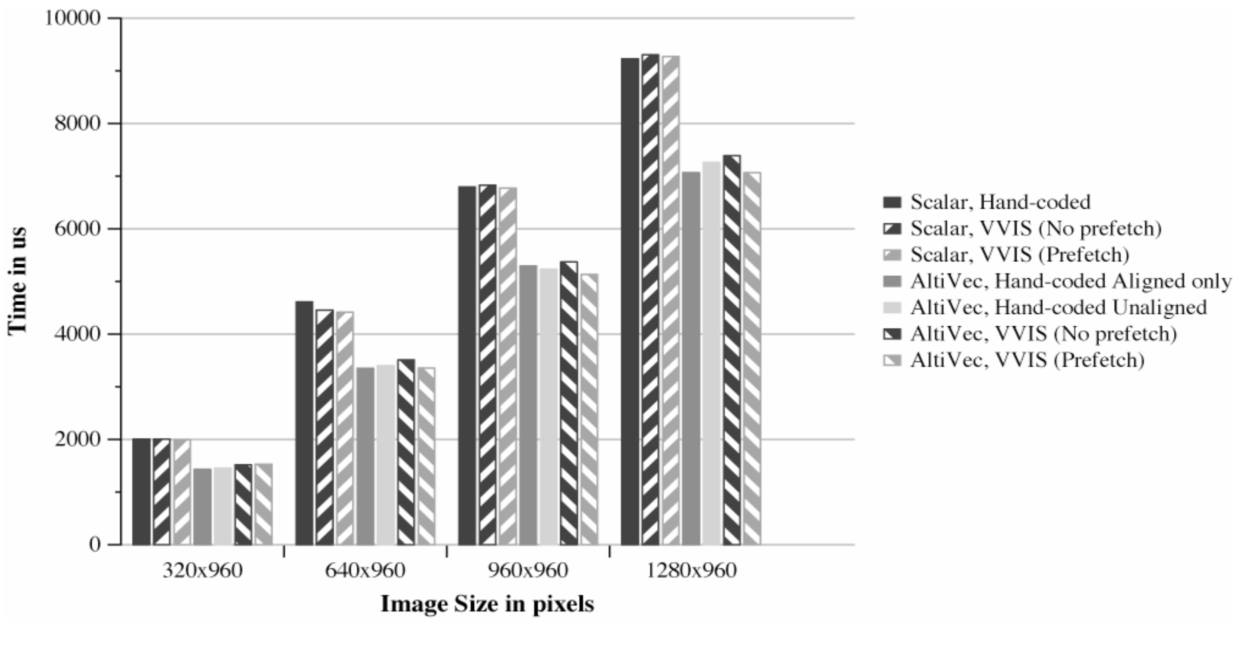


Figure 6. Performance of different vvis::transform implementations when the source and destination single-channel images are different, and the functor adds input values to themselves

PowerBook G4 with one PowerPC 7447A 1.3GHz with 32K L1 instruction cache, 32K L1 data cache, and 512K L2 cache, from programs that were compiled with the -Os (optimise for size) switch. Each result shown is the lowest time obtained after 20 runs. In each run, the source and destination images were allocated, and the source image was filled with data, after which vvis::transform was timed. The functor used added each input value to itself. All source and destination images used were aligned. The hand-coded scalar and AltiVec programs are based on the programs presented in (Lai *et al.*, 2002).

Figure 6 shows that for single-channel, unsigned char images, VVIS is comparable with both scalar and AltiVec hand-coded programs. For unsigned char operations, the VVM implementation used had no significant overheads.

11. Conclusion

Existing generic libraries, such as STL and VIGRA, are difficult to vectorise because iterators do not provide algorithms with information on how data are arranged in memory. Without this information, the algorithm cannot decide whether to use the scalar processor or the VPU to process the data. A generic, vectorised library needs to consider how functors invoke VPU instructions, how algorithms access vectors efficiently, and how edges, unaligned data, and prefetching are handled. The generic, vectorised, machine-vision library design presented in this paper addresses these issues.

The functors access the VPU through an abstract VPU. An abstract VPU is a virtual VPU that represents a set of real VPUs through an idealised instruction set and common constraints. The implementation used has no significant overheads in scalar mode, and for char types in AltiVec mode. Functors must also provide two implementations, one for the scalar processor and one for the VPU. This is necessary because the solution proposed uses both the scalar processor and the VPU to process data.

Since VPU programs are difficult to implement efficiently, a categorisation scheme based on input-to-output correlation was used to reduce the number of algorithms required. Three categories were specified for VVIS: quantitative, transformative and convolutive. Quantitative operations require one input element per input set to produce zero or more output elements per output set. Transformative operations are a subset of quantitative and convolutive operations, requiring one input element per input set to produce one output element per output set. Convolutive operations accept a rectangle of input elements per input set to produce one output element per output set.

Storages provide information on how data are arranged in memory to the algorithm, allowing the algorithm to automatically select appropriate implementations. Three main storage types were specified: contiguous, unknown or illife. Contiguous and unknown storages are one-dimensional while illife storages are n-dimensional storages. Only contiguous storages are expected to be processed using the VPU. Two types of contiguous storages were also specified: contiguous aligned storages, and contiguous unaligned storages. The iterator returned by begin() is always aligned for contiguous aligned storages, but may be unaligned for contiguous unaligned storages. Different algorithm implementations are required for different storage types. To support processing of different storage types simultaneously, storage types are designed to be subsets of one another. This allows an algorithm to gracefully degrade VPU usage and to provide efficient performance in the absence of VPUs.

Edges are handled by an iterator with orthogonal scalar and vvm::vector components. The contiguous storage's iterator allows traversal using scalar, vvm::vector or pixel steps. Scalar steps change only the scalar component and are only valid if the iterator does not cross a vvm::vector boundary. Vvm::vector steps change only the vvm::vector component. Pixel steps can change both the scalar and vvm::vector components. The orthogonal components make handling edges easy: once the vvm::vector component is exhausted, only the edge remains.

Contiguous unaligned storages allow algorithms to handle unaligned data directly. To facilitate such implementations, all contiguous storages ensure that the last edge can be loaded using the VPU without triggering memory exceptions. In cases where an algorithm that processes unaligned data is too difficult to implement, contiguous unaligned storages can be converted to contiguous aligned storages.

Prefetching is delegated to accessors. Accessors handle prefetching because only accessors know which data are actually going to be used. For example, an accessor might provide access to only the red channel of a RGB image, and thus it should only prefetch the red channel.

Runtime performance results from a transformative algorithm are presented. These results show that with an abstract VPU with no significant overheads, the solution is comparable in performance to hand-coded programs in both scalar and AltiVec mode.

12. References

Apple Computer Inc. (2002), *Code Optimization*. http://developer.apple.com/hardware/ve/code_optimization.html

Apple Computer Inc. (2003), *PowerPC G5 Performance Primer*. http://developer.apple.com/technotes/tn/tn2087.html

Apple Computer Inc. (2004), *Performance Issues: Memory Usage*. http://developer.apple.com/hardware/ve/performance_memory.html

Georgia Tech Research Corporation (2001), *VSIPL 1.01 API*. http://www.vsipl.org/PubInfo/vsiplv1p01_final1.pdf

Geraud, T., Fabre, Y., Duret-Lutz, A., Papadopoulos-Orfanos, D. & Mangin, J.-F. (2000), Obtaining genericity for image processing and pattern recognition algorithms, *in* 'Pattern Recognition, 2000. Proceedings. 15th International Conference on'.

Köethe, U. (1998), 'On data access via iterators (working draft)'. http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/doc/documents/DataAccessors.ps

Köethe, U. (1999), 'Reusable software in computer vision', *B. Jähne, H. Haussecker, P. Geissler: "Handbook on Computer Vision and Applications"* **3**. http://kogs-www.informatik.uni-hamburg.de/~koethe/papers/handbook.ps.gz

Köethe, U. (2000*a*), 'Expression templates for automated functor creation'. http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/doc/documents/FunctorFactory.ps

Köethe, U. (2000*b*), Generische Programmierung für die Bildverarbeitung, PhD thesis, Universität Hamburg. http://kogs-www.informatik.uni-hamburg.de/~koethe/papers/DissPrint.ps.gz

Köethe, U. (2000*c*), 'STL-style generic programming with images', *C++ Report Magazine* pp. 24–30.http://kogs-www.informatik.uni-hamburg.de/~koethe/papers/GenericProg2DC++Report.ps.gz

Köethe, U. (2001), *VIGRA Reference Manual for 1.1.2*. http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/doc/index.html

Kühl, D. & Weihe, K. (1997), 'Data access templates', *C++ Report* 9/7, 15, 18–21.

Lai, B.-C. (2004), Vector processor software for machine vision, PhD thesis, University of Wollongong. Under Examination.

Lai, B.-C. & McKerrow, P. J. (2001), Programming the velocity engine, *in* N. Smythe, ed., 'e-Xplore 2001: a face to face odyssey', Apple University Consortium.

Lai, B.-C., McKerrow, P. J. & Abrantes, J. (2002), Vectorized machine vision algorithms using AltiVec, Australasian Conference on Robotics & Automation.

Lai, B.-C., McKerrow, P. J. & Abrantes, J. (2005), 'The abstract vector processor', *Microprocessors and Microsystems* . Under Review.

Mittal, M., Peleg, A. & Weiser, U. (1997), 'MMX technology architecture overview', *Intel Technology Journal '97* .

Motorola Inc. (1999), *AltiVec Technology Programming Interface Manual*. http://e-www.motorola.com/brdata/pdfdb/microprocessors/32_bit/powerpc/altivec/altivecpim.pdf

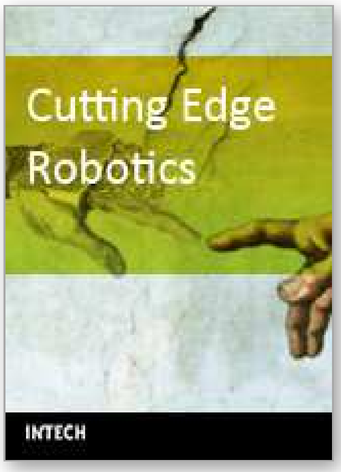
Musser & Stepanov (1989), Generic programming, *in* 'ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation. http://citeseer.nj.nec.com/musser88generic.html

Ollmann, I. (2001), Altivec.http://www.alienorb/AltiVec/Altivec.pdf

VSIPL website (2001). Maintained by Dr. Mark Richards of Georgia Tech Research Institue for DARPA and U.S. Navy.

Weihe, K. (2002), 'Towards improved static safety: Expressing meaning by type', *C/C++ Users Journal* pp. 32,34–36,38–40.

Weiser, U. (1996), 'MMX technology extension to the Intel architecture', *IEEE Micro* pp. 42–50.



Cutting Edge Robotics
Edited by Vedran Kordic, Aleksandar Lazinica and Munir Merdan

ISBN 3-86611-038-3
Hard cover, 784 pages
Publisher Pro Literatur Verlag, Germany
Published online 01, July, 2005
Published in print edition July, 2005